Code and Test Smells
Understanding and Detecting Them

Fabio Palomba
Senior Research Associate
University of Zurich
https://dibt.unimol.it/fpalomba/
Code and Test Smells
Understanding and Detecting Them
During software evolution changes cause a drift of the original design, reducing its quality.
Low design quality ...

... has been associated with lower productivity, greater rework, and more significant efforts for developers


“Bad Code Smells are symptoms of poor design or implementation choices”

[Martin Fowler]
package org.apache.xerces.xinclude;

import java.io.CharConversionException;

/**
 * This is a pipeline component which performs XInclude handling, according to the
 * W3C specification for XML Inclusions.
 */

This component analyzes each event in the pipeline, looking for &lt;include&gt;
* elements. An &lt;include&gt; element is one which has a namespace of
* &lt;code&gt;http://www.w3.org/2001/XInclude&lt;/code&gt; and a name of &lt;code&gt;include&lt;/code&gt;.
* When it finds an &lt;include&gt; element, it attempts to include the file specified
* in the &lt;code&gt;&lt;href&gt;&lt;/code&gt; attribute of the element. If inclusion succeeds, all
* children of the &lt;include&gt; element are ignored (with the exception of
* checking for any valid children as outlined in the &lt;code&gt;include&lt;/code&gt;.
* The inclusion fails, the &lt;include&gt; element is processed.
* See the &lt;a href=&quot;http://www.w3.org/TR/xinclude&quot;&gt;specification&lt;/a&gt; for
* more information on how XInclude is to be used.
* 
* This component requires the following features and properties from the
* component manager that uses it:
* 
* &lt;ul&gt;
* &lt;li&gt;&lt;code&gt;http://xml.org/sax/features/allow-dtd-notations&lt;/code&gt;&lt;/li&gt;
* &lt;li&gt;&lt;code&gt;http://xml.org/sax/features/allow-external-entity-declarations&lt;/code&gt;&lt;/li&gt;
* &lt;li&gt;&lt;code&gt;http://apache.org/xml/properties/internationalization/language-selector&lt;/code&gt;&lt;/li&gt;
* Optional property:
* &lt;ul&gt;
* &lt;li&gt;&lt;code&gt;http://apache.org/xml/properties/input-buffer-size&lt;/code&gt;&lt;/li&gt;
* &lt;/ul&gt;
* 
* Furthermore, the &lt;code&gt;NamespaceContext&lt;/code&gt; used in the pipeline is required
* to be an instance of &lt;code&gt;XIncludeNamespaceSupport&lt;/code&gt;.
* 
* Currently, this implementation has only partial support for the XInclude specification.
* Specifically, it is missing support for XPointer document fragments. Thus, only whole
*
Blob (or God Class)

A Blob (also named God Class) is a “class implementing several responsibilities, having a large number of attributes, operations and dependencies with data classes”.

[Martin Fowler]
Blob (or God Class)

A Blob (also named God Class) is a “class implementing several responsibilities, having a large number of attributes, operations and dependencies with data classes”.

[Martin Fowler]

Consequences

Increasing maintenance costs due to the difficulty of comprehending and maintaining the class.
40+ different smells
Negative Impact of Bad Smells

Bad Smells hinder code comprehensibility

[Abbes et al. CSMR 2011]
Negative Impact of Bad Smells

Bad Smells increase maintenance costs

[Banker et al. Communications of the ACM]
Bad Smells increase change- and fault-proneness

[Khomh et al. EMSE 2012]
Studies tried explaining their lifespan

Developers are aware of code smells, but not very concerned about their impact

[Peters and Zaidman - CSMR 2012]
In the vast majority of these cases code smell disappearance was not the result of targeted refactoring activities but rather a side effect of adaptive maintenance.

[Chatzigeorgiou et al. - QUATIC 2010]
Developers deliberately postpone refactorings for different reasons [Arcoverde et al. - IWRT 2011]

and longevity...
Developers perceive refactoring involves substantial cost and risks

[Kim et al. - TSE 2014]
WHEN AND WHY YOUR CODE STARTS TO SMELL BAD
5

Blob
Class Data Should Be Private
Complex Class
Functional Decomposition
Spaghetti Code

smells considered from the catalogues by Fowler and Brown
Class Data Should Be Private
A class exposing its attributes, violating the information hiding principle.

Complex Class
A class having high cyclomatic complexity

Functional Decomposition
A class where inheritance and polymorphism are poorly used, declaring many fields and implementing few methods

Spaghetti Code
A class without a structure that declares long methods without parameters
Study Design

3 different ecosystems analyzed

The Apache Software Foundation

Android

eclipse
200

total analyzed systems
When are code smells introduced?
WHEN blobs are introduced

Generally, blobs affect a class since its creation

There are several cases in which a blob is introduced during maintenance activities
Why are code smells introduced?
WHY are code smells introduced

Maintenance Activity

- Blob
- Class Data Should Be Private
- Complex Class
- Functional Decomposition
- Spaghetti Code
WHY are code smells introduced

Workload

- Blob
- Class Data Should Be Private
- Complex Class
- Functional Decomposition
- Spaghetti Code

[Bar chart showing workload distribution for each code smell:]

- Blob: High (70%), Medium (30%)
- Class Data Should Be Private: High (70%), Medium (30%)
- Complex Class: High (70%), Medium (30%)
- Functional Decomposition: High (70%), Medium (30%)
- Spaghetti Code: High (70%), Medium (30%)
WHY are code smells introduced

Newcomer

- Blob: True 25, False 75
- Class Data Should Be Private: True 25, False 75
- Complex Class: True 25, False 75
- Functional Decomposition: True 25, False 75
- Spaghetti Code: True 25, False 75
Do They Really Smell Bad?
A Study on Developers’ Perception of Bad Code Smells
“We don’t see things as they are, we see things as we are”

Anais Nin
Study Design

Class Data Should Be Private
Complex Class
Feature Envy
God Class
Inappropriate Intimacy
Lazy Class
Long Method
Long Parameter List
Middle Man
Refused Bequest
Spaghetti Code
Speculative Generality

Argo UML 0.34
Eclipse 3.6.1
jEdit 4.5.1

Original Developers:
10

Industrial Developers:
9

Master’s Students:
15
Developers are able to perceive smells related to long/complex code, while several instances are perceived depending on the intensity of the problem.

[Palomba et al. - ICSME 2014]
FEATURE ENVY
Refactoring operations are generally focused on code components for which quality metrics do not suggest there might be a need for refactoring operations.
The relation between code smells and refactoring is stronger. 42% of refactoring operations are performed on code entities affected by code smells.
However, often refactoring fails in removing code smells!

Only 7% of the performed operations actually remove the code smells from the affected class.
More Automation is Needed!
More Automation is Needed!

Detectors able to Take into Account the Findings on Code Smell Introduction!
More Automation is Needed!

Detectors able to Take into Account the Findings on Code Smell Introduction!

Detectors able to Produce Suggestions Closer to the Developers’ Perception of Design Problems!
Where to refactor
To detect code smells, several approaches and tools have been proposed, most of them relying on structural analysis.
Metric-based code smell detection

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCOM</td>
<td>15</td>
</tr>
<tr>
<td>WMC</td>
<td>20</td>
</tr>
<tr>
<td>CBO</td>
<td>25</td>
</tr>
</tbody>
</table>

$t = 17$
Metric-based code smell detection

AND combination

$t = 17$

- LCOM
- WMC
- CBO

Nasoul Moha, Yann-Gaël Guichêneuc, Laurence Duchien, and Anna-Françoise Le Meur

Abstract—Code and design smells are poor solutions to recurring implementation and design problems. They may hinder the evolution of a system by making it hard for software engineers to carry out changes. We propose three contributions to the research field related to code and design smells: 1) DECOR, a method that embeds and defines all the steps necessary for the specification and detection of code and design smells, 2) DETEX, a detection technique that instantiates the method, and 3) an empirical validation in terms of precision and recall of DETEX. The originality of DETEX stems from the ability for software engineers to specify smells at a high level of abstraction using a consistent vocabulary and domain-specific language for automatically generating detection algorithms. Lastly, DECOR allows for specification of code smells, the antipatterns list, functional decomposition of Spaghetti Code, and various Army Kites, and their 15 underlying code smells, and we automatically generate their detection algorithms. We apply and validate the detection algorithms in terms of precision and recall on RENEX V.7.0, and discuss the precision of these algorithms on 11 open source systems.

Index Terms—antipatterns, design smells, code smells, specification, metamodelling, detection, java.

1 INTRODUCTION

Software systems need to evolve continually to cope with ever-changing requirements and environments. However, opposite to design patterns [1], code and design smells—“poor” solutions to recurring implementation and design problems—may hinder their evolution by making it hard for software engineers to carry out changes.

Code and design smells include low-level or local problems such as code smells [2], which are usually symptoms of more global design smells such as antipatterns [3]. Code smells are indicators or symptoms of the possible presence of design smells. Fowler [2] presented 22 code smells, structures in the source code that suggest the possibility of refactoring. Duplicated code, long methods, large classes, and large parameter lists are just a few symptoms of design smells and opportunities for refactoring.

One example of a design 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 exclude that, in a particular context, a smell can be the best way to actually design or implement a system. For example, patterns generated automatically by parser generators are often Spaghetti Code, i.e., very large classes with very long methods. Yet, although such classes “smell,” software engineers must manually evaluate their possible negative impact according to the context. The detection of smells can substantially reduce the cost of subsequent activities in the development and maintenance phases [4]. However, detection in large systems is a very time-consuming and error-prone activity [5] because smells cut across classes and methods and their descriptions leave much room for interpretation.

Several approaches, as detailed in Section 2, have been proposed to specify and detect smells. However, they have three limitations. First, the authors do not explain the analysis leading to the specifications of smells and the underlying detection framework. Second, the translation of the specifications into detection algorithms is often a black box, which prevents replication. Finally, the authors do not present the results of their detection on a representative set of smells and systems to allow comparisons among approaches. So far, reported results concern proprietary systems and a reduced number of smells.

We present three contributions to overcome these limitations. First, we propose DECOR (Design & CODEntities) (DECOR), a method that describes all the steps necessary for the specification and detection of code and design smells. Our second contribution is DETEX, a detection technique that instantiates the method. Finally, we empirically validate DETEX in terms of precision and recall.
The Blob (also called God class) corresponds to a large controller class that depends on data stored in surrounding data classes. A large class declares many fields and methods with a low cohesion. A controller class monopolizes most of the processing done by a system, takes most of the decisions, and closely directs the processing of other classes. Controller classes can be identified using suspicious names such as Process, Control, Manage, System, and so on. A data class contains only data and performs no processing on these data. It is composed of highly cohesive fields and accessors.

[Moha et al. TSE 2010]
RULE_CARD : Blob {

RULE : Blob {ASSOC: associated FROM : mainClass ONE TO : DataClass MANY};

RULE : MainClass {UNION LargeClass, LowCohesion, ControllerClass};

RULE : LargeClass { ( METRIC : NMD + NAD, VERY_HIGH, 20) } ;

RULE : LowCohesion { ( METRIC : LCOM5, VERY_HIGH , 20) } ;

RULE : ControllerClass { UNION (SEMANTIC : METHODNAME, {Process, Control , Ctrl , Command , Cmd, Proc, UI, Manage, Drive} ) (SEMANTIC : CLASSNAME, { Process, Control, Ctrl, Command , Cmd, Proc , UI, Manage, Drive , System, Subsystem } ) } } ;

RULE : DataClass { (STRUCT: METHOD_ACCESSOR, 90%)} ;

[ Moha et al. TSE 2010 ]

Naouel Moha, Yann-Gaël Guimhaëne, Laurence Duchien, and Anna-Françoise Le Meur

Abstract—Code and design smells are poor solutions to recurring implementation and design problems. They may hinder the evolution of a system by making it hard for software engineers to carry out changes. We propose three contributions to the research field related to code and design smells: 1) DECOR, a method that embodies and defines all the steps necessary for the specification and detection of code and design smells; 2) DETEX, a detection technique that instantiates this method, and 3) an empirical validation of its precision and recall of DETEX. The originality of DETEX stems from the ability for software engineers to specify smells at a high level of abstraction using a consistent vocabulary and domain-specific language for automatically generating detection algorithms. Larry Constantine, we specify that well-known design smells: the antipatterns Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, and their 15 underlying code smells, are interpreted automatically by DETEX and that we automatically generate their detection algorithms. We apply and validate the detection algorithms in terms of precision and recall on TESSA CO and discuss the precision of these algorithms on 11 open-source systems.

1 INTRODUCTION

Software systems need to evolve continually to cope with ever-changing requirements and environments. However, oppose to design patterns [1], code and design smells—"poor" solutions to recurring implementation and design problems—may hinder their evolution by making it hard for software engineers to carry out changes.

Code and design smells include local or global defects such as code smells [2], which are usually symptoms of more global design smells such as antipatterns [3]. Code smells are indicators or symptoms of the possible presence of design smells. Fowler [2] presented 22 code smells, structures in the source code that suggest the possibility of refactorings. Duplicated code, long methods, large classes, and long parameter lists are just a few symptoms of design smells and opportunities for refactorings.

One example of a design 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 many 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 exclude that, in a particular context, a smell can be the best way to actually design or implement a system. For example, patterns generated automatically by pattern generators are often Spaghetti Code, i.e., very large classes with very long methods. Yet, although such classes "smell," software engineers must manually evaluate their possible negative impact according to the context.

The detection of smells can substantially reduce the cost of subsequent activities in the design and maintenance phases [4]. However, detection in large systems is a very time-consuming and error-prone activity [5] because smells cut across classes and methods, making their descriptions leave much room for interpretation.

Several approaches, as detailed in Section 2, have been proposed to specify and detect smells. However, they have three limitations. First, the authors do not explain the analysis leading to the specifications of smells and the underlying detection framework. Second, the translation of the specifications into detection algorithms is often an black box, which prevents replication. Finally, the authors do not present the results of their detection on a representative set of systems and only mention usage through approaches. So far, reported results concern proprietary systems and a reduced number of smells.

We present three contributions to overcome these limitations. First, we propose Design & Code Refactoring (DECOR), a method that describes all the steps necessary for the specification and detection of code and design smells.

Performances

Detect instances of four code smells (i.e., Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife) on 9 software systems

Average Recall: 100%
Average Precision: 60.5%

[Moha et al. TSE 2010]
But some smells are intrinsically characterized by how code evolves 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>
Every time you make a subclass of one class, you also have to make a subclass of another.
Parallel Inheritance

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

Historical Information for Smell Detection
## Extracting Change History Information

### Commits on Nov 20, 2018

- **A typo**
  - Gintas Grigelionis committed 11 days ago
  - Commit: `ac4ff1`

- **Fix javadoc**
  - Gintas Grigelionis committed 12 days ago
  - Commit: `3e0890f`

### Commits on Nov 19, 2018

- **Make DataType and Reference generic**
  - Gintas Grigelionis committed 12 days ago
  - Commit: `57895fd`

- **Remove unused imports**
  - Gintas Grigelionis committed 13 days ago
  - Commit: `bd82d18`

- **Refactor getZipEntryStream**
  - Gintas Grigelionis committed 13 days ago
  - Commit: `2c2c9b0`

### Commits on Nov 18, 2018

- **Avoid leaks in AntAnalyzer**
  - Gintas Grigelionis committed 14 days ago
  - Commit: `aff7eef`
Extracting Change History Information

Branch: master

Commits on Nov 20, 2018

- A typo
  - Gintas Grigelionis committed 11 days ago

- Fix javadoc
  - Gintas Grigelionis committed 12 days ago

Commits on Nov 19, 2018

- Make DataType and Reference generic
  - Gintas Grigelionis committed 12 days ago

- Remove unused imports
  - Gintas Grigelionis committed 13 days ago

- Refactor getZipEntryStream
  - Gintas Grigelionis committed 13 days ago

Commits on Nov 18, 2018

- Avoid leaks in AntAnalyzer
  - Gintas Grigelionis committed 14 days ago
Extracting Change History Information

Make DataType and Reference generic

Gintas Grigelionis committed 12 days ago

Showing 57 changed files with 379 additions and 261 deletions.

172 <untar src="$(output)/test11.tar.bz2" dest="$(output)/untar" compression="bzip2"/>
173 </target>
174

175 <target name="testTarFilesetWithReference">
176 + <fileset id="xml.fileset" dir="." includes="*.xml"/>
177 + <tar destfile="$(output)/testtar.tar">
178 + <tarfileset prefix="pre" refid="xml.fileset"/>
179 + </tar>
180 + </target>

175 + <target name="feather">
176 + <tar destfile="$(output)/asf-logo.gif.tar"
Extracting Change History Information

git log

log download

Historical Information
Extracting Change History Information

git log

log download

files modified
Extracting Change History Information

- git log
- files modified

log download

commit i

commit i+1

code analyzer
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
Association Rule Mining

Changes occurring in snapshots

Files

C1: A → C
C2: B → C
C3: A → C
C4: B → D
C5: B → D
C6: A → C
Association Rule Mining

Files

Changes occurring in snapshots

C1: A → C

C2: B → C

C3: B → C

C4: B → D

C5: B → D

C6: C → E
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.
Evaluation
detection accuracy

20 open source systems

Comparing HIST with static code analysis technique on a manually built oracle
## Evaluation

detection accuracy

### 20 open source systems

<table>
<thead>
<tr>
<th></th>
<th>HIST F-Measure</th>
<th>CA technique F-Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shotgun Surgery</td>
<td>92%</td>
<td>0%</td>
</tr>
<tr>
<td>Parallel Inheritance</td>
<td>71%</td>
<td>9%</td>
</tr>
<tr>
<td>Divergent Change</td>
<td>82%</td>
<td>11%</td>
</tr>
<tr>
<td>Blob</td>
<td>64%</td>
<td>48%</td>
</tr>
<tr>
<td>Feature Envy</td>
<td>77%</td>
<td>68%</td>
</tr>
</tbody>
</table>

Comparing HIST with static code analysis technique on a manually built oracle.
Evaluation
detection accuracy

20 open source systems

Comparing HIST with static code analysis technique on a manually built oracle

HIST and the CA techniques are highly complementary
Structural and Historical Analysis are only a part of the whole story.
Toward a New Dimension of Code Smell Detection
/* Insert a new user in the system.  
* @param pUser: the user to insert.*/
public void insert(User pUser) {
    connect = DBConnection.getConnection();

    String sql = "INSERT INTO USER"
        + "(login,first_name,last_name,password",
        + "email,cell,id_parent) " + "VALUES ("
        + pUser.getLogin() + "",
        + pUser.getFirstName() + "",
        + pUser.getLastName() + "",
        + pUser.getPassword() + "",
        + pUser.getEMail() + "",
        + pUser.getCell() + "",
        + pUser.getIdParent() + ")";
    executeOperation(connect, sql);
}

/* Delete an user from the system.  
* @param pUser: the user to delete.*/
public void delete(User pUser) {
    connect = DBConnection.getConnection();

    String sql = "DELETE FROM USER"
        + "WHERE id_user = "
        + pUser.getId();
    executeOperation(connect, sql);
}
Indeed, source code vocabulary can be an useful additional source of information
Extracting and Normalizing Text

```java
private Connection connect = DBConnection.getConnection();
```
Extracting and Normalizing Text

```java
private Connection connect = DBConnection.getConnection();
```

Separating Composed Identifiers

```java
private Connection connect = DB Connection.getConnection();
```
Extracting and Normalizing Text

private Connection connect = DBConnection.getConnection();

Separating Composed Identifiers

private Connection connect = DB Connection.getConnection();

Lower Case Reduction

private connection connect = db connection.getConnection();
Extracting and Normalizing Text

private Connection connect = DBConnection.getConnection();

Separating Composed Identifiers

private Connection connect = DB Connection.get Connection();

Lower Case Reduction

private connection connect = db connection.get connection();

Removing Special Characters, programming keywords, and common English terms

connection connect = db connection get connection
Extracting and Normalizing Text

```
private Connection connect = DBConnection.getConnection();
```

- **Separating Composed Identifiers**

```
private Connection connect = DBConnection.getConnection();
```

- **Lower Case Reduction**

```
private connection connect = db connection.getConnection();
```

- **Removing Special Characters, programming keywords, and common English terms**

```
connection connect = db connection.getConnection();
```

- **Stemming**

```
connect connect = db connect get connect
```
TACO

Textual Analysis for Code smell detection
We believe that code affected by a smell contains unrelated textual content
Text Preprocessing

- Code Component
- Textual component extractor
- IR normalization process
  - Stemming
  - Term separation
  - ... Stop word removal

Smell Detector

- 0.86 avg. smelliness level
- Dissimilarity computation
- Block extractor
To detect smells, we need a threshold over the probability distribution. As cut point, we select the median of the non-null values of the smelliness.
We instantiate TACO to detect 5 different code smells characterized by promiscuous responsibilities.

- Long Method
- Blob
- Promiscuous Package

![Tangled wires](image)
TACO can detect 5 different code smells characterized by promiscuous responsibilities.

- **Long Method**
- **Blob**
- **Promiscuous Package**
- **Feature Envy**
- **Misplaced Class**
public void insert(User pUser) {

    connect = DBConnection.getConnection();

    String sql = "INSERT INTO USER"
        + "(login, first_name, last_name, password"
        + ", email, cell, id_parent) " + "VALUES ("
        + pUser.getLogin() + ","
        + pUser.getFirstName() + ","
        + pUser.getLastName() + ","
        + pUser.getPassword() + ","
        + pUser.getEmail() + ","
        + pUser.getCell() + ","
        + pUser.getIdParent() + ");"

    String sql = "DELETE FROM USER "
        + "WHERE id_user = "
        + pUser.getId();
public void insert(User pUser) {
    
    connect = DBConnection.getConnection();
    
    String sql = "INSERT INTO USER"
        + "(login, first_name, last_name, password"
        + ", email, cell, id_parent) " + "VALUES ("
        + pUser.getLogin() + ",",
        + pUser.getFirstName() + ",",
        + pUser.getLastName() + ",",
        + pUser.getPassword() + ",",
        + pUser.getEMail() + ",",
        + pUser.getCell() + ",",
        + pUser.getIdParent() + ")";
    
    String sql = "DELETE FROM USER "
        + "WHERE id_user = "
        + pUser.getId();
}
Detecting Long Method instances

public void insert(User pUser) {

    connect = DBConnection.getConnection();

    String sql = "INSERT INTO USER"
        + "(login,first_name,last_name,password"
        + ",email,cell,id_parent) " + "VALUES ("
        + pUser.getLogin() + ","
        + pUser.getFirstName() + ","
        + pUser.getLastName() + ","
        + pUser.getPassword() + ","
        + pUser.getEmail() + ","
        + pUser.getCell() + ","
        + pUser.getIdParent() + ")";

    String sql = "DELETE FROM USER"
        + "WHERE id_user = "
        + pUser.getId();
Detecting Long Method instances

```java
public void insert(User pUser){
    connect = DBConnection.getConnection();
    String sql = "INSERT INTO USER"
        + "(login,first_name,last_name,password"
        + ",email,cell,id_parent) " + "VALUES ("
        + pUser.getLogin() + ","
        + pUser.getFirstName() + ","
        + pUser.getLastName() + ","
        + pUser.getPassword() + ","
        + pUser.getEMail() + ","
        + pUser.getCell() + ","
        + pUser.getIdParent() + ");"
    }
    String sql = "DELETE FROM USER "
        + "WHERE id_user = "
        + pUser.getId();
```

X. Whang, L. Pollock, K. Shanker
JSEP 2013
Detecting Feature Envy instances
Detecting Feature Envy instances

Extracting the class $C_{\text{closest}}$ having the highest textual similarity with $M_i$
Detecting Feature Envy instances

Feature Envy Probability Computation

\[ C_{\text{closest}} \]

- method1()
- method2()
- ...
- methodN()

\[ C_0 \]

- method1()
- method2()
- ...
- methodN()
TACO - Evaluating its performance

<table>
<thead>
<tr>
<th></th>
<th>TACO</th>
<th>Alternative Structural Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM</td>
<td>76</td>
<td>43</td>
</tr>
<tr>
<td>Blob</td>
<td>79</td>
<td>60</td>
</tr>
<tr>
<td>PP</td>
<td>73</td>
<td>55</td>
</tr>
<tr>
<td>FE</td>
<td>70</td>
<td>62</td>
</tr>
<tr>
<td>MC</td>
<td>81</td>
<td>43</td>
</tr>
</tbody>
</table>
TACO - Evaluating its performance

+22% on average in terms of F-Measure
TACO - Evaluating its performance

Method: findTypesAndPackages()

Class: CompletionEngine - Eclipse Core

Goal: Discover the classes and the packages of a given project
TACO - Evaluating its performance

Method: findTypesAndPackages()
Class: CompletionEngine - Eclipse Core
Goal: Discover the classes and the packages of a given project

65 lines of code
TACO - Evaluating its performance

Method: findTypesAndPackages()
Class: CompletionEngine - Eclipse Core
Goal: Discover the classes and the packages of a given project

65 lines of code

A Structural Approach cannot detect the smell!
TACO - Evaluating its performance

Method: findTypesAndPackages()
Class: CompletionEngine - Eclipse Core
Goal: Discover the classes and the packages of a given project

65 lines of code

A Structural Approach cannot detect the smell!

TACO, instead, is able to detect a Long Method instance
TACO - Evaluating complementarity with structural approaches
Textual and Structural Information are Highly Complementary
Toward a combination of code smell detection techniques?
Code and **Test Smells**
Understanding and Detecting Them
Smells in Test Code

Refactoring Test Code

Arie van Deursen  
Leon Moonen

CW1  
The Netherlands
http://www.cwi.nl/~arie.leon/  
(arie.leon@cw1.nl)

Alex van den Bergh  
Gerard Kok

Software Improvement Group  
The Netherlands
http://www.software-improvers.com/  
(alex.bergh@software-improvers.com)

ABSTRACT

Two key aspects of extreme programming (XP) are unit testing and merciless refactoring. Given the fact that the ideal test code/production code ratio approaches 1:1, it is not surprising that unit tests are being refactored. We found that refactoring test code is different from refactoring production code in two ways: (i) there is a distinct set of bad smells involved, and (ii) improving test code involves additional test-specific refactorings. To share our experiences with other XP practitioners, we describe a set of bad smells that indicate trouble in test code, and a collection of test refactorings to remove these smells.

Keywords

Refactoring, unit testing, extreme programming.

1 INTRODUCTION

"If there is a technique at the heart of extreme programming (XP), it is unit testing" [1]. As part of their programming activity, XP developers write and maintain (white box) unit tests continually. These tests are automated, written in the same programming language as the production code, considered an explicit part of the code, and put under revision control.

The XP process encourages writing a test class for every class in the system. Methods in these test classes are used to verify complicated functionality and unusual circumstances. Moreover, they are used to document code by explicitly indicating what the expected results of a method should be for typical cases. Last but not least, tests are added upon receiving a bug report to check for the bug and to check the bug fix [2]. A typical test for a particular method includes: (1) code to set up the fixture (the data used for testing), (2) the call of the method, (3) a comparison of the actual results with the expected values, and (4) code to tear down the fixture. Writing tests is usually supported by frameworks such as NUnit [3].

The test code/production code ratio may vary from project to project, but is ideally considered to approach a ratio of 1:1. In our project we currently have a 2:3 ratio, although others have reported a lower ratio [4]. One of the cornerstones of XP is that having many tests available helps the developers to overcome their fear for change: the tests will provide immediate feedback if the system gets broken at a critical place. The downside of having many tests, however, is that changes in functionality will typically involve changes in the test code as well. The more test code we get, the more important it becomes that this test code is as easily modifiable as the production code.

The key XP practice to keep code flexible is "refactor mercilessly": transforming the code in order to bring it to the simplest possible state. To support this, a catalog of "code smells" and a wide range of refactorings is available, varying from simple modifications up to ways to introduce design patterns systematically in existing code [5].

When trying to apply refactorings to the test code of our project we discovered that refactoring test code is different from refactoring production code. Test code has a distinct set of smells, dealing with the ways in which test cases are organized, how they are implemented, and how they interact with each other. Moreover, improving test code involves a mixture of refactorings from [5] specialized to test code improvements, as well as a set of additional refactorings, involving the modification of test classes, ways of grouping test cases, and so on.

The goal of this paper is to share our experience in improving our test code with other XP practitioners. To that end, we describe a set of test smells indicating trouble in test code, and a collection of test refactorings explaining how to overcome some of these problems through a simple program modification.

This paper assumes some familiarity with the NUnit framework [3] and refactorings as described by Fowler [5]. We will refer to refactorings described in this book using NUnit.

"Test Smells represent a set of a poor design solutions to write tests "

[Van Deursen et al. - XP 2001]
public void test12 () throws Throwable {
    JSTerm jSTerm0 = new JSTerm();
    jSTerm0.makeVariable();
    jSTerm0.add((Object) "");
    jSTerm0.matches(jSTerm0);
    assertEquals(false, jSTerm0.isGround());
    assertEquals(true, jSTerm0.isVariable());
}
public void test12 () throws Throwable {
    JSTerm jSTerm0 = new JSTerm();
    jSTerm0.makeVariable () ;
    jSTerm0.add((Object) "") ;
    jSTerm0.matches(jSTerm0);
    assertEquals (false, jSTerm0.isGround ());
    assertEquals(true, jSTerm0.isVariable());
}

The test method checks the production method isGround()
Smells in Test Code

```java
public void test12 () throws Throwable {
    JSTerm jSTerm0 = new JSTerm();
    jSTerm0.makeVariable();
    jSTerm0.add((Object) "");
    jSTerm0.matches(jSTerm0);
    assertEquals(false, jSTerm0.isGround());
    assertEquals(true, jSTerm0.isVariable());
}
```

But also the production method isVariable()
public void test12 () throws Throwable {
    JSTerm jSTerm0 = new JSTerm();
    jSTerm0.makeVariable ();
    jSTerm0.add((Object) """);
    jSTerm0.matches(jSTerm0);
    assertEquals (false, jSTerm0.isGround ());
    assertEquals (true, jSTerm0.isVariable());
}

This is an Eager Test, namely a test which checks more than one method of the class to be tested, making difficult the comprehension of the actual test target.
A test case is affected by a Resource Optimism when it makes assumptions about the state or the existence of external resources, providing a non-deterministic result that depend on the state of the resources.

An Assertion Roulette comes from having a number of assertions in a test method that have no explanation. If an assertion fails, the identification of the assert that failed can be difficult.
Smells in Test Code

Tests affected by test smells are more change- and fault-prone than tests not participating in design flaws and affect the reliability of production code.

In 54% of the cases, test code flakiness can be induced by the presence of some design flaw in test code.
Detecting test smells using heuristics

```java
public void test12 () throws Throwable {
    JSTerm jSTerm0 = new JSTerm();
    jSTerm0.makeVariable();
    jSTerm0.add((Object) "");
    jSTerm0.matches(jSTerm0);
    assertEquals(false, jSTerm0.isGround());
    assertEquals(true, jSTerm0.isVariable());
}
```

Test smell detected if the number of method calls $> 3$
Text Preprocessing

- Code Component
- Textual component extractor
- IR normalization process
  - Stemming
  - Term separation
  - Stop word removal

Smell Detector

- avg. smelliness level: 0.86
- Dissimilarity computation
- Block extractor
TASTE: Detecting test smells using the textual component of test code

test method m

...  
A.x() 
...  
A.y()
TASTE: Detecting test smells using the textual component of test code

production class A

```java
public void x() {
    // some content
}
```

```java
public void y() {
    // some other content
}
```

test method m

```java
...  
A.x()  
...  
A.y()  
...  
```
TASTE: Detecting test smells using the textual component of test code

production class A

```java
public void x() {
}
public void y() {
}
```

test method m'

```java
... 
// some content 
... 
// some other content 
```
TASTE: Detecting test smells using the textual component of test code
TASTE: Detecting test smells using the textual component of test code

test method m'
...
// some content
...
// some other content

IR normalization

\[
\text{mean } i \neq j \quad \text{sim}(m'_i, m'_j)
\]
TASTE: Detecting test smells using the textual component of test code

\[
p_{ET}(t) = 1 - \frac{\sum_{i \neq j} \text{sim}(m'_i, m'_j)}{\text{mean}}
\]
TASTE: Detecting test smells using the textual component of test code

test method $m'$

\[
\begin{align*}
... \\
// \text{ some content} \\
... \\
// \text{ some other content}
\end{align*}
\]

IR normalization

\[p_{ET}(t) > 0.5\]
Code and Test Smells
Understanding and Detecting Them

Fabio Palomba
Senior Research Associate
University of Zurich
https://dibt.unimol.it/fpalomba/