Detecting Bad Smells in Source Code Using Change History Information

Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, Denys Poshyvanyk
Context

Code

Bad Smells

HIST

Historical Information for Smell Detection

Study Design

Results

Evaluation on eight systems
Context

Code

Bad Smells
An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension

Maurice Abbes1,4, Francis Khenfer2, Yona Gold Gofshen2, Giovanni Antoniol3
1 Dept. d'Informatique et de Recherche Opérationnelle, Université de Montréal, Montréal, Canada
2 Dept. of Elec. and Comp. Engineering, Queen's University, Kingston, Ontario, Canada
3 PIQ Team, SOCCER Lab, D agility, Ecole Polytechnique de Montréal, Montréal, Canada
Emails: maurice.abbes@umontreal.ca, founer.khenfer@queenu.ca
yona.gold.gofshen@polymtl.ca, antoniol@ieee.org

Abbes et al. CSMR 2011

Bad Smells hinder code comprehensibility

1 Introduction

Context: In theory, antipatterns are “poor” solutions to recurring design problems; they stem from experienced software development and describe common pitfalls in object-oriented programming, e.g., Brown’s 40 antipatterns (1). Antipatterns are generally introduced in systems by developers not having sufficient knowledge and experience in solving a particular design problem or having implemented some design patterns. Coplien (2) described an antipattern as “something that looks like a good idea, but which backfires badly, when applied.” In practice, antipatterns relate to and manifest themselves as code smells in the source code, symptoms of implementation and design problems (3).

An example of antipattern is the Blob class, which is widely used to encapsulate a complex idea in a single class. Maturity of a system is a factor in the detection and avoidance of antipatterns, as the experience of the developers is a key aspect in identifying and mitigating these problems. The presence of antipatterns can hinder code comprehensibility, which is crucial for software maintenance and evolution. Bad smells, such as poor code design or unclear documentation, can affect the understandability of the system, making it more difficult for new developers to work on the code. Therefore, it is essential to identify and address these issues to improve the overall quality and maintainability of the software.
Bad Smells increase change and fault proneness
Identification of Move Method Refactoring Opportunities

Nikolaos Tsantilis, Student Member, IEEE, and Alexander Chatzigeorgiou, Member, IEEE

Abstract—Placement of refactoring methods within classes in an object-oriented system is usually guided by conceptual criteria and decided by automatic heuristics. Merging data and behavior between classes can help reduce coupling and increase cohesion; but it is necessary to ensure that such refactoring should not impact on the system. Move Method refactoring opportunities exist as a way for solving many common Feature Envy problems. An approach that employs the notion of distance between system entities (attributes/methods) and classes extracts a list of behavior preserving refactoring based on the examination of a set of criteria. In practice, a software system may not have such problems in many different places. Therefore, our approach measures the effect of all refactoring suggestions based on a novel Entity Placement metric that quantifies how well entities have been placed in system classes. The analyzed methodology can be expanded as a semi-automatic approach since the designer will eventually choose whether a suggested refactoring should be applied or not based on conceptual or other design quality criteria. The evaluation of the proposed approach has been performed considering qualitative, metric, contextual, and efficiency aspects of the refactored systems in a number of open-source projects.

Index Terms—Move Method refactoring, Feature Envy, object-oriented design, Jaccard distance, data quality.

1 INTRODUCTION

A Common: several principles and laws of object-oriented design [18], [26], designers generally strive for low coupling and high cohesion. A number of empirical studies have investigated the relation of coupling and cohesion metrics with external quality indicators. [27] and [28] have shown that coupling metrics can serve as predictors of software quality classes. The researchers of [27] and [28] have shown that coupling metrics (as measured using the Leupold Dependency Metrics) correlate highly with maintenance costs and fewer maintenance problems. Consequently, low coupling and high cohesion are regarded as indicators of good quality design in terms of maintenance.

Coupling or cohesion problems manifest themselves in many different ways, with Feature Envy being the most common symptom. Feature Envy is a form of violating the principle of grouping behavior with related data and is often a sign of overuse of the interface for passing message/data to a method. Line of sight refactoring opportunities exist as a way for solving many common Feature Envy problems when methods are not used as intended and behave as if they were part of some other class. Move Method refactoring opportunities exist to provide a method for solving many common Feature Envy problems. An approach that employs the notion of distance between system entities (attributes/methods) and classes extracts a list of behavior preserving refactoring based on the examination of a set of criteria. In practice, a software system may not have such problems in many different places. Therefore, our approach measures the effect of all refactoring suggestions based on a novel Entity Placement metric that quantifies how well entities have been placed in system classes. The analyzed methodology can be expanded as a semi-automatic approach since the designer will eventually choose whether a suggested refactoring should be applied or not based on conceptual or other design quality criteria. The evaluation of the proposed approach has been performed considering qualitative, metric, contextual, and efficiency aspects of the refactored systems in a number of open-source projects.

Index Terms—Move Method refactoring, Feature Envy, object-oriented design, Jaccard distance, data quality.

1 INTRODUCTION

Software systems need to evolve continually to cope with changing requirements and environments. However, opposite to design patterns [19], code and design smells — poor practices in implementing design and implementation — may hinder their evolution by making it hard for software engineers to carry out changes. Code and design smells include low-/local problems such as code smells [21] which are usually symptoms of worse design smells such as anti-patterns [22]. Code smells are indicators of shortcomings in the existing codebase that suggest the possibility of refactoring. Disguised code smells are code that is not refactorable but appears to be refactoring the refactoring suggestions. One example of a code smell is the Spaghetti Code Antipattern, which is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes without structure that declare long methods without parameters. The names of the classes and methods may suggest procedural programming. Spaghetti Code does not exploit object-oriented mechanisms, such as polymorphism and inheritance, and prevents their use. We use the term "smells" to denote both code and design smells. This use does not refer to that, in fact, a smell can be the best way to actually design or implement a system. For example, patterns generated automatically by pattern generators, such as design patterns [23], have significant advantages over hand-crafted ones [24]. However, unlike such class "smells," software engineers must manually evaluate their refactoring suggestions for the source code smells and opportunities for refactoring. One example of a code smell is the Spaghetti Code Antipattern, which is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes without structure that declare long methods without parameters. The names of the classes and methods may suggest procedural programming. Spaghetti Code does not exploit object-oriented mechanisms, such as polymorphism and inheritance, and prevents their use. We use the term "smells" to denote both code and design smells. This use does not refer to that, in fact, a smell can be the best way to actually design or implement a system. For example, patterns generated automatically by pattern generators, such as design patterns [23], have significant advantages over hand-crafted ones [24]. However, unlike such class "smells," software engineers must manually evaluate their refactoring suggestions for the source code smells and opportunities for refactoring. One example of a code smell is the Spaghetti Code Antipattern, which is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes without structure that declare long methods without parameters. The names of the classes and methods may suggest procedural programming. Spaghetti Code does not exploit object-oriented mechanisms, such as polymorphism and inheritance, and prevents their use. We use the term "smells" to denote both code and design smells. This use does not refer to that, in fact, a smell can be the best way to actually design or implement a system. For example, patterns generated automatically by pattern generators, such as design patterns [23], have significant advantages over hand-crafted ones [24]. However, unlike such class "smells," software engineers must manually evaluate their refactoring suggestions for the source code smells and opportunities for refactoring. One example of a code smell is the Spaghetti Code Antipattern, which is characteristic of procedural thinking in object-oriented programming.
Some smells are intrinsically characterized on how code evolve over time.
Parallel Inheritance

Every time you make a subclass of one class, you also have to make a subclass of another

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>method1()</td>
<td>method1()</td>
</tr>
<tr>
<td>method2()</td>
<td></td>
</tr>
</tbody>
</table>
Parallel Inheritance

Every time you make a subclass of one class, you also have to make a subclass of another.
Every time you make a subclass of one class, you also have to make a subclass of another.
HIST

Historical Information for Smell deTection
**Change History Extractor**

- log download
- code analyzer

**Code Smells Detector**

- Association rule discovery to capture co-changes between entities
- Analysis of change frequency of some specific entities

Historical Information for Smell deTection
A class is changed in different ways for different reasons

Solution: Extract Class Refactoring

Detection

Classes containing at least two sets of methods such that:

(i) all methods in the set change together as detected by the association rules

(ii) each method in the set does not change with methods in other sets
A class implementing several responsibilities, having a large size, and dependencies with data classes

**Solution:**
Extract Class refactoring

**Detection**

Blobs are identified as classes frequently modified in commits involving at least another class.
<table>
<thead>
<tr>
<th>Software</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Tomcat</td>
<td></td>
</tr>
<tr>
<td>Apache Ant</td>
<td></td>
</tr>
<tr>
<td>jEdit</td>
<td></td>
</tr>
<tr>
<td>Android APIs: framework-opt-telephony</td>
<td></td>
</tr>
<tr>
<td>Android APIs: framework-base</td>
<td></td>
</tr>
<tr>
<td>Android APIs: framework-support</td>
<td></td>
</tr>
<tr>
<td>Android APIs: sdk</td>
<td></td>
</tr>
<tr>
<td>Android APIs: tool-base</td>
<td></td>
</tr>
</tbody>
</table>
Two Master students manually identified instances of the five smells of interest

We applied HIST on the selected snapshot, measuring its performances in terms of recall, precision, and F-Measure

We compared HIST with static code analysis techniques.

Blob: HIST vs DECOR [Mohamed et al., TSE 2010]

Feature Envy: HIST vs JDeodorant [Tsantalis et al., TSE 2009]

Divergent Change: HIST vs [Classes having a low cohesion as measured by the Connectivity metric]

Shotgun Surgery: HIST vs [Classes having at least a method invoking at least 4 different classes]

Parallel Inheritance: HIST vs [Pairs of classes (i) both belonging to hierarchies and (ii) have the same prefix in the class name]
Results
Feature Envy

![Bar Chart]

- Precision
- Recall

HIST
JDeodorant
Feature Envy

Precision
Recall

HIST \ JDeodorant
JDeodorant \ HIST

Precision
Recall

Feature Envy
Divergent Change

Precision

Recall

HIST

CA Technique

Precision

Recall
Divergent Change

Precision

Recall
Shotgun Surgery

- HIST
- CA Technique

Precision

Recall

- 80
- 100
Shotgun Surgery

Precision
Recall

80
100

CA Technique
HIST

Precision
Recall

HIST \ CA Technique
CA Technique \ HIST
Parallel Inheritance

![Graph showing Precision and Recall for HIST and CA Technique]

- **Precision**:
  - HIST: 61
  - CA Technique: 45

- **Recall**:
  - HIST: 61
  - CA Technique: 45
Parallel Inheritance

Precision

Recall

HIST \ CA Tecnique

CA Tecnique \ HIST

61

61

45
Results - Summary
Results - Summary

Average F-Measure

HIST
CA Technique

+48%
Can we define a hybrid approach to detect smells?
Does a class affected by a smell represent a problem even if it does not change so often?
Some smells are intrinsically characterized on how code evolve over time.

HIST

Historical Information for Smell deTection

Study Design

Can we define a hybrid approach to detect smells?

A class affected by a smell represents a problem if it does not change?

Results - Summary

Feature Envy