Influence of programming style in transformation bad smells: Mining of ETL Repositories

Nicolás Bonet[1a], Kelly Garcés[1a], Rubby Casallas[1a], María Elsa Correal[1b], and Ran Wei[2]

[1] Universidad de los Andes, Bogota D.C. Colombia, School of Engineering. Department of Systems and Computing Engineering[a] and Departament of Industrial Engineering[b]
{n.bonet2476, kj.garces971, rcasalla, mcorreal}@uniandes.edu.co
[2] The University of York. UK. Department of Computer Science
{ran.wei}@cs.york.ac.uk

Abstract. Bad smells affect maintainability and performance of model-to-model transformations. A number of studies have defined a set of transformation bad smells, and proposed techniques to recognize and —according to their complexity— fix them in a (semi-)automated way. In education, it is necessary to make students aware of this subject and provide them with guidelines to improve the quality of their transformations. This paper presents some common bad smells in model transformations written by master students from *Universidad de los Andes* and compares them with that of publicly available repositories of ETL transformations, for the purpose of knowing whether programming style affects the incidence of smells. Three contributions are presented: i) Two new bad smell patterns enriching the existing catalogs; ii) A process that includes the automated extraction of transformation metrics and bad smells metrics from the repositories, and a statistical analysis that helps in identifying the relations between such metrics; and iii) A tool that supports the process. By applying our approach on the datasets, we discuss whether it is easier for students with imperative programming language background to make use of appropriate declarative constructs of a transformation language compared to imperative ones. We conclude that students must be encouraged and guided to use declarative constructs whereas possible when developing declarative transformations, that results in artifacts that are more maintainable and with a better performance.

Keywords: Model-Driven Engineering, Model Transformation, Quality, Metric, Bad Smells, Epsilon Transformation Language (ETL), Educational Purpose.

1 Introduction

Model-Driven Engineering (MDE) has gained some popularity in industry because companies and developers are beginning to adopt MDE approaches to

reduce software complexity and improve its maintainability. In this context, universities play a major role since they teach MDE to students, who become software developers after they graduate. It is essential to teach and students about the best practice of MDE for them to become better software developers/architects in the future.

Whilst MDE becomes popular, it has pointed out that measuring the quality of model management programs, especially model-to-model (M2M) transformations, is crucial. We have identified several studies [7,8,3,2,11,10] aiming to find a relation between traditional software quality attributes and M2M transformations, in effort to determine which aspects of the languages have a direct impact on these attributes. Additionally, we have found research that studies how bad smells (bad software practices) can affect the maintainability and performance of transformations [13,12,9,1,4]. Most of the available studies target transformations written in ATL, but a few tackle other languages such as Epsilon Transformation Language (ETL)[1], QVT and Henshin.

To the best of our knowledge, there has been no research that studies how coding styles (either imperative or declarative) may affect the number of bad smells found in transformations. Thus, we propose an approach (Section 2) that: i) statically analyzes repositories of transformations; ii) extracts transformation metrics (metrics of the language constructs); iii) detects bad smells based on a pre-defined catalog; and iv) establishes a relation between these two kind-of metrics, based on statistical techniques (Section 4). Other contributions of our work are: two new bad smells that enrich the existing catalog found in the literature and a tool that operationalizes the process and is built on top of the Epsilon languages and Haetae[2].

We have applied the approach to a case study (Section 3) that includes ETL transformations developed by students from *Universidad de los Andes* and by other stakeholders (the contributors of ETL from University of York and Github users) for the sake of comparison. From the results (Section 5), we were able to identify the quality of the transformations made by our students, and code styles to increase such a quality. Section 6 introduces the implications that the analysis has for instructors and students, and summaries future work.

2 Mining ETL Repositories: Process Overview

The repository mining consists of three steps. In the first step, a set of transformation metrics are calculated and bad smells are identified on an available corpus of files. In the second step, a report is automatically generated discriminating the information obtained for each transformation of the corpus. Finally, the collected data is analyzed to identify relations between transformation metrics and bad smells to advise instructors about which could be the weakest points

[1] Epsilon Transformation Language (ETL). URL: `https://www.eclipse.org/epsilon/doc/etl/`. July 13, 2017.

[2] Epsilon Labs, haetae. URL: `https://github.com/epsilonlabs/haetae`. July 13, 2017.

with respect to quality on their students work. In the remaining sections of this paper, we present how this process has been instantiated in a case study.

3 The case study setup

3.1 Case study context

This study is part of a research which aims at improving the programming skills of masters-degree students that attend MDE lectures, a vast majority of the students (20 in average each semester since 2014) have no experiences on MDE. This lecture has been part of the MSc Software Engineering course and we are constantly looking for better ways of improving the teaching methodologies to increase the quality of the software produced by students. In this context, the following research questions were asked:

- **RQ1:** What is the distribution of bad smells across datasets?
- **RQ2:** What is the variety of code smells in the model transformations involved in the case study?
- **RQ3:** How the total number of bad smells are influenced by programming style (i.e., declarative or imperative)?

The transformation metrics used as the base line of the study are the following: the number of: *matched/lazy rules, operations with/without context, calls to lazy rules per rule, calls to operations per rule, variables per rule, if, loops, unused operations and unused parameters*[3].

These metrics were chosen as we wanted to identify how the transformation strategy was relevant to the amount of bad smells that could be found on the dataset. Based on [6] and our own experience, we have identified that transformations developed using an *imperative* programming style tend to majorly use *operations*, *if*, and *loops*. This also involves transformations that have few matched rules, but the body of those rules is built using plenty of imperative constructs. On the other hand, transformations that contain *matched rules*, *guards*, and *OCL queries* show a more *declarative* approach.

3.2 Characteristics of the dataset

To perform the study we extracted a total of 286 ETL transformations from multiple sources, they were grouped in three different datasets referred to as *Uniandes, York,* and *Github*. In the *Uniandes* dataset, the transformations are used in reverse and forward MDE projects. This dataset is publicly available[4] and can be used for further investigation. In the *York* dataset, transformations are built by developers with vast experience in ETL, they were retrieved from the official Epsilon Website.

[3] Both matched and lazy rules specify the way in which target model elements must be generated from source model elements. The execution of matched rule is scheduled by the engine depending on the type of elements that are applicable, whereas the execution of lazy rules is determined by the programmer.

Finally, the transformations of the *Github* dataset are the ones found in public GitHub repositories. The metamodels involved in these transformations are unknown as well as the experience of developers. With respect to the datasets' size, *Uniandes* has 191 files and 27526 LOC, *York* 23 files and 1550 LOC, and *Github* 72 files and 26948 LOC.

4 Bad smells detection

4.1 Catalog of bad smells

We followed 4 steps to build our bad smells catalog: i) Review the literature that categorizes transformation bad smells, i.e., [13,12,5] ; ii) Adapt bad smells in related works to ETL, since there are few differences between ETL constructs and the ones from other M2M transformation languages (e.g. ATL); iii) Perform code review of the dataset to identify new smells out of the scope of previous work; iv) Build a consolidated catalog.

Reviewing previous catalogs of bad smells [13,12,5] generates 27 bad smells (overlapped smells are omitted). In our work, we consider 6 of these previously categorized bad smells and contribute 2 new bad smells observed in our datasets. The reason for including just 6 of 27 is that these can be discovered via static analysis, in a fully automatic way. In contrast, for the discovery of the rest of bad smells, it is necessary to have more information such as execution traces (e.g., that indicate when an operation is computational intensive) or some knowledge about the transformation context (e.g., to know if rules have meaningful names).

The 8 bad smells of our catalog can be classified into two categories, defined in [13]: i) *Restructuring* (these bad smells are related with how the transformation is built and structured); and ii) *OCL Optimization* (As ETL is an OCL-based language, optimization of those queries is extremely important to improve the transformation performance). For each bad smell we give an abbreviation, a description, and how it is discovered. In addition, for existing bad smells we mention the identifier managed in the original source [13] for cross-referencing purposes, and for new bad smells we give an example. We summarized bad smells taken from previous work in a Table available in this link[5], this section focuses on new smells.

4.2 New bad smells

Restructuring: Imperative filtering

- **Abbreviation:** IEO.
- **Description:** A *for* statement has an *if* to prevent the logic being applied to all elements, this is less efficient than an OCL first-order logic expression and is also harder to understand.
- **Discover Algorithm:** The discoverer checks each *for* statement made on the transformation and if the *for* logic is wrapped inside an *if*, then it increases the count for this bad smell.

[4] GITHUB, ETLMetrics. URL: `https://github.com/NicolasBonet/ETLMetrics/`. July 13, 2017.

– **Example:** An example is available at this link[6].

OCL Optimization: Imperative element creation in loops

– **Abbreviation:** CNL.
– **Description:** Creating elements inside loops are discouraged as the creation of multiple elements can be achieved by rules, the use of guards is useful to prevent it from happening over all elements if necessary.
– **Discovery Algorithm:** The discoverer checks each loop statement (i.e., while and for) and verifies if at least one occurrence of the reserved word "new" —which is intended to create new elements— is found in the block.
– **Example:** An example is available at this link[6].

5 Statistics and data analysis

In this chapter, we present the results of our case study taking into account the stated research questions.

5.1 Distribution of bad smells

RQ1: *What is the distribution of bad smells across the datasets?*. We found that 118 out of 286, or 41.26% of the transformations, have no bad smells, while 13.64% present more than 10 occurrences. We also established the occurrence-rate of bad smells by transformation (λ_t), which is defined as follows:

$\lambda_t = \frac{\sum_{i=0}^{I} badSmells(i)}{i}; I = Transformations$

We saw that *Uniandes* presents rates lower than *Github* and higher than *York*. From these results, and taking into account that the main promoters of ETL are at the University of York, it is worth to define patterns that favor quality in conjunction with them (Section 6 describes three patterns as a starting point).

5.2 Variety of bad smells

RQ2: *What is the variety of smells found in the transformations involved in the case study?*. The *discoverer* detected the occurrences of the bad smells in the datasets and the results follow: There are three types of bad smells with the highest occurrences across all datasets (see Table[7] for a brief list of these smells): i) *CSF* (a chain of select/first in OCL is less efficient than using selectFirst); ii) *IEO* (if statements embedded into a for are less efficient than OCL filters), and iii) *CNL* (creation of new elements in loops). Each of these bad smells with more than 200 occurrences found in the entire set of transformations. On the other hand, the bad smells referred to as *TOC* (trivial operations called once) and *REB* (rule body is embedded into if blocks) were the less common ones, each one with less than 60 occurrences. When the data is interpreted dataset by

[5] GITHUB, Bad smell catalog. URL: `https://github.com/phillipus85/ETLMetrics/blob/master/EduSymp2017/resources/Catalog.pdf`. July 13, 2017.
[6] GITHUB, IEO and CNL code. URL: `https://github.com/phillipus85/ETLMetrics/blob/master/EduSymp2017/resources/IEOandCNL.pdf`. July 13, 2017.

dataset, it can be seen that for *Uniandes* the error *DOE* (duplicated and complex OCL expressions) is the most common one. For *York*, the most common smell is *TOC* (trivial operations called once). Note that the discoverer allows one to configure the number of lines that an operation should have to be considered are trivial. For this experiment, we set this threshold to 1-3 lines based on what we have observed in the developed code. Finally, in *Github* the occurrences are quite similar to the overall result.

5.3 Influence of programming style on bad smells incidence

RQ3: *How the total number of bad smells are influenced by the programming style (declarative or imperative)?*. To answer this question we made a Negative Binomial Regression between the transformation metrics defined in Section 3 (matched/lazy rules, operations, etc.) and bad smells metrics. This regression helps us to measure the influence of a particular construct on the occurrence-rate of bad smells (i.e., λ_t) by means of a Relative Risk (RR). In our context, a RR is a way to compare occurrence-rate of bad smells in transformations that uses a certain language construct with that of transformations not using it. We calculated RR values for each construct, one at a time, based on a Negative Binomial Regression Model. Since RR is a quotient, an RR of 1.20 for a construct means that using that particular construct increases bad smells by 20%, while a RR of 1 means that the construct has not influence in bad smells. The influence is assessed testing if RR is equal to 1, significantly.

We applied the statistical model to metrics coming from the three datasets —as a whole set— and from each dataset, separately. Table 1 shows the results for the entire set and they are similar to the results of each particular dataset. The table consists of the following three columns: the language construct, RR and significant effect. In fact, it is the latter that indicates a strong relation between the number of bad smells and the use of *variables per rules*, control statements like *while, for, if* and *calls to operations per rule*. There is a trade-off between the transformations quality and the time to produce a fully functional MDE code generator. For the students, it is easier to use an imperative style (which is the their background programming style) than a declarative approach that they do not manage to master in 4 months (duration of the course).

Table 1 also shows a strong relation between *unused parameters/operations* and bad smells, this is due to the project of the course is developed in an incremental way and the students were going through the learning curve. Therefore, early versions of the transformation includes operations and parameters that become unused in the subsequent versions, however, these operations remain in the code resulting an increase of bad smells which hampers maintainability.

We have the following metrics which are not related with bad smells: *matched rules, lazy rules, operations without context,* and *calls to lazy rules per rule*. In the case of matched rules, this should not come as a surprise since transformation languages are built to be used with this declarative construct. In addition, it was interesting to notice that lazy rules have no significant impact on smells either, this could be because lazy rules are being invoked from matched rules and the

Table 1. Influence of language constructs on bad smells

Metric	IRR	Significant Effect
Matched rules	1.0093	No
Lazy rules	1.0130	No
Operations With Context	1.1219	Yes
Operations Without Context	1.0236	No
Ifs	1.0427	Yes
Loops	1.1631	Yes
Variables per rule	1.1989	Yes
Calls to operations per rule	1.0788	Yes
Calls to lazy rules per rule	1.0035	No
Unused operations	1.1527	Yes
Unused parameters	2.4112	Yes

lazy code is not necessarily imperative. The latter is necessary in some contexts, for example, when an input element has to be transformed into several output elements that share no references. Finally, the influence of *OWC* (operations without context) on smell incidence does come as an interesting fact. In principal, the use of context should lead to a proper invocation of the operation to avoid errors. However, some students abuse this kind-of operation to create output elements, instead of using matched/lazy rules.

6 Conclusions and future work

It is important that instructors sensitize students about taking into consideration not only transformations functionality but also the quality of transformations for maintenance and performance whilst they develope model management programs. This work gives empirical evidence that a declarative programming style influences positively the quality because the transformations are less bad-smell-prone. In this context, instructors should present the catalog of bad smells to students and explain them how the imperative style increase the probability of smells occurrences in code. In addition, instructors should consider the quality in the evaluation criteria of practical work to motivate students to avoid bad smells. Finally, the following concrete tips can be given to students: i) Favor lazy rules compared to operations if matched rules do not allow to express certain mappings (e.g., a source element that have to be transformed to different target elements that share no references); ii) Prefer guards and declarative rules instead of *if*, *loops*, and creation of new elements in the *statement block* of rules; and iii) Keep the code updated and avoid dead code (such as unused operations/parameters).

The proposed discoverer is able to found occurrences of 8 types of bad smells. It would be worth to extend our discoverer to cover the remaining 21 bad smells categorized in the state-of-art[13,12,5]. In addition, our discoverer could also be

[7] GITHUB, Bad smell metrics. URL: `https://github.com/phillipus85/ETLMetrics/blob/master/EduSymp2017/resources/Metrics.pdf`. July 13, 2017.

generalized to identify bad smells in transformations written in other languages such as QvT, ATL, ASF+SDF. Finally, since most of transformation developers use specific IDEs, it would be useful to integrate the discoverer to these IDEs in order to provide feedback to the programmers at implementation time. In addition, it would be useful to integrate the previous work made around (semi-)automated refactoring.

References

1. Alkhazi, B., Ruas, T., Kessentini, M., Wimmer, M., Grosky, W.I.: Automated refactoring of atl model transformations: A search-based approach. In: Proceedings of MODELS '16. pp. 295–304. ACM, New York, NY, USA (2016)
2. van Amstel, M.F., van den Brand, M.: Using metrics for assessing the quality of atl model transformations. In: Workshop on Model Transformation with ATL. pp. 19–33. Springer (2010)
3. van Amstel, M.F., Lange, C.F., van den Brand, M.G.: Using metrics for assessing the quality of asf+ sdf model transformations. In: International Conference on Theory and Practice of Model Transformations. pp. 239–248. Springer (2009)
4. Cuadrado, J.S., Guerra, E., de Lara, J.: Static analysis of model transformations. IEEE Transactions on Software Engineering PP(99), 1–1 (2016)
5. Cuadrado, J.S., Jouault, F., Molina, J.G., Bézivin, J.: Optimization patterns for ocl-based model transformations. In: International Conference on Model Driven Engineering Languages and Systems. pp. 273–284. Springer (2008)
6. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Mining correlations of atl model transformation and metamodel metrics. In: Proceedings of the Seventh International Workshop on Modeling in Software Engineering. pp. 54–59. MiSE '15, IEEE Press, Piscataway, NJ, USA (2015)
7. Kapová, L., Goldschmidt, T., Becker, S., Henss, J.: Evaluating Maintainability with Code Metrics for Model-to-Model Transformations, pp. 151–166. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
8. Rentschler, A., Noorshams, Q., Happe, L., Reussner, R.: Interactive Visual Analytics for Efficient Maintenance of Model Transformations, pp. 141–157. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
9. Tichy, M., Krause, C., Liebel, G.: Detecting performance bad smells for henshin model transformations. In: AMT@MoDELS. CEUR Workshop Proceedings, vol. 1077. CEUR-WS.org (2013)
10. Van Amstel, M.F., Van Den Brand, M.G.J.: Model transformation analysis: Staying ahead of the maintenance nightmare. In: Proceedings of the 4th International Conference on Theory and Practice of Model Transformations. pp. 108–122. ICMT'11, Springer-Verlag, Berlin, Heidelberg (2011)
11. Vignaga, A.: Metrics for measuring atl model transformations, tech report (2009)
12. Wei, R., Kolovos, D.S.: Automated analysis, validation and suboptimal code detection in model management programs. In: BigMDE@STAF. CEUR Workshop Proceedings, vol. 1206, pp. 48–57. CEUR-WS.org (2014)
13. Wimmer, M., Jouault, F., Cabot, J.: A catalogue of refactorings for model-to-model transformations. Journal of Object Technology 11(2), 2–1 (2012)