An experience report on using code smells detection tools

Francesca Arcelli Fontana, Elia Mariani
Andrea Morniroli, Raul Sormani, Alberto Tonello

RefTest – May 25, 2011
Berlin
Outline

• Code smells and refactoring
• Code smells detection tools
• Experiment set up
• Experiment results
• Lessons learned
• Future developments
Code smells and Refactoring

- Smells: characteristics of software that may indicate a design problem [Fowler 99].
- Taxonomies [Wake03, Mantyla 03, Keriesky 04, Lanza 06, Moha 10]
- Refactoring: a disciplined technique for restructuring an existing body of code, Refactoring alters the internal structure of code without changing its behavior.
- A smell is a symptom, refactoring is a cure
- Use the right refactoring for the right issue
  - A same symptom may have different cures
  - Thus, every smell may suggests more than one refactoring
Code smells and Refactoring

- Not necessarily all the code smells have to be removed
- It is better to remove them as early as possible. Tool support is important, since many code smells can go unnoticed while programmers are working.
- Code smells are not formally defined, hence their detection can provide uncertain and unsafe results (tailoring of the definitions)
- The detection techniques used by the tools are usually based on the computation of a particular set of combined metrics, or standard object-oriented metrics, or metrics defined ad hoc for the smell detection purpose (define the threshold values for these metrics).
- The aim with detection strategies is to make design rules (and their violations) quantifiable, and thus to be able to detect design problems in an object-oriented software system, i.e., to find those design fragments that are affected by a particular design problem.
Code smell detection tools

- **JDeodorant**: Feature Envy, God Class, Long Method, Type Checking
- **PMD**: Long Method, Large Class, Long Parameter List, Duplicate Code, Dead Code
- **Stench Blossom**: Data Clumps, Long Method, Large Class, Feature Envy, Message Chain, (8 smells, detecting by reading the code)
- **iPlasma**: Brain Class, Brain Method, Data Class, Feature Envy, God Class, Extensive Coupling, Feature Envy, God Class, ..
- **inFusion**: Brain method, Data Class, Feature Envy, God Class, Refused Parent Bequest, Significant Duplication
- **Decor**: Lazy Class, Long Method, Large Class, Long Parameter List, Refused Bequest, Speculative Generality, Message Chains, Data Class, Tradition Breaker (29 code and design smells)
- **CheckStyle**: Long Method, Long parameter list, Large Class, Dupl.Code
- **Codecity**: Feature Envy, Brain Class, God Class, Data Class, Brain Method, Intensive Coupling, Dispersed Coupling, Shotgun Surgery
- **CodeVizard**: God Class, Shotgun Surgery, ..
Considerations on code smell detection tools

- Automatic Reafactoring: Jdeodorant
- Link to the Code: Jdeodorant, PMD, Stench Blossom, JDeodorant
- Other languages (Java): InFusion(C, C++), iPlasma (C++)
- System size computation
- Forward engineering(development): Stench Blossom
- Computation of the number of the smells: Stench Blossom does not provide numerical values, but only a visual threshold, the size of a petal is directly proportional to the entity of the code smells. The only possible procedure to find code smells is to manually browse the source code, looking for a petal whose size is big enough to make us suppose that there is a code smell.
Detecting smells and Metrics computation

• How to fix **thresholds** of the metrics?
• Define for each metric a filter that captures best the symptom that the metric is intended to quantify.
  – (1) pick-up a comparator and
  – (2) to set an adequate threshold, in conformity with its semantics.

• Setting thresholds can be an effective approach for detecting **domain-specific code** smells. The healthiness of a module in a domain may depend on the **semantic** information it has, such as the functionality or the logic that the module implements. For example:

• Brain Method smell is a method that contains too many conditional branches. Although there are design solutions that can reduce the use of conditional branches, they may not satisfy other design constraints. In some cases using conditional branches can be the best choice among the design alternatives.

• Intensive coupling, but the “the coupling degree of a method” can be considered acceptable for the system in a specific domain. In other words, we can think that such degree of coupling would not cause maintenance problems.
Experiment set up

GanttProject System

<table>
<thead>
<tr>
<th></th>
<th>1.10</th>
<th>1.10.1</th>
<th>1.10.2</th>
<th>1.10.3</th>
<th>1.11.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packages</td>
<td>27</td>
<td>27</td>
<td>27</td>
<td>28</td>
<td>33</td>
</tr>
<tr>
<td>Classes</td>
<td>393</td>
<td>395</td>
<td>396</td>
<td>402</td>
<td>549</td>
</tr>
<tr>
<td>Methods</td>
<td>1913</td>
<td>1921</td>
<td>1924</td>
<td>1937</td>
<td>2724</td>
</tr>
</tbody>
</table>

No standard benchmark for tool’s comparison is available, the authors of DECOR provided the precision and recall of a detection technique on an open-source system (multi application platform for planning and project management) and on a representative set of six smells. Manual validation could be subjective.
Experiment results- God Class

- The Code smell Large Class detected by DECOR is equivalent to the God Class smell recognized by JDeodorant, inFusion and iPlasma. In fact, both specifications define classes that are trying to do too much... These classes also have low cohesion.

1. They heavily access data of other simpler classes, either directly or using accessor methods.
2. They are large and complex
3. They have a lot of non-communicative behavior i.e., there is a low cohesion between the methods belonging to that class.
God Class detection

1. Class uses directly more than a few attributes of other classes.
   Since ATFD (Access to Foreign Data) measures how many foreign attributes are
   used by the class, it is clear that the higher the ATFD value for a class, the higher is
   the probability that a class is (or is about to become) a God Class (the extent to
   which a class uses attributes of other classes).
   ATFD > a maximun number of tolerable foreign attributes to be used (FEW where
   FEW is considered as [2-5]).

2. Functional complexity of the class is very high.
   This is expressed using the WMC (Weighted Method Count) metric: sum of the
   statical complexity of all methods in a class (consider McCabe’s cyclomatic
   complexity metric as a complexity measure \(\Rightarrow\) GREATER THAN VERY HIGH

3. Class cohesion is low. As a God Class performs several distinct functionalities
   involving disjunct sets of attributes, this has a negative impact on the class’s
   cohesion.
   Tight Class Cohesion (TCC) is the relative number of methods directly connected
   via accesses of attributes \(\Rightarrow\) LESS than...1/3.. ([0..33]), less than one-third of the
   method pairs have in common the usage of the same attribute.
Experiment results

• As an example, the following expression evaluates if the “God class” smell is present:

\[(WMC \geq 47) \land (ATFD > 5) \land (TCC < 1/3)\]

• where WMC, weighted methods per class, ATFD, number of accesses to foreign class data and TCC tight class cohesion.

• For Jdeodorant the God Class is detected considering the number of line of code.
# Metrics theresholds examples -iPlasma

<table>
<thead>
<tr>
<th>Metric</th>
<th>LOW</th>
<th>AVE</th>
<th>HIGH</th>
<th>VERY HIGH</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC/Class</td>
<td>28</td>
<td>70</td>
<td>130</td>
<td>195</td>
</tr>
<tr>
<td>WMC</td>
<td>5</td>
<td>14</td>
<td>31</td>
<td>47</td>
</tr>
<tr>
<td>LOC/Method</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>NOM/Class</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>
Experiment results

• God Class

Decor-Benchmark: 9 God class

Table 9: God Class comparisons

<table>
<thead>
<tr>
<th></th>
<th>1.10</th>
<th>1.10.1</th>
<th>1.10.2</th>
<th>1.10.3</th>
<th>1.11.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDeodorant</td>
<td>7 (1,78%)</td>
<td>6 (1,51%)</td>
<td>8 (2,02%)</td>
<td>5 (1,24%)</td>
<td>22 (4,03%)</td>
</tr>
<tr>
<td>InFusion</td>
<td>11 (2,79%)</td>
<td>11 (2,78%)</td>
<td>11 (2,77%)</td>
<td>11 (2,73%)</td>
<td>13 (2,38%)</td>
</tr>
<tr>
<td>iPlasma</td>
<td>10 (2,54%)</td>
<td>10 (2,53%)</td>
<td>10 (2,52%)</td>
<td>10 (2,48%)</td>
<td>13 (2,38%)</td>
</tr>
</tbody>
</table>

Figure 4: God Class
Experiment results - Long Parameter List

- PMD: 65 LPL with 4 parameter  (DECOR: 54 LPL)

An experience report on using code smells detection tools

![Long Parameter list Comparison](image-url)
Experiment results - Large Class

- PMD – Line of code 415 (DECOR-9 Large Class)

![Large Class graph](image-url)

An experience report on using code smells detection tools
Experiment results - Code smell evolution

Figure 12: Global System Evolution
Lesson learned

- A comparison of the tools is a complex task!
  - Computing the size of the system, detection techniques, the metrics thresholds, refactoring, link to the code, tool interface,..
- We aim to propose a new taxonomy of smells based on refinements of smells definition, clear definition of the detection techniques (necessary and sufficient conditions)
- We gained useful information to be exploited in a tool for code smell detection.
An experience report on using code smells detection tools
Marple-Code smells Detector

Currently we are able to detect the following smells:

1) Feature Envy (Jdeodorant, iPlasma, inFusion, Marple)
2) Indecent Exposure (Marple)
3) Long Parameter List (Checkstyle, PMD, iPlasma, Marple)
4) Refused Bequest (iPlasma, Marple)
5) Data Class (iPlasma, inFusion, Marple)
6) Lazy Class (Marple)

Future smells to detect:
Comments, Duplicate Code

We are interested to provide the same validation for precision/recall on GanttProject system.
Future works

• Extend the manual validation for other smells to improve comparison
• Extend the experimentation with other tools as CodeVizard, Checkstyle,..and on different system as Ant,.. .
• Improve the Marple Code Smell Detector tool through the *lessons learned* in particular:
  • provide a clear detection technique and clear metrics thresholds
  • provide a metrics-based approach that incorporates tailored, domain-specific detection rules.
• Analyze the impact of refactoring to remove code smells on some well known object-oriented quality metrics [Arcelli, Spinelli WRT 2011, ICSE Workshop]
Future works-Impact of Refactoring

- Our approach is based on the following steps:
  1. computing the metrics on a software system
  2. detecting the code smell
  3. automated refactoring for one smell
  4. reapplying step 1
  5. analysing the increment/decrement for each metric value
  6. reapplying step 3-5 for each smell
  7. identifying the smells, whose refactoring better improves the quality of the analysed system.

- We have considered metrics proposed in the literature to measure cohesion, coupling and complexity of a system.
- We considered Feature Envy, Long Method, Shotgun Surgery, Large Class
References


An experience report on using code smells detection tools
References


