Preliminary Catalogue of Anti-pattern and Code Smell False Positives

Francesca Arcelli Fontana[†], Jens Dietrich[*], Bartosz Walter[‡], Aiko Yamashita[§], and Marco Zanoni[†]

School of Engineering and Advanced Technology, Massey University, Palmerston North, New Zealand j.b.dietrich@massey.ac.nz

[†]Department of Informatics, Systems and Communication, University of Milano-Bicocca, Milano, Italy {marco.zanoni, arcelli}@disco.unimib.it

[‡]Faculty of Computing, Poznan University of Technology, Poznan, Poland bartosz.walter@cs.put.poznan.p

[§]Department of Information Technology, Oslo and Akershus University of Applied Sciences, Oslo, Norway
aiko.yamashita@hioa.no

Technical Report
RA-5/2015

Poznań University of Technology

November 2015

ABSTRACT

Anti-patterns and code smells are archetypes used for describing software design shortcomings that can negatively affect software quality, in particular maintainability. Tools, metrics and methodologies have been developed to identify these structural archetypes, based on the assumption that they can point at problematic code. However, recent empirical studies have shown that some of these archetypes are ubiquitous in real world programs, and many of them are found to not be as detrimental to quality as previously conjectured. We are therefore interested to revisit common smells and anti-patterns, and build a catalogue of cases that constitute candidates for “false positives”. A *Meta-synthesis* was conducted on: a) the last 10 years of empirical studies conducted on code smells and anti-patterns, b) practical examples from grey literature (e.g., blogs), and c) case studies from industry and open source projects. We furthermore propose a preliminary classification to improve the current conceptual framework for understanding better the effects of code smells and anti-patterns in practice. We hope that the development and further refinement of such classification can support researchers and tool vendors in their endeavor for developing more pragmatic, context-relevant detection and analysis tools for anti-patterns and code smells.

TABLE OF CONTENT

NAME OF SMELL/ANTI-PATTERN: REFUSED BEQUEST

A. Definition / Description

Subclasses don’t want or need everything they inherit [1]. A metrics-based definition proposed by Marinescu [2] is as follows:

Refused Bequest? = AIUR lower 1

Where:
Average Inheritance Usage Ratio (AIUR)

B. Argumentation: Empirical study

In the study by Sjøberg et al. [3], involving medium-sized industrial Java systems, **Refused Bequest was** the only smell that remained **significant when considering file size and change, and registered a decrease in effort.** In the study by Li & Shatnawi [4], which analyzed large Open Source Java systems, **Refused Bequest was not associated significantly with software faults**. The study by Palomba [5] indicates that developers perceive Refused Bequest as negative depending on the system’s domain. In particular, for ArgoUML and JEdit all respondents almost never perceived classes affected by Refused Bequest as problematic. The situation was quite different in Eclipse, where quite extreme cases were encountered (such as 52 out of 53 methods were overridden).

C. Code examples:

Marker interfaces with a single method[1].

D. Contextual factors:

Factor	Implication
Interpretation of maintainability	The effect depends on the actual operationalization, in terms of: • Perceived maintainability effect • Defects
Extreme cases	The perception of the negative impact of refused bequest is contingent on whether there are extreme values associated to this smell or not.
Type of Refusal	The use of certain language features to actively refuse inherited functionality may affect the programmer’s perception whether a given construct is perceived as a smell or not. For example, Javas optional methods are implemented as the ability to override a method by just throwing a *java.lang. UnsupportedOperationException*. This often violated core OO design principles, such as Liskov’s Substitution Principle [6], and is therefore

[1] Bloch, Joshua (2008). "Item 37: Use marker interfaces to define types". *Effective Java (Second edition)*. Addison-Wesley. p. 179. ISBN 978-0-321-35668-0.

	more likely to be perceived as problematic by a programmer than alternative solutions (such as empty method bodies).

E. References:

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, “Refactoring: Improving the Design of Existing Code”. Addison Wesley, 1999

[2] M. Lanza and R. Marinescu, ”Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate and Improve the Design of Object-Oriented Systems”. Springer, 2006.

[3] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba, "Quantifying the Effect of Code Smells on Maintenance Effort," IEEE Trans. on Software Engineering, 2013, vol.39, no.8, pp. 1144-1156.

[4] W. Li and R. Shatnawi, “An Empirical Study of the Bad Smells and Class Error Probability in the Post-Release Object-Oriented System Evolution”, J. Systems Software, 2007, vol. 80, no. 7, pp. 1120-1128.

[5] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, "Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells," Proc. IEEE Int. Conf. on Software Maintenance and Evolution (ICSME), 2014, pp. 101-110.

[6] B. H. Liskov, J. M. Wing, “A behavioral notion of subtyping”. ACM Trans. Program. Lang. Syst. 16 (6): 1811–1841. 1994.

NAME OF SMELL/ANTI-PATTERN: MESSAGE CHAINS

A. Definition / Description

Fowler et al. [1] describes this smell as follows: “You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another object, and so on. You may see these as a long line of *getThis()* methods, or as a sequence of temps.”
Message Chains conflicts with Builder patterns used to avoid exponential explosion in constructor parameters when objects are created.

B. Argumentation:

A characteristic of the builder pattern is that the methods always return the same object. This is also called Fluent Interface [2], which allows to chain calls that refer to the same object without retyping the variable name. In this way, we do not “navigate through the object graph”, because only one object receives the method calls. This seemed to have been the main concern behind method chains and the related Law of Demeter.

C. Example:

Modified from [1]:
```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
  .maximumSize(10000)
    .expireAfterWrite(10, TimeUnit.MINUTES)
    .removalListener(MY_LISTENER)
    .weakKeys()
    .softValues()
    .recordStats()
    .build(new CacheLoader<Key, Graph>() … });
```

E. Identified False Positive Examples

Test Class Method

Similarly to Dispersed Coupling, sometimes a method of a test class can be detected as a Message Chains, because it calls many methods in different classes.

Possible detection strategies:

- At least one method of the class that contains the Dispersed Coupling calls a method of classes that are part of *org.junit* or *junit.framework* (or another test-related library) packages;
- A class inherits (directly or indirectly) from *org.junit.TestCase* or *junit.framework.TestCase* (or respective classes in other test-related libraries)
- At least one method of the class is annotated as *@Test*

Builder DP

When using a Builder DP implemented as reported in [3], the chained calls to the Builder can be detected as Message Chains.

Possible detection strategies:

- Use design pattern detection to identify builders. Builder methods typically return *this/self* to facilitate chaining.

F. References:

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the Design of Existing Code". Addison Wesley, 1999

[2] http://martinfowler.com/bliki/FluentInterface.html

[3] Joshua Bloch, "Effective Java (2nd Edition)". Prentice Hall, 2008.

[4] http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/cache/CacheBuilder.html

NAME OF SMELL/ANTI-PATTERN: GOD METHOD

A. Definition / Description:

A class has the God Method bad smell if at least one of its methods is very large compared to the other methods in the same class. God Method centralizes the class functionality in a single method [1]. A metrics based definition proposed by Marinescu [2] is as follows:

(LOC top 20 % except LOC < 70) and (NOP > 4 or NOLV > 4) and MNOB > 4

Where:
Lines Of Code (LOC)
Number Of Parameters (NOP)
Number Of Local Variables (NOLV)
Max Number Of Branches (MNOB)

B. Synonyms:
Spaghetti code
Long Method
Brain Method

C. Argumentation: Empirical evidence
Abbes et al., [3] conducted an experiment in which 24 students and professionals were asked questions about the code in the OSSs YAMM, JVerFileSystem, AURA, GanttProject, JFreeChart and Xerces. They found that God Classes and God Methods alone had no effect over effort and correctness, but compared with the code without both of these smells, the code with combinations of God Class and God Method had a statistically significant increase in effort and a statistically significant decrease in the percentage of correctness.

D. Contextual factors:

Factor	**Implication**
Interaction effects with other smells or structural attributes.	From the cases observed, it is possible to observe that its effect is contingent on the **interaction effect** by means of **collocation** or potentially **coupling** of God Methods with other smells, like in this case God Class.
Generated Code	Generated code often contains very large and complex methods, e.g., parsers automatically generated by tools such as ANTLR.

E. Identified False Positive Examples
Please refer to the corresponding section on God Class.

F. References:

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, “Refactoring: Improving the Design of Existing Code”. Addison Wesley, 1999.

[2] M. Lanza and R. Marinescu, “Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate and Improve the Design of Object-Oriented Systems”. Springer, 2006.

[3] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, “An empirical study of the impact of two antipatterns, Blob and Spaghetti code, on program comprehension”. In: Proc. CSMR. 2011, pp. 181–190.

NAME OF SMELL/ANTI-PATTERN: GOD CLASS

A. Definition / Description
A class has the God Class smell if the class takes too many responsibilities relative to the classes to which it is coupled. The God Class centralizes the system functionality in one class, which is in contradiction with the decomposition design principles [1]. A metrics based definition proposed by Marinescu [2] is as follows:

AOFD top 20 % and AOFD > 4 and WMPC1 > 20 and TCC < 33

Where:
Access Of Foreign Data (AOFD)
Weighted Methods Per Class 1 (WMPC1)
Tight Class Cohesion (TCC)

B. Synonyms:
Large Class
Blob
Controller
Brain Class

C. Argumentation: Empirical study
Abbes et al. [3] conducted an experiment in which 24 students and professionals were asked questions about the code in six OSSs. They concluded that classes and methods identified as God Classes and God Methods in isolation had no effect on effort or correctness of the tasks, but when appearing together, they led to a statistically significant increase in response effort and a statistically significant decrease in the percentage of correct answers, i.e., significantly higher problems with understanding the code.

Olbrich et al. [4] reported on an experiment involving the analysis of three open-source systems: OSSs Lucene, Xerces and Log4j. They conducted a nonparametric hypothesis testing of the code and bug-tracker information. They observed that God and Brain classes were changed less frequently and had fewer defects than other classes when adjusted for the class size.

Some recent studies indicate that software artifacts and their relationships evolve into structures that can be described by scale-free networks [5,6]. The reasons are not entirely clear, but a possible explanation is that these networks are formed through “preferential attachment” when software evolves. This could explain the existence of God Classes in many programs.

D. Contextual factors:

Factor	**Implication**
Comprehension Subject’s background (student, academic, industrial developer, etc.)	A design (not code) without a God Class was judged and measured to be better (in terms of time and quality) than a design for the same system with a God Class in a study by Deligiannis [7], but this study only considered academic participants. Similarly, in a controlled experiment with 22 undergraduate students as participants, Deligiannis found that classes without a God Class were more complete, correct and consistent than in a cases with design including God Classes. The results from these studies may be contingent upon the background of the participants and the perspective of maintainability (ease of comprehensionin this case).
Interaction effects with other smells or structural attributes.	From the cases observed in the study by Abbes [3], it is possible to observe that its effect is contingent upon the **interaction effect** by means of **collocation** or potentially **coupling** of God Methods with other smells, like in this case God Class.
Size (LOC)	In the same way as complexity measures should take into account size for its interpretation, this code smell needs to be normalized with respect to size in order to assess its impact.
Architectural role (Controller)	Controllers are responsible for dispatching incoming messages to other classes, so they exhibit several outgoing couplings.

E. Identified False Positive Examples

GUI Library

When a class implements an element of a graphical interface, it often uses a set of well-known libraries (e.g., Swing). An element of a graphical interface can require large amounts of code, due to a large number of the multiple aspects to handle (e.g., shape, colour, position, events). As a result, such class can be frequently identified as God Class. However, this issue is so common that it may not be relevant to report. Moreover, it is also very dependent on the applied toolkit.

Possible detection strategies:

- The class directly implements an interface from *javax.swing.table* package;
- At least one of the ancestors of the class belongs to one the following packages: *javax.swing.table*, *javax.swing.treetable*, or is *javax.swing.JComponent*.

GUI Builder
Often, GUIs are created using graphical editors that generate code. When this code is generated as a single class, it is often detected as God Class.

Test Class
Test classes are generally organized in a specific package, so that the user can simply exclude them from the final project packaging (and from further analyses, e.g. smell detection or metrics collection). During our analysis on the Qualitas Corpus, we found that this is not completely true for a large number of projects. Test classes are not part of the primary analysis, respect to the system design quality (while they may be analyzed with specific approaches).

Possible detection strategies:

- At least one method of the class calls a method of classes that are part of *org.junit* or *junit.framework* packages;
- A class inherits (directly or indirectly) from *org.junit.TestCase* or *junit.framework.TestCase* (or respective classes in other test-related libraries)
- At least one method of the class is annotated as *@Test*

Entity Modeling Class
Entity modeling frameworks, such as EMF in Eclipse, tends to generate large and complex classes to represent the entities. Because the source code of these classes is automatically generated, and is managed through the framework facilities, they should not be considered as God Classes.

Possible detection strategies:

- At least the class or one of the ancestors of the class implements *org.eclipse.emf.ecore.EObject* interface.

Parser Class
A parser generally does much work because of its scope. It tends to be a God Class, but we think it does not represent a problem for system quality, as already discussed in the literature [4]. Splitting the logic of a parser among more classes could make it even harder to understand. Additionally, its code is usually generated by tools like ANTLR, it makes little sense to consider it from a maintenance point of view.

Possible detection strategies:

- The class name contains one of the following words: “*Parse*”, “*Parser*”, “*Parsing*”. We search for the words after we split the class name according to the CamelCase notation for Java classes.
- At least two methods of the class contain one of the following word: “parse”, “Parse”, “parser”, “Parser”, “parsing”, “Parsing”. We search for the words after we split the class name according to the CamelCase notation for Java methods.

- Use a list of code patters to match the parser classes. For instance, ANTLR-generated parsers and lexers can be easily recognized by subclassing the Parser and Lexer classes in the antlr runtime package. For example, see class *org.argouml.language.cpp.reveng.CPPParser* in ArgoUML (ver. 0.34).

Visitor Class

Classes instantiating the visitor design pattern often implements a lot of (usually over-loaded) *visit* methods. These *visit* method can introduce a lot of complexity, and interact with the visited structure, which often merely offers data to the visitor. A single visitor could be re-structured, but if its interface contains many `visit()` directives, it could be difficult to achieve a better modularization.

Possible detection strategies:

- The class name contains one of the following words: “Visit”, “Visitor”. We search for the words after we split the class name according to the CamelCase notation for Java classes.
- At least two methods of the class contain one of the following word: “visit”, “Visit”, “visitor”, “Visitor”. We search for the words after we split the class name according to the CamelCase notation for Java methods.
- The call graph can be analysed to check whether the call sites of visit methods are within accept methods in the visited class, and within this methods, this is passed as a single parameter.

Persistence Class

Classes that handle persistence in a system tend to be large and complex. This is because they usually interact heavily with a database. Implementing the persistence action in a single class may be a good design choice. In this way, it is simple to identify the classes that handle the persistence and to modify them.

Possible detection strategies:

- At least half of the visible methods (i.e., not private methods) call a method in a class in one of the following packages: `java.sql`, `javax.sql` or a package that is part of an ORM framework such as hibernate, batik or Eclipse link.

F. References:

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, “Refactoring: Improving the Design of Existing Code”. Addison Wesley, 1999.

[2] M. Lanza and R. Marinescu, “Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate and Improve the Design of Object-Oriented Systems”. Springer, 2006.

[3] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, “An empirical study of the impact of two antipatterns, Blob and Spaghetti code, on program comprehension”. In: Proceedings of CSMR, 2011, pp. 181–190.

[4] S. M. inherits, D. S. Cruzes, and D. I. K. Sjøberg, “Are all code smells harmful? A study of God classes and brain classes in the evolution of three open source systems”. In: Proceedings of ICSM, 2010, pp. 1–10.

[5] C. Taube-Schock, R. J. Walker, and I.H. Witten. “Can we avoid high coupling?” Proceedings of ECOOP, 2011.

[6] A. Potanin, J. Noble, M. Frean, and R. Biddle, “Scale-free geometry in OO programs”. Communications of the ACM, 48(5), 2005, pp. 99-103.

[7] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd, “A controlled experiment investigation of an object-oriented design heuristic for maintainability”. Journal of Systems and Software, 72(2), July 2004, pp. 129-143.

NAME OF SMELL/ANTI-PATTERN: FEATURE ENVY

A. Definition / Description

A method that seems more interested in another class other than the one it's actually in. Fowler recommends putting a method in the class that contains most of the data the method needs [1]. A metrics based definition proposed by Marinescu [2] is as follows:

AID > 4 and AID top 10 % and ALD < 3 and NIC < 3

Where:
Access of Import Data (AID)
Access of Local Data (ALD)

B. Argumentation: Empirical study

In the study by Li and Shatnawi [3], where several versions of Eclipse were analyzed, **Feature Envy appeared not associated significantly with software defects**.

In the study by D'Ambros et al., 2010 [4], where nonparametric hypothesis testing of the code in the OSSs Lucene, Maven, Mina, CDT, PDE, UI, Equinox was conducted, **neither Feature Envy** nor Shotgun Surgery **was consistently correlated with defects across systems.**

The study by Palomba [5] indicates that the perception of Feature Envy is strongly affected by the fact if the reviewer is the developer who authored the code or not.

C. Contextual factors:

Factor	Implication
Interpretation of maintainability	The effect depends on the actual operationalization, in terms of: • Perceived maintainability effect • Defects
Background: original developer vs. maintainer	In terms of perceived as negative for maintainability, Palomba et al. states that the perception of Feature Envy as an important aspect for maintainability depends on the background of the developer (if s/he is the original developer or not).

D. Identified False Positive Examples

Visitor DP

In the Visitor design pattern, `visit()` methods can be detected as Feature Envy instances, since they inspect the contents of the visited element. This is an essential part of this pattern, and cannot be considered an issue for the system.

E. References:

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, “Refactoring: Improving the Design of Existing Code”. Addison Wesley, 1999

[2] M. Lanza and R. Marinescu, “Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate and Improve the Design of Object-Oriented Systems”. Springer, 2006.

[3] W. Li and R. Shatnawi, “An Empirical Study of the Bad Smells and Class Error Probability in the Post-Release Object-Oriented System Evolution”, J. Systems Software, 2007, vol. 80, no. 7, pp. 1120-1128.

[4] M. D’Ambros, A. Bacchelli, and M. Lanza, “On the Impact of Design Flaws on Software Defects.” Proceedings ICQS, 2010, pp. 23-31.

[5] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, “Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells”, Proceedings ICSME 2014, pp.101-110.

NAME OF SMELL/ANTI-PATTERN: DISPERSED COUPLING

A. Definition / Description
Lanza et al. [1]: *“This is the case of an operation which is excessively tied to many other operations in the system, and additionally these provider methods that are dispersed among many classes. In other words, this is the case where a single operation communicates with an excessive number of provider classes, whereby the communication with each of the classes is not very intense i.e., the operation calls one or a few methods from each class.”*

B. Argumentation via Identified False Positive Examples from [2]

Test Class Method
The same we introduced for Shotgun Surgery. Sometimes a method of a test class can be detected as Dispersed Coupling, because it calls many methods in different classes.

Possible detection strategies:

- At least one method of the class calls a method of classes that are part of *org.junit* or *junit.framework* packages;
- A class inherits (directly or indirectly) from *org.junit.TestCase* or *junit.framework.TestCase* (or respective classes in other test-related libraries)
- At least one method of the class is annotated as *@Test*

C. References:
[1] M. Lanza and R. Marinescu, “Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate and Improve the Design of Object-Oriented Systems”. Springer 2006.

[2] F. Arcelli Fontana, V. Ferme, and M. Zanoni, “Filtering Code Smells Detection Results”. *Proceedings ICSE 2015.*

NAME OF SMELL/ANTI-PATTERN: DUPLICATED CODE

A. Definition / Description
Same or similar code structure repeated within a class or between classes [1].

B. Synonyms:
Clones

C. Argumentation:

C.1. Empirical studies
Kapser et al. [2] reported on a study where academic experts judged whether they considered duplicated code harmful or not in code pieces of two OSSs. They concluded that **some instances of duplicated code** were connected with a code that was considered to represent a **good style** of programming and **contributed to a system‘s stability**.

Monden et al. [3] performed an analysis of a COBOL legacy system and concluded that **cloned (duplicated) modules were more reliable** but **required more effort** than non-cloned modules.

Rahman et al. [4] performed a descriptive analysis and nonparametric hypothesis testing of code and bug tracker in the OSSs Apache httpd, Nautilus, Evolution and Gimp. They found that most of the **defective code was not significantly associated with duplicated code**. The code that was duplicated less frequently across the system was more error-prone than the code that was duplicated more frequently.

Lozano et al. [5] compared the maintenance effort of methods in the periods when they contained and did not contain a clone. They found that there was **no increase in the maintenance effort in 50%** of the periods after methods transitioned from not containing to containing a clone. However, **when there was an increase in effort, this increase could be substantial**. They report that the effect of clones on maintenance effort depended more on the areas of the system where the clones were located than on the cloning itself.

Kim et al. [6] reported on the analysis of two medium-sized 23 open-source libraries (Carol and dnsjava) and concluded that 36% of the total amount of code that had been duplicated **changed consistently** (i.e., they remained as identical duplicates via simultaneous updates), while the remaining part evolved independently.

Jürgens [7] performed a descriptive analysis of 3 industrial C# systems, 1 OSS Java system and 1 industrial COBOL system. In the Java and C# code, the inconsistently changed duplicated code contained **more faults** than average code. In the **COBOL code**, inconsistent changes **did not lead to more faults**.

Göde and Harder [8] investigated how often changes occurred in duplicated code. They found that consecutive changes to duplicated code tend to occur and that **the majority of these changes were consistent** (i.e., simultaneous updates were made in the original and the duplicated code) or were intentionally inconsistent. They also found that "unwanted" inconsistencies **led only to defects with low severity**.

Moreover, Bettenburg et al. [9] analyzed the effect of inconsistent changes to code clones on software quality at release level, and found that cloning **does not affect the post release quality** of the system they studied

The results by Selim et al. [10] showed that cloned code is more error prone than non-cloned code in ArgoUML, but **not** in Ant.

Kapser and Godfrey [11] identified different cloning patterns (thus, they came up with a clone taxonomy). They found that **many types of clones were helpful in improving the quality** of the system. They found that as much as 71% of the clones had a positive impact on the maintainability of the software. They underlined the significance of managing code clones by (automatically) keeping the relevant clones "in sync", and they suggested static analysis techniques to identify "cloning hotspots".

Krinke [12] compared the age of cloned code to the age of non-cloned code in three open source products: ArgoUML, JBoss and JEdit, by observing 200 weeks of evolution, and found that **cloned code is more stable** (i.e., less volatile and 'older') than non-cloned code. They conclude that **it cannot be generally assumed that the maintenance of cloned code is more expensive** than the maintenance of non-cloned code. Göde and Harder [13] replicated Krinke's study and could confirm his results.

C.2. Theoretical explanation

Kim et al. [14] pointed out that many clones are intentionally introduced to code due to language restrictions. They indicated that refactoring for removing clones might not always actually improve the software. Alternatively, they suggest applying "clone management", for example, to keep them "under scrutiny" and see how they evolve over time. IDEs can assist by making developers aware of duplication (e.g., to be informed about the clones introduced deliberately due to hard time constraints, repeated copy and paste practices, etc).

D. Contextual factors:

Factor	**Implication**
Interpretation of maintainability	The effect depends of the actual operationalization. Duplicated code seems to be a false positive if the effects are interpreted in terms of: • Perceived maintainability • Reliability • Consistently changed (reduced side-effects)

	• Defects However, if the effects are interpreted in terms of effort, then it may be considered a true positive.
Industrial vs. Open source	The studies involving Open Source systems did not find effects of clones on defects, however in the industrial system there was en effect found.
Platform/Programming language	Monden and Jürgens results coincide with respect to COBOL systems. However, for Java systems duplicated code contained more faults. The same goes for the study by Selim.
Development method	Generated code is not a smell since it does not undergo direct maintenance.
Tools	The availability of effective refactoring browsers that allow bulk updates makes clones easier to accept.
Optimisations	In some cases, clones are used to optimize performance. A good example is inlining performed by the Java compiler, which might be detected as a clone by byte-code based analysis tools.

D. References:

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the Design of Existing Code". Addison Wesley, 1999.

[2] C. Kapser and M. Godfrey, "'Cloning considered harmful' considered harmful: patterns of cloning in software." Empirical Software Engineering, vol. 13(6), 2008, pp. 645-692.

[3] A. Monden et al., "Software quality analysis by code clones in industrial legacy software." Proceedings IEEE Symposium on Software Metrics. 2002, pp. 87–94.

[4] F. Rahman and P. Devanbu, "Clones: What is that smell?". Proceedings ICSE 2010, pp. 72-81.

[5] A. Lozano and M. Wermelinger, "Assessing the effect of clones on change-ability." In: Proceedings ICSM 2008, pp. 227-236.

[6] M. Kim et al., "An empirical study of code clone genealogies." In: European Softw. Eng. Conf. and ACM SIGSOFT Symposium on Foundations of Softw. Eng. 2005, pp. 187-196.

[7] E. Juergens et al, "Do code clones matter?". Proceedings ICSE 2009,

pp. 485-495.

[8] N. Göde and J. Harder, "Oops! . . . I Changed It Again.", http://dl.acm.org/citation.cfm?id=1985408

[9] Bettenburg et al., "An empirical study on inconsistent changes to code clones at the release level", http://sail.cs.queensu.ca/publications/pubs/bettenburg-wcre09.pdf

[10] Selim et al., “Studying the impact of clones on software defects”, http://www.researchgate.net/publication/221200439_Studying_the_Impact_of_Clones_on_Software_Defects/file/32bfe512ee4e124146.pdf

[11] Kapser and Godfrey, “Supporting the analysis of clones in software systems: a case study.”, https://plg.uwaterloo.ca/~migod/papers/2005/icsm05-cloning-jsme.pdf

[12]: J. Krinke, “Is Cloned Code more stable than Non-Cloned Code?”, http://www0.cs.ucl.ac.uk/staff/j.krinke/publications/iwsc11.pdf

[13]: Göde and Harder, “Clone stability”, http://www.researchgate.net/publication/224227182_Clone_Stability

[14]: Kim et al., “An Ethnographic study of copy and paste programming practices in OOPL”, http://tlau.org/research/papers/M.Kim-EthnographicStudyofCopyPaste-ISESE.pdf

[15] A. Yamashita, “Attack of the clones: How much is science and how much is fiction?” http://fagblogg.mesan.no/attack-of-the-clones

NAME OF SMELL/ANTI-PATTERN: DATA CLUMPS

A. Definition / Description
Clumps of data items that are always found together whether within classes or between classes [1].

B. Argumentation: Empirical study
In the study by Yamashita [2] involving four medium-sized industrial Java systems, maintenance-related problems were recorded and associated to source files causing them. Data Clump displayed **lower** odds of being contained in a file causing problems during maintenance.

C. Contextual factors:
The effect of Data Clumps could be conditioned by the fact that the system observed was an industrial, medium to small size system. The study by Yamashita et al. [3] indicated that the constellation (and potential problems) caused by interconnected Data Clumps may differ across industrial systems and Open Source systems.

Factor	**Implication**
Open Source vs. Industrial	Is less likely to be problematic in Industrial systems, due to stronger emphasis on encapsulation [3].
Domain	Data Clumps in information systems are spotted quite fast and abstract via extract class refactoring. In systems that are dependent of configuration files (see code analysis in [3]) such as search engines (Elastic Search) and algorithmic-intensive applications (Mahout), it could potentially be more problematic.

As a side note, we can add that the **Primitive Obsession** smell is very similar to Data Clumps, in terms of the causes of its introduction. For this reason, it is subject to the same false positive categories.

D. References:

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the Design of Existing Code". Addison Wesley, 1999.

[2] A. Yamashita, "Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data". Empirical Software Engineering, vol. 19(4), pp. 1111-1143.

[3] A. Yamashita, M. Zanoni, F. Arcelli Fontana, and B. Walter, "Inter-smell relations in industrial and open source systems: A replication and comparative analysis", Proceedings ICSME 2015, pp.121-130

NAME OF SMELL/ANTI-PATTERN: DATA CLASS

A. Definition / Description
Classes that have fields, getting and setting methods for the fields, and nothing else; are dumb data holders; are being manipulated in far too much detail by other classes [1].

A class that consists only of public data members, or of simple getting and setting methods. This lets clients depend on the mutability and representation of the class [2].

"Dumb" data holders without complex functionality, but other classes strongly rely on them. The lack of functionally relevant methods may indicate that related data and behavior are not kept in one place; this is a sign of a non-object-oriented design. [3]

Data Class describes an invalid class that has no responsibility except of storing the state. The state is either immutable or needs to be directly manipulated by the clients.

B. Synonyms:
Data Transfer Object [6]

C. Argumentation:

C.1. Empirical study: Li & Shatnawi [4] – large Open Source Java systems: Data Class was not associated significantly with software faults. Sjøberg & Yamashita [7]: Data Class was not related with increased maintenance effort, and Khomh et al. [5]: presence of Data Class is not correlated with change proneness.

C.2. Theoretical explanation: Data Classes break the single responsibility principle, as external classes directly manipulate the state of the object. However, due to technology limitations, it's sometimes desirable to wrap data into a single object without granting it functions to manipulate the data, just to transfer it together. In most cases it is justified by performance tuning efforts (transferring a single object is less expensive than sending several attributes individually). Data Transfer Object is an enterprise pattern that describes an object for transferring data between tiers or processes. Fowler defined a Data Transfer Object in [6].

D. Contextual factors:

Factor	**Implication**
Distributed processing/IPC/multi-tier application	Data Classes group individual data items for improving performance of data transfer between application layers.
Perspective on maintenance (change proneness)	Khomh et al. [5] analyzed the source code of Eclipse IDE and found that, in general, classes containing the Data Class code smell were changed more often than classes without it. The effect of the changes on the total cost however, are contingent of the evolution of the system (e.g., if there are more changes needed on the abstractions used in the system) and the degree of automation and cost of the individual changes of the Data Classes (e.g., if the changes are automated, then is of no importance if the classes are changed very often).

E. Identified False Positive Examples

Exception Handling Class
Classes that handle exception can often be identified as instances of Data Class. These classes cannot be considered "smelly" because they represent a good design choice.

Possible detection strategies:
- At least one of the ancestors of the class is *java.lang.Throwable*.

Serializable Class:
Classes that have to be serialized often "smell" like a Data Class. This is because they are data holders, without complex functionalities. We think that if a class has been explicitly declared serializable by a developer, it does not represent a problem for the quality of the system. It is a design choice, because of a precise requirement: the object needs to be serialized.

Possible detection strategies:
- At least one of the ancestors of the class is *java.io.Serializable*.

Test Class:
Test classes are generally organized in specific packages, so that the user can simply exclude them from the final project packaging (and from detection results). During our analysis on the Qualitas Corpus, we found that this is not true for a large number of projects. Test classes are not part of the primary analysis, respect to the system design quality (while they may be analyzed with specific approaches).

Possible detection strategies:
Possible detection strategies:

- At least one method of the class calls a method of classes that are part of *org.junit* or *junit.framework* packages;
- A class inherits (directly or indirectly) from *org.junit.TestCase* or *junit.framework.TestCase* (or respective classes in other test-related libraries)
- At least one method of the class is annotated as *@Test*

Logger Class:
Classes used to wrap and offer logger functionalities can be identified as instances of Data Class. This is because they generally have not very complex methods and some getter and setter methods to configure and retrieve logger properties. We think that wrapping logger functionalities in a class is not an issue for the system quality.

Possible detection strategies:

- The class name contains one of the following words: “Log”, “Logger”. We search for the words after we split the class name according to the CamelCase notation for Java classes.
- At least two methods of the class contain one of the following word: “log”, “Log”, “logger”, “Logger”. We search for the words after we split the class name according to the CamelCase notation for Java methods.

F. References:

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, “Refactoring: Improving the Design of Existing Code”. Addison Wesley, 1999

[2] W. Wake, “Refactoring Workbook”. Addison Wesley, 2003.

[3] M. Lanza and R. Marinescu, “Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate and Improve the Design of Object-Oriented Systems”. Springer, 2006.

[4] W. Li and R. Shatnawi, “An Empirical Study of the Bad Smells and Class Error Probability in the Post-Release Object-Oriented System Evolution”, J. Systems Software, 2007, vol. 80, no. 7, pp. 1120-1128.

[5] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc. “An Exploratory Study of the Impact of Code Smells on Software Change-proneness.” Proceedings WCRE 2009, pp. 75-84.

[6] M. Fowler, “Patterns of Enterprise Application Architecture”. Addison-Wesley, 2002.

[7] D. Sjøberg, A. Yamashita, B. Anda, A. Mockus, and T. Dybå, „Quantifying the Effect of Code Smells on Maintenance Effort. IEEE Trans. on *Software Engineering,* (99), 1. doi:10.1109/TSE.2012.89.

NAME OF SMELL/ANTI-PATTERN: SHOTGUN SURGERY

A. Definition / Description

Fowler et al. [1]: *"You whiff this when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes."*

Lanza et al. [2]: *"This design disharmony means that a change in an operation implies many (small) changes to a lot of different operations and classes."*

B. Argumentation via Identified False Positive examples from [3]

Exception Handling Method

Methods in classes that handle exceptions can often be identified as infected with Shotgun Surgery. This happens because an exception can be caught in different parts of the system, hence many classes access the methods offered by the exception handling class. This can be considered a good design choice, as the structure of Exceptions is usually stable.

Possible detection strategies:

- At least one of the ancestors of the class that contains the method is *java.lang.Throwable*.

Test Class Method

The same we introduced for God Class. Sometimes a method of a test class can be detected as a Shotgun Surgery instance, because many other test classes use them, e.g., a method that generate or retrieve data used by a large number of test cases.

Possible detection strategies:

- At least one method of the class calls a method of classes that are part of *org.junit* or *junit.framework* packages;
- A class inherits (directly or indirectly) from *org.junit.TestCase* or *junit.framework.TestCase* (or respective classes in other test-related libraries)
- At least one method of the class is annotated as *@Test*

Getter/Setter Method

Getter and setter methods can be more complex than the basic one-line implementation. Some of them interact with other methods of the same class of from other classes. We think that this type of methods do not represent a problem because they are more stable compared to other methods (especially one-liners).

Possible detection strategies:

- The method contains one of the following word: “set”, “get”. We search for the words after we split the class name according to the CamelCase notation for Java methods.

C. References:

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, “Refactoring: Improving the Design of Existing Code”. Addison Wesley, 1999

[2] M. Lanza and R. Marinescu, “Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate and Improve the Design of Object-Oriented Systems”. Springer, 2006.

[3] F. Arcelli Fontana, V. Ferme, and M. Zanoni. “Filtering Code Smells Detection Results”. Proceedings ICSE 2015

NAME OF SMELL/ANTI-PATTERN: CIRCULAR DEPENDENCIES BETWEEN PACKAGES

A. Definition:
Cyclic dependencies between program artifacts – in the case of Java classes and in particular packages – are widely portrayed as an anti-pattern [1,2].

B. Argumentation:

There are several studies indicating that cyclic dependencies are surprisingly common in many real-world programs [3,4].

Fallari et al. argue that package cycles that form in branches of the package containment tree are less critical as they are organically formed when packages grow and are reorganized into subpackages [5].
Al Mutawa et al. studied the topology of cyclic structures in the Java package graph (referring to strongly connected components as tangles) and found that most tangles do form in branches of the package containment tree, centered on a parent package that acts as a hub [6]. This, in conjunction with [5], offers an explanation why such cycles are so ubiquitous in real-world programs.
However, Oyetoyan et al. found no correlation between cycle topology and changes frequency of the classes involved in cycles [7]. They did however find evidence that classes near cycles are more prone to change.

Possible detection strategies for the topology of the cycles and their position within the package containment tree are based on the metrics proposed in [6].

C. Identified False Positive examples:
Visitor
The structure of the Visitor design pattern can create cycles the visitor and visited elements. This issue is detailed in the next section, and visible in Figure 1.

Since the Visitor design pattern is an established solution to the problem of decoupling algorithms and crawling of data structures, and a solution to the unavailability of double dispatching in different object-oriented programming languages, it can be considered a good choice, even when creating circular dependencies. Anyway, it can be advisable to try moving the classes (when possible) in a single package, to avoid creating circular dependencies among packages, which generate many different maintenance problems.

Abstract Factory
This creational design patterns can incur in circular dependencies. This happens when the abstract factory knows about a default concrete factory, for example. This generates also Subtype Knowledge and is therefore detailed in the next section. This kind of structure can be accepted, since most more correct solutions will be much more complex.

Parser Class
Parsing code generated by tools like ANTLR tend to rely on generated data structure that are tangled and interdependent. This is not a problem for the quality of the system for different reasons:

- As already discussed, generated code is not directly maintained, and therefore should not be addressed by static analysis
- The generated code and data structures are usually generated in a single package or in a separate group of packages, it can be considered as an external library from the point of view of the project.

D. References

[1] D. L. Parnas, "Designing software for ease of extension and contraction", *IEEE Trans. on Software Engineering,* no. 2(1979), pp. 128–138.

[2] R. C. Martin, "Design principles and design patterns," *Object Mentor*, vol. 1, p. 34, 2000.

[3] H. Melton, and E. Tempero, "An empirical study of cycles among classes in Java". *Empirical Software Engineering*, *12*(4), 2007, pp. 389-415.

[4] J. Dietrich, C. McCartin, E. Tempero, and S. M. A. Shah, "Barriers to modularity – an empirical study to assess the potential for modularisation of java programs". *Research into Practice – Reality and Gaps*, Springer, 2010, pp. 135-150.

[5] J. R. Falleri, S. Denier, J. Laval, P. Vismara, and S. Ducasse, "Efficient retrieval and ranking of undesired package cycles in large software systems". *Objects, Models, Components, Patterns*, Springer 2011, pp. 260-275

[6] H. Al-Mutawa, J. Dietrich, S. Marsland, and C. McCartin, "On the shape of circular dependencies in java programs". Proceedings ASWEC 2014, pp. 48-57.

[7] T. D. Oyetoyan, J. R. Falleri, J. Dietrich, and K. Jezek, "Circular dependencies and change-proneness: An empirical study". Proceedings SANER 2015, pp. 241-250.

SUBTYPE KNOWLEDGE

A. Definition:

Subtype knowledge is an anti-pattern proposed by Riel [1]. It directly violates a core object design principle – dependency inversion [2], and often leads to cyclic dependencies between packages if classes participating in this anti-pattern are located in different packages.

B. Argumentation:

Empirical studies have demonstrated that instances of subtype knowledge are common in real world programs [3]. We observed that the use of certain design patterns causes unintended instances of this anti-pattern, which may be considered false positives.

B.1 Subtype Knowledge and Visitor

The use of the popular Visitor design pattern [4] directly creates instances of this anti-pattern. The visitor pattern is a popular to “plug” functionality into complex data structures such as trees and graphs, and is widely used in parser (AST) APIs, and to address limitations of mainstream programming languages like Java, with respect to method dispatch [5].

A visitor pattern instance consists of *visitors* and visited *elements*. There are abstract and concrete visitors and elements. The visitors reference all concrete element types as parameters in the (overloaded) visit methods, while both the abstract and the concrete element types use the abstract visitor type as parameter type in the accept methods.

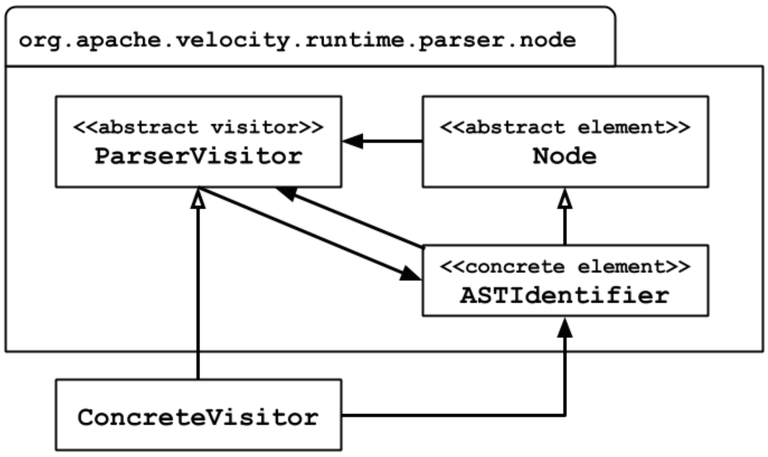

Figure 1: Visitor instance used in Apache Velocity

An example of a Visitor instance is shown in Fig. 1. This example has been extracted from the Apache Velocity project. Velocity uses a domain specific language (DSL) to represent templates, and this code is part of the parser for this DSL. Note that the number of concrete elements is typically large: in this example there are 33 such classes, each representing a particular AST node type. This can result in large strongly connected components that represent tangles of many classes and packages with multiple circular dependencies amongst them.

Interestingly, an acyclic version of the visitor has been proposed [6]. But acyclic visitors are even more complex than standard visitors as an additional role is required, and it appears that they are not widely used. The abstract type has a dependency chain on one of its own subtypes. This is caused by the inheritance relationship between the concrete elements (such as ASTIdentifier) and the abstract element (Node).

Possible detection strategies for instance of Visitors include naming pattern and call site / call graph analysis (details have been discussed above).

B.1 Subtype Knowledge and Singletons

The purpose of the Singleton pattern [4] is to provide classes that can by design have only one instance. Singletons are often used to provide a single access point for global services, such as object creation (in combination with the factory pattern [4]), logging, or access to configuration information. There are good reasons to use Singleton in combination with abstraction. For instance, an abstract class is defined first that provides the specification of the services provided by the singleton, and also provides access to the actual singleton instance via a static access method. However, the actual singleton instance is an instance of a concrete, instantiable subclass. If the Singleton acts as a factory, this separation results in an abstract factory.

Because the abstract singleton base class references the concrete subclass, a subtype knowledge antipattern occurs. An example can be seen in Azureus/Vuze [2]. Here *$p.impl.NetworkAdminImpl* extends *$p.NetworkAdmin*, and *$p.NetworkAdmin* references *$p.impl.NetworkAdminImpl,* as can be seen in the code snippet below.

```
public static synchronized NetworkAdmin getSingleton () {
   if ( singleton == null ){
     singleton = new NetworkAdminImpl () ;
  }
  return( singleton );
}
```

[2] $p is short for *com.aelitis.azureus.core.networkmanager.admin*

As the abstract class and its concrete subclass are in separate packages, this also manifests a circular dependency between packages instance.

It is often argued that this is indeed an anti-pattern, and there are several widely used approaches to break the reference from the abstract to the concrete class, such as using reflection (with the name of the service provider concrete class read from a configuration files, deployment descriptor file or service provider component metadata accessed via service loaders), or to install the singleton instance when the concrete class is loaded or instantiated (this approach is used when JDBC drivers are registered). However, there are good reasons not to do this as any of these methods introduces additional complexity and therefore has an impact on maintainability. Also, these approaches just hide or delay the dependency, but don't avoid it. On the other hand, if there is one default implementation that is used in almost all cases, then the approach used by Azureus/Vuze looks like the best solution.

C. References

[1] D. L. Parnas, "Designing software for ease of extension and contraction"*, IEEE Trans on Software Engineering*, no. 2, pp. 128-138, 1979.

[2] R. C. Martin, "Design principles and design patterns," *Object Mentor*, vol. 1, p. 34, 2000.

[3] J. Dietrich, C. McCartin, E. Tempero, and S. M. A. Shah, (2010). "Barriers to modularity-an empirical study to assess the potential for modularisation of java programs". *Research into Practice–Reality and Gaps*, Springer 2010, pp. 135-150.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1995.

[5] J. M. Vlissides, "Extensible and efficient double dispatch in single- dispatch object-oriented programming languages," Apr. 13 2004, uSsPatent 6,721,807.

[6] R. C. Martin, "Acyclic visitor", *Pattern languages of program design, Vol. 3,* Addison-Wesley 1997, pp. 93-103.