Investigating the Impact of Design Debt on Software Quality

Prioritizing Design Debt Investment Opportunities

Nico Zazworka

Carolyn Seaman, Forrest Shull, Michele Shaw

Fraunhofer USA Center for Experimental Software Engineering

UMBC An Honors University in Maryland
Design Debt
Potential Indicators

Code Decay

Code Metrics

Architecture Violations

Lack of Design Patterns

God Class

Data Class

Code Smells

Code Clones

Tradition Breaker

Intensive Coupling
Research Questions

• Are Code Smells, i.e. God Classes, valid indicators for design debt?
  – Do God Classes have a negative impact on:
    • Maintainability and
    • Correctness

• Can we give advice on which design debt to pay first?
  – Which God Classes are easy to fix and promise high gain in software quality?
  – Which God Classes are hard to fix and promise low gain in software quality?
The God Class

• Also known as “Large Class” [Fowler99]

• Marinescu [Mar04]
  – Centralizes intelligence
  – Multiple responsibilities
  – Delegates minor detail
  – Uses data of other classes
God Class Detection

ATFD > 5
Access to foreign data

WMC > 46
Weighted method count

TCC < 0.33
Tight class cohesion

AND

GOD CLASS
Case Study

• Small software development company
  – 30 employees: C# developers, web-designers
  – 2 active development projects
    • Project J: 35kLOC, 11 months, 4 developers
    • Project F: 45kLOC, 17 months, 4 developers

• Previously performed a code smell study in the same environment

• Small part of developers were familiar with technical debt metaphor

• Data: subversion repository and JIRA bug tracker
God Classes and Maintainability

• Assumption: maintainability can be estimated by investigating how often a class to be changed
  – Rational: classes that have to be changes too often, e.g. with each revision, are indicators for maintenance bottlenecks

• H1: The change likelihood of god classes is higher than for non-god classes

<table>
<thead>
<tr>
<th>Revision</th>
<th>1452</th>
<th>1457</th>
<th>1471</th>
<th>1472</th>
<th>1424</th>
<th>Likelihood</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changed God Classes</td>
<td>0/4</td>
<td>1/4</td>
<td>1/4</td>
<td>2/4</td>
<td>2/4</td>
<td>0.300</td>
</tr>
<tr>
<td>Changed Non-God Classes</td>
<td>1/223</td>
<td>4/223</td>
<td>6/225</td>
<td>4/225</td>
<td>2/225</td>
<td>0.015</td>
</tr>
</tbody>
</table>

Example for change likelihood for god classes and non-god classes in project F
### Maintainability Results

- God classes are 5-7 times more change prone
- Do we need to normalize this data by size?

<table>
<thead>
<tr>
<th></th>
<th>Project F</th>
<th></th>
<th>Project J</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>God Classes</td>
<td>Non-God Classes</td>
<td>God Classes</td>
<td>Non-God Classes</td>
</tr>
<tr>
<td>N</td>
<td>545</td>
<td>658</td>
<td>282</td>
<td>328</td>
</tr>
<tr>
<td>mean</td>
<td>0.07848</td>
<td>0.01619</td>
<td>0.12565</td>
<td>0.01725</td>
</tr>
<tr>
<td>s</td>
<td>0.18448</td>
<td>0.03837</td>
<td>0.24754</td>
<td>0.02391</td>
</tr>
<tr>
<td>p-value</td>
<td>4.282e-14</td>
<td></td>
<td>2.461e-12</td>
<td></td>
</tr>
</tbody>
</table>
Investigating Normalization

• Assumption: “A class that is twice as large, is twice as change prone.”

• Method: Measure correlation between:
  – Size (LOC)
  – Change Likelihood

• Results (Pearson CC):
  – Project F: -0.029
  – Project J: 0.42

• Dividing by LOC might over-normalize result
  – Project J normalized result still statistically significant
God Classes and Defects

- H2: The defect likelihood of god classes is higher than for non-god classes
- Data: JIRA bugs are linked to subversion change sets (=classes that were part of the bug fix)

<table>
<thead>
<tr>
<th>Defect (JIRA issue)</th>
<th>J-166</th>
<th>J-161</th>
<th>J-377</th>
<th>J-396</th>
<th>J-228</th>
<th>Likelihood</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fix Revisions</td>
<td>9097, 9098</td>
<td>8939</td>
<td>11990</td>
<td>12842, 12844</td>
<td>10269</td>
<td></td>
</tr>
<tr>
<td>God Classes</td>
<td>1/3</td>
<td>0/1</td>
<td>0/8</td>
<td>3/8</td>
<td>0/3</td>
<td>0.1417</td>
</tr>
<tr>
<td>Non-God Classes</td>
<td>0/94</td>
<td>1/94</td>
<td>1/156</td>
<td>0/157</td>
<td>1/101</td>
<td>0.0067</td>
</tr>
</tbody>
</table>

Example for defect fix likelihood for god classes and non-god classes in project J
Defect Results

- God classes are 4-17 times more defect prone
- Do we need to normalize this data by size?

<table>
<thead>
<tr>
<th></th>
<th>Project F</th>
<th>Project J</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>God Classes</td>
<td>Non-God Classes</td>
</tr>
<tr>
<td>N</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>mean</td>
<td>0.03939</td>
<td>0.00956</td>
</tr>
<tr>
<td>s</td>
<td>0.13669</td>
<td>0.01094</td>
</tr>
<tr>
<td>p-value</td>
<td>0.2276 (not sig.)</td>
<td>0.008217</td>
</tr>
</tbody>
</table>
Investigating Normalization

- Assumption: “A class that is twice as large, is twice as defect prone.”
- Method: Measure correlation between:
  - Size (LOC)
  - Defect Likelihood
- Results (Pearson CC):
  - Project F: 0.011
  - Project J: -0.018
- Dividing by LOC will over-normalize result
## Related Research

<table>
<thead>
<tr>
<th>Related Work</th>
<th>Investigated Software</th>
<th>God classes more change prone if not normalized? (p&lt;0.05)</th>
<th>God classes more change prone if LOC normalized? (p&lt;0.05)</th>
<th>God classes more defect prone if not normalized? (p&lt;0.05)</th>
<th>God classes more defect prone if LOC normalized? (p&lt;0.05)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Li 2007</td>
<td>Eclipse</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Olbrich 2009</td>
<td>Lucene, Xerces</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>(p&lt;0.05)</td>
</tr>
<tr>
<td>Schumacher 2010</td>
<td>Two commercial applications</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Olbrich 2010</td>
<td>Lucene, Xerces, Log4j</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Khomh 2009</td>
<td>Azereus, Eclipse</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Study results presented here</td>
<td>Two commercial applications</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
</tbody>
</table>
Paying Design Debt

• Moving from identifying TD to managing TD
• Paying off debt is an investment opportunity with tradeoffs:
  – Value of debt (how much is it going to cost to fix it?)
  – Interest rate (how much does it slow down development?)
  – Probability (what is the chance that the debt affects productivity?)
• Goal: select the most profitable opportunities, ignore non-profitable ones.
• Profitable (good cost/benefit ratio)
  – Low value
  – High interest rate
Cost of Paying Debt

- Refactoring
- Idea: facilitate metrics in detection model
- Argument: a class being close to the thresholds will be easier to refactor than one that is multiple magnitudes outside.
- Method: rank god classes according to their distance to the thresholds

ATFD > 5
Access to foreign data

WMC > 46
Weighted method count

TCC < 0.33
Tight class cohesion

AND

GOD CLASS
## God Class Ranking: Cost

<table>
<thead>
<tr>
<th>God Class Name</th>
<th>WMC (&gt;46) Value</th>
<th>WMC (&gt;46) Rank</th>
<th>TCC (&lt;0.33) Value</th>
<th>TCC (&lt;0.33) Rank</th>
<th>ATFD (&gt;5) Value</th>
<th>ATFD (&gt;5) Rank</th>
<th>Overall Score and Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>GodClass1</td>
<td>49</td>
<td>3</td>
<td>0.0</td>
<td>8</td>
<td>20</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>GodClass2</td>
<td>87</td>
<td>8</td>
<td>0.005</td>
<td>7</td>
<td>28</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>GodClass3</td>
<td>107</td>
<td>9</td>
<td>0.0</td>
<td>8</td>
<td>28</td>
<td>7</td>
<td>24</td>
</tr>
<tr>
<td>GodClass4</td>
<td>69</td>
<td>7</td>
<td>0.026</td>
<td>6</td>
<td>34</td>
<td>9</td>
<td>22</td>
</tr>
<tr>
<td>GodClass5</td>
<td>49</td>
<td>3</td>
<td>0.065</td>
<td>5</td>
<td>9</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>GodClass6</td>
<td>60</td>
<td>5</td>
<td>0.177</td>
<td>4</td>
<td>19</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>GodClass7</td>
<td>47</td>
<td>1</td>
<td>0.219</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>GodClass8</td>
<td>48</td>
<td>2</td>
<td>0.199</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>GodClass9</td>
<td>61</td>
<td>6</td>
<td>0.192</td>
<td>3</td>
<td>19</td>
<td>4</td>
<td>13</td>
</tr>
</tbody>
</table>
God Class Ranking: Interest

• **Interest**: negative effect on software quality
  – Maintainability
  – Defects

• **Method**: use change and defect likelihood to estimate and rank impact

<table>
<thead>
<tr>
<th>God Class Name</th>
<th>Change Likelihood Value</th>
<th>Change Likelihood Rank</th>
<th>Defect Likelihood Value</th>
<th>Defect Likelihood Rank</th>
<th>Overall Score and Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>GodClass1</td>
<td>0.016</td>
<td>1</td>
<td>0.0</td>
<td>1</td>
<td>2 1</td>
</tr>
<tr>
<td>GodClass2</td>
<td>0.097</td>
<td>8</td>
<td>0.0</td>
<td>1</td>
<td>9 4</td>
</tr>
<tr>
<td>GodClass3</td>
<td>0.102</td>
<td>9</td>
<td>0.029</td>
<td>5</td>
<td>14 9</td>
</tr>
<tr>
<td>GodClass4</td>
<td>0.068</td>
<td>7</td>
<td>0.177</td>
<td>6</td>
<td>13 7</td>
</tr>
<tr>
<td>GodClass5</td>
<td>0.040</td>
<td>3</td>
<td>0.0</td>
<td>1</td>
<td>4 3</td>
</tr>
<tr>
<td>GodClass6</td>
<td>0.0455</td>
<td>4</td>
<td>0.133</td>
<td>7</td>
<td>11 5</td>
</tr>
<tr>
<td>GodClass7</td>
<td>0.0458</td>
<td>5</td>
<td>0.133</td>
<td>7</td>
<td>12 6</td>
</tr>
<tr>
<td>GodClass8</td>
<td>0.052</td>
<td>6</td>
<td>0.133</td>
<td>7</td>
<td>13 7</td>
</tr>
<tr>
<td>GodClass9</td>
<td>0.027</td>
<td>2</td>
<td>0.0</td>
<td>1</td>
<td>3 2</td>
</tr>
</tbody>
</table>
Cost/Benefit Matrix
Future Work

• Evaluation of other code smells and other indictors
• Empirical evaluation of cost/benefit model
  – Are our assumptions on correlation of metrics and refactoring cost true?
  – Are god classes after refactoring indeed less change and defect prone?
  – Can we advance from a ranking to a more precise prediction model?
• Managing design debt and god classes:
  – When should a god class be refactored?
  – When is it acceptable to introduce a god class for short term gains?
QUESTIONS?

Dr. Nico Zazworka
Research Scientist

Center for Experimental Software Engineering
University of Maryland
Phone: 240 487 2928
Email: nzazworka@fc-md.umd.edu