Code and Test Smells
Understanding and Detecting Them

Fabio Palomba
Assistant Professor
University of Salerno (Italy)
https://fpalomba.github.io
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]
### XIncludeHandler.java

* Copyright 2003-2005 The Apache Software Foundation.
* package org.apache.xerces.xinclude;

```java
import java.io.CharConversionException;
```

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

```
```

```java
This component analyzes each event in the pipeline, looking for &lt;include&gt;
```

```java
&lt;include&gt; element is one which has a namespace of
```

```java
http://www.w3.org/2001/XInclude" and a sequence of &lt;include&gt; elements.
```

```java
When it finds an &lt;include&gt; element, it attempts to include the file specified
```

```java
in the <code>&lt;include href=""&gt;</code> attribute of the element. If inclusion succeeds, all
```

```java
children of the &lt;include&gt; element are ignored (with the exception of
```

```java```
checking for any valid children as outlined in the inclusion
```

```java```
falls, the <code>&lt;include href=""&gt;&lt;/include&gt;</code> child of the &lt;include&gt; element is processed.
```

```java```
See the <a href="http://www.w3.org/TR/xinclude" title="XInclude specification"/> for
```

```java```
more information on how XInclude is to be used.
```

```java```
This component requires the following features and properties from the
```

```java```
component manager that uses it:
```

```java```
- &lt;http://xml.org/sax/features/valid-dtd-construction-ok&gt;&lt;/feature&gt;
```

```java```
- &lt;http://xml.org/xmlproperties/external-general-entities-enabled&gt;&lt;/property&gt;
```

```java```
- Optional property:
```

```java```
- &lt;http://apache.org/xml/properties/input-buffer-size&gt;&lt;/property&gt;
```

```java```
Furthermore, the <code>NamespaceContext</code> used in the pipeline is required
```

```java```
to be an instance of <code>XIncludeNamespaceSupport</code>.
```

```java```
Currently, this implementation has only partial support for the XInclude specification.
```

```java```
Specifically, it is missing support for XPointer document fragments. Thus, only whole
```
```
```
```
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
40+ different smells
... and even more

Energy-related code smells
Security-related code smells
Performance-related code smells
40+ different smells
... and even more

Quality-related code smells
Bad Smells hinder code comprehensibility

[Abbes et al. CSMR 2011]
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]
and longevity...

Developers deliberately postpone refactoring for different reasons [Arcoverde et al. - IWRT 2011]
Developers perceive refactoring involves substantial cost and risks
[Kim et al. - TSE 2014]
WHEN AND WHY YOUR CODE STARTS TO SMELL BAD
Study Design

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

5 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
Study Design

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

Bug Fixing | Enhancement | New Feature | Refactoring

0 | 25 | 50 | 75 | 100
WHY are code smells introduced

Workload

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

- [High Workload: 80%]
- [Medium Workload: 20%]
- [Low Workload: 0%]
WHY are code smells introduced

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

Original Developers:
- 10

Industrial Developers:
- 9

Master’s Students:
- 15

Software Tools:
- Argo UML 0.34
- Eclipse 3.6.1
- jEdit 4.5.1
Study Design

Developer

Smelly Class
In your opinion, does this code component exhibit any design and/or implementation problem?
In your opinion, does this code component exhibit any design and/or implementation problem?

- If YES, please explain what are, in your opinion, the problems affecting the code component.
In your opinion, does this code component exhibit any design and/or implementation problem?

- If YES, please explain what are, in your opinion, the problems affecting the code component.

- If YES, please rate the severity of the design and/or implementation problem by assigning a score on the following five-points Likert scale: 1 (very low), 2 (low), 3 (medium), 4 (high), 5 (very high).
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 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

Google

`code smell detection`

Scholar

Circa 46.500 risultati (0,02 sec)
To detect code smells, several approaches and tools have been proposed, most of them relying on structural analysis.
How would you detect code smells?
Metric-based code smell detection

\[ t = 17 \]

- LCOM
- WMC
- CBO
Metric-based code smell detection

AND combination

t = 17

Nasour Moha, Yann-Gaël Guillonneau, Laurence Duchien, and Anna-Françoise Lo 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 apply bug fixes. 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 this 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. Using DETEX, we specify that well-known design smells: the antipatterns Idea, Functional Decomposition, Spaghetti Code, and Sister Army Knife, 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 REACTC v1.7.0, and discuss the precision of these algorithms on 11 open-source systems.

Index Terms—antipatterns, design smells, code smells, specification, reverseengineering, detection, analysis.

1 INTRODUCTION

Software systems need to evolve continually to cope with ever-changing requirements and environments. However, to achieve 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 long 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 [4], 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 particular contexts, 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 of large systems is a very time and resource-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 black box, which prevents replication. Finally, the authors do not present the results of their detection as a representative set of small and systems to allow comparison 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 & DETEX (DECOR), a method that describes all the steps necessary for the specification and detection of code and design smells.
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]
DECOR

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]

Naoou Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anna-Françoise Lo 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 smells; 2) DETEX, a detection technique that instantiates this 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.

In summary, we specify four well-known design smells: the antipatterns Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, 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 12 open-source systems.

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

1 INTRODUCTION

Software systems need to evolve continually to cope with non-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 long 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, parameters 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 and resource-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 black box, which prevents replication. Finally, the authors do not present the results of their detection on a representative set of smalls 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 & CORlection (DECOR), a method that describes all the steps necessary for the specification and detection of code and design

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>
Parallel Inheritance

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
Historical Information for Smell Detection
Extracting Change History Information

apache / ant

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

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

**git log**

*log download*
Extracting Change History Information

log download

git log

files modified
Extracting Change History Information

`git log`  
files modified

`log download`

commit i

commit i+1

code analyzer

getUser

getData

getLogin

method

method

method

method

class User

has been added

modified

moved
**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

Files

Changes occurring in snapshots

C1

C2

C3

C4

C5

C6

A

B

C

D

E
Association Rule Mining

Files

Changes occurring in snapshots
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

<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

20 open source systems
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
The textual content of source code can provide useful hints for 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

```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() + ")";

    executeOperation(connect, sql);
}
```

```java
public void delete(User pUser) {
    connect = DBConnection.getConnection();

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

    executeOperation(connect, sql);
}
```
private Connection connect = DBConnection.getConnection();
private Connection connect = DBConnection.getConnection();

Separating Composed Identifiers

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

Separating Composed Identifiers

private Connection connect = DB Connection.getConnection();

Lower Case Reduction

private connection connect = db connection.getConnection();
private Connection connect = DBConnection.getConnection();

Separating Composed Identifiers

private Connection connect = DB Connection.getConnection();

Lower Case Reduction

private connection connect = db connection.getConnection();

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 = DB Connection.getConnection();

Lower Case Reduction

private connection connect = db connection.getConnection();

Removing Special Characters, programming keywords, and common English terms

connection connect = db connection.get connection

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
- Stemming
- Term separation
- Stop word removal
- IR normalization process

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.
TACO can identify 5 different code smells characterized by promiscuous responsibilities.
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();

X. Whang, L. Pollock, K. Shanker
"Automatic Segmentation of Method Code Into Meaningful Blocks: Design and Evaluation"
JSEP 2013
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();

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

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

[Van Deursen et al. - XP 2001]

Test smells related to the way developers write test fixtures and test cases
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()
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()
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.
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

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

test method m

...
TASTE: Detecting test smells using the textual component of test code

production class A

public void x() {
    // some content
}

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

test method m

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

production class A

public void x() {
    // some content
}

public void y() {
    // some other content
}
TASTE: Detecting test smells using the textual component of test code

test method $m'$

\[ \ldots \]

// some content

\[ \ldots \]

// some other content

IR normalization
TASTE: Detecting test smells using the textual component of test code

Test method $m'$

```plaintext
...  
// some content  
...  
// some other content
```

IR normalization

$$i \neq j \quad \text{sim}(m'_i, m'_j)$$

mean
TASTE: Detecting test smells using the textual component of test code

test method $m'$

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

IR normalization

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

```
x()  
// some content  
...  

y()  
// some other content  
```

IR normalization

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

Fabio Palomba
Assistant Professor
University of Salerno (Italy)
https://fpalomba.github.io