Towards the Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems

Petru Florin Mihancea and Radu Marinescu
Conference on Software Maintenance and Reengineering (2005)

Presented by
AhRim Han

2009. 6. 3

© KAIST SE LAB 2009
Contents

- Introduction
- Related work
- Detection strategy tuning machine
- Case study
- Conclusion & future work
The flaws of the design structure of an object-oriented software have a strong negative impact:
- Increase bug-fixing and maintenance costs
- Degrade software quality (e.g., the maintainability and flexibility)

Thus, identification and detection of design flaws is essential for evaluation and improvement of software quality.
Previous studies for design flaws detection of the object-oriented software

- A large number of metrics were used and combined in order to quantify a particular design flaw
  - Example

\[
ToSplit(S) = S' \quad \begin{cases} 
S' \subseteq S, \forall C \in S' \\
(TCC(C), \text{LowerThan}(0.2)) \wedge \\
(NOM(C), \text{HigherThan}(7))
\end{cases}
\]

- Two important questions need to be resolved to use this detection strategy
  - What set of metrics will be used?
  - How to determine the proper thresholds for those metrics?

It is oftentimes empirical and unreliable.
We strongly need automatic approach for design flaws detection, because

- The accuracy of the design flaws detection is heavily dependent on the proper selection of threshold values
- It is very difficult to only rely on informed human intuitions when analyzing a large software system

In this paper, they present the “tuning machine”

- A novel method for establishing proper threshold values for a detection strategy
  - Based on inferring the threshold values based on a set of reference (flawed and healthy) examples
Related work

- Improving the accuracy for detection strategy
  - Find the proper threshold values for software metrics based on experience and hints from literature
      - Guided by similar past experiences and by hints from metrics’ authors
      - Highly empirical
  - Combine detection strategies applied on a single version with additional information about the history of the system
      - Use time perspective such as the information about the change stability of a class or the persistency of a design flaw over time

Detection strategy tuning machine (1/4)

- Describes a generic methodology which helps
  - To find the proper threshold values of a detection strategy
  - To estimate its overall accuracy by tuning the thresholds

Background

Definitions

- Detection strategy
  - A model of the quantified design flaw inferred from a set of examples and consistent with those examples
  - Example

\[
\text{GodClass}(S) = S' \mid S' \subseteq S, \forall C \in S' \quad (\text{ATFD}(C), \text{TopValues}(20\%)) \land (\text{ATFD}(C), \text{HigherThan}(4)) \land \\
((\text{WMC}(C), \text{HigherThan}(20)) \lor (\text{TCC}(C), \text{LowerThan}(0.33))) / 20
\]
Detection strategy tuning machine (2/4)

Background (Cont’d)

- **Definitions**
  - **Positive examples of a design flaw**
    - Design entity (e.g., class) affected by that particular flaw
  - **Negative examples of a design flaw**
    - Design entity (e.g., class) *not* affected by that particular flaw
  - **False negative for a detection strategy**
    - Positive examples of the quantified design flaw which is not identified as such by the strategy
    - Not consistent with positive examples
  - **False positive for a detection strategy**
    - *Negative* examples of the quantified design flaw which is not identified as such by the strategy
    - Not consistent with *negative* examples
Main idea for tuning thresholds of a detection strategy

- Maximize the accuracy of the classification of the positive and negative examples
  - In other words, resulted tuned strategy is consistent with the largest possible number of positive and negative examples of the design flaw quantified by this detection strategy
Main steps:
1) Collect the examples
2) Tuning and validation
3) Try a trade-off
Step 1. Collect the examples

Contain informal description of many object-oriented design problems collected from literature

- 7 object-oriented software systems written in Java
- Each of the system has approximately 30 classes
- Developed during SE course by students with little experience in OO design

Mediate some inter-report conflicts

- 11 volunteers who are senior students and had followed an advanced object-oriented design course analyzed these programs
- Design entities that were reported as being affected by a particular design flaw were marked as positive examples
- Remaining entities were marked as negative examples
Tuning

- Genetic algorithm is used to find the threshold values of a detection strategy
  - Chromosome (or potential solution)
    - Encodes one threshold for each elementary term of the strategy to be tuned
  - Fitness function
    \[ f(X) = A \times b \times Fn\_No(X) + A \times (1 - b) \times Fp\_No(X) + 1 \]
    - \( X \) is the potential solution
    - \( Fn\_No(X)/Fp\_No(X) \) is the number of false negatives/positives obtained when the tuned strategy is applied on the tuning set using the threshold values encoded in \( X \)
    - \( A \) is a system parameter called penalty amplitude; it helps to adjust the value of the penalty for one inconsistency produced by \( X \)
    - \( b \) is a system parameter called balance (range: 0~1)
Validation

- Evaluate the accuracy of the resulted detection strategy
  - How many false negatives have occurred
    - The number of positive examples missed
  - How many false positives have occurred
    - The number of negative examples which viewed as positive ones

Tuning and the validation can be repeated multiple times

- When a cross validation technique is applied
Step 3. Try a trade-off

- This step is needed when the tuning set contains many conflicting positive and negative examples
  - No set of thresholds that simultaneously ensure the consistency of the strategy for both side of examples
    - The obtained strategy will miss many real flaws or introduce many false positives

- This balance feature can repeatedly search for an acceptable trade-off
  - Reducing the number of false negatives while increasing the number of false positives

- If we cannot find a fair one, we have to give up and correct the skeleton of the strategy
Case study (1/4)

- Tuning GodClass detection strategy
  - GodClass design flaw
    - Classes which tend to centralize the intelligence of a software system
    - Deviates from an object-oriented design heuristic which states that the intelligence of a system has to be uniformly distributed among the top-level classes of the system design
  - Skeleton of this detection strategy

\[
GodClass(S) = S' \mid S' \subseteq S, \forall C \in S' \\
(ATFD(C), TopValues(\_)) \land (ATFD(C), HigherThan(\_)) \land \\
(WMC(C), HigherThan(\_)) \lor (TCC(C), LowerThan(\_))
\]

- Access to Foreign Data (ATFD) – to detect for classes which use data from other classes
- Weighted Method Count (WMC) – to detect for classes which are complex
- Tight Class Cohesion (TCC) – to detect for classes which have low cohesion between methods
Results

Accuracy

• Tuning set
  – Before tuning process:
    » 6 false negatives/ 4 false positives occurred
  – After tuning process:
    » 1 false negatives/ 2 false positives occurred

• Validation set
  – Before tuning process:
    » Initial threshold hinders the detection of the positive example
  – After tuning process:
    » This positive example is detected; one false positive appears

<table>
<thead>
<tr>
<th>Set type</th>
<th>Positive examples</th>
<th>Negative examples</th>
<th>Using original thresholds</th>
<th>Using tuned thresholds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>False negatives</td>
<td>False positives</td>
</tr>
<tr>
<td>Tuning</td>
<td>9</td>
<td>142</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Validation</td>
<td>1</td>
<td>22</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Results (Cont’d)

- New thresholds

More than one set of thresholds which produce the same accuracy as presented in the previous page’s table

If we choose to use the solution identifier 3, then

\[
\text{GodClass}(S) = S' \left\| \begin{array}{l}
S' \subseteq S, \forall C \in S' \\
(\text{ATFD}(C), \text{TopValues}(18\%)) \land (\text{ATFD}(C), \text{HigherThan}(2)) \land \\
((\text{WMC}(C), \text{HigherThan}(40)) \lor (\text{TCC}(C), \text{LowerThan}(0.33)))
\end{array} \right.
\]
Results (Cont’d)

- New thresholds

Cluster 2 can capture true particularities of the GodClass design flaw.

Cluster 1 is likely affected by the overfitting phenomenon (probably due to lack of enough examples).

Scatter diagram for two terms above
Issues for applying the approach

- Collecting positive and negative examples is hard and time consuming
  - But, it’s not impossible!
    - If one really wants to do it, s/he can

- More than one sets of thresholds can have same accuracy on the tuning and validation sets
  - Which one to use in order to detect that design flaw?
    - Possible solution: use all of them obtaining a multiple thresholds detection strategy
      - Repeat the analysis for each set of thresholds
      - Entities detected as being flawed for at least one time can be ordered depending on how many threshold sets have detected them
Conclusion and future work

**Conclusion**
- Provide a method to determine threshold values used by a detection strategy
  - Can find the proper threshold values and improve the accuracy of that detection strategy

**Future work**
- Collect examples for other design flaws
  - ShotgunSurgery, RefusedBequest, etc.
Thank You.