2015 12th Working Conference on Mining Software Repositories

Landfill: an Open Dataset of Code Smells with Public Evaluation

Fabio Palomba*, Dario Di Nucci*, Michele Tufano†
Gabriele Bavota‡, Rocco Oliveto§, Denys Poshyvanyk†, Andrea De Lucia*
*University of Salerno, Fisciano (SA), Italy — †The College of William and Mary, Williamsburg, VA, USA
‡Free University of Bozen-Bolzano, Bolzano, Italy — §University of Molise, Pesche (IS), Italy
fpalomba@unisa.it, ddinucci@unisa.it, mtufano@email.wm.edu
gabriele.bavota@unibz.it, rocco.oliveto@unimol.it, denys@cs.wm.edu, adelucia@unisa.it

***Abstract*—Code smells are symptoms of poor design and implementation choices that may hinder code comprehension and possibly increase change- and fault-proneness of source code. Several techniques have been proposed in the literature for detecting code smells. These techniques are generally evaluated by comparing their accuracy on a set of detected candidate code smells against a manually-produced oracle. Unfortunately, such comprehensive sets of annotated code smells are not available in the literature with only few exceptions. In this paper we contribute (i) a dataset of 243 instances of five types of code smells identified from 20 open source software projects, (ii) a systematic procedure for validating code smell datasets, (iii) LANDFILL, a Web-based platform for sharing code smell datasets, and (iv) a set of APIs for programmatically accessing LANDFILL's contents. Anyone can contribute to Landfill by (i) improving existing datasets (e.g., adding missing instances of code smells, flagging possibly incorrectly classified instances), and (ii) sharing and posting new datasets. Landfill is available at www.sesa.unisa.it/landfill/, while the video demonstrating its features in action is available at http://www.sesa.unisa.it/tools/landfill.jsp.**

I. INTRODUCTION

Code smells (or simply *smells*) have been defined by Fowler as *symptoms of poor design and implementation choices* [4]. These smells violate widely accepted principles of good design, such as pursuing high cohesion and low coupling of software modules. For example, a *Blob* is a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. *Blob* classes can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs, and to add new features. Previous studies indicated that smells hinder comprehension [1], and possibly increase change- and fault-proneness [5], [6]. In summary, smells need to be carefully detected and monitored and, whenever necessary, refactoring actions should be planned for removing them.

For all these reasons, several techniques and tools have been proposed in the literature to detect smells in source code [7], [8], [9], [11]. The detection accuracy of a generic technique T is generally evaluated in terms of precision and recall by comparing the set of candidate smells detected by T against a manually-produced oracle, reporting all instances of smells present in one or more software systems. Unfortunately, to the best of our knowledge, there are very few or no comprehensive datasets of manually validated smells publicly available[1], and those available are often limited to specific smells (*e.g.,* Blob [12]). Also, there is no mechanism for researchers to contribute to such datasets with other instances or different kinds of smells, and to provide further evaluation. As a result, different techniques are evaluated against different oracles, manually defined by different people on different systems. Such a strategy has a number of drawbacks:

- *Oracle subjectiveness.* The manual identification of smells is naturally affected by a very high degree of subjectiveness. Indeed, often people do not agree on the presence of a smell instance [2]. Thus, even oracles built for the same software system and for the same set of smells are very likely to be different if built by different people.
- *Difficulties in comparing different techniques and generalizing the results.* Given the lack of publicly available oracles, competitive techniques are usually compared on a small oracle built in an *ad-hoc* fashion by the authors for the sake of comparison. However, this not only poses questions to the validity of the results, but also to their generalizability.
- *Very high effort required to build the oracle.* Building an oracle reporting instances of smells in a software system implies manually inspecting all the classes of such a system by looking for smell instances. Clearly, this process is time-consuming and error-prone.

In this paper we (i) present LANDFILL, a Web-based platform for sharing code smell datasets and validating them; (ii) contribute to LANDFILL by providing the first dataset consisting of 243 manually validated instances of five code smells, namely *Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*, *Blob*, and *Feature Envy* from 20 open source projects. To the best of our knowledge, this constitutes the largest collection of manually validated smells publicly available as of today. Such a dataset has also been used to validate our work on detection of code smells using historical information [9], [10]; (iii) a methodology for manually creating code smell oracles;

[1]The available datasets are generally replication packages reporting smell instances for few software systems (for instance, see [8]).

* Fabio Palomba is partially funded by the University of Molise.

978-0-7695-5594-2/15 $31.00 © 2015 IEEE
DOI 10.1109/MSR.2015.69

and (iv) a set of APIs for programmatically accessing the platform's contents.

Given LANDFILL and the methodology we propose, anyone can contribute with the aim of increasing the quality and quantity of the datasets by improving existing datasets (*e.g.,* adding missing instances of code smells, marking potentially incorrect instances, etc), and sharing new datasets.

Possible applications. There are several use cases that the research community can take advantage of while using LANDFILL. The first, and most obvious, concerns the evaluation and comparison of code smell detection tools. Such comparisons would be cheaper to perform, and based on the oracles that can be verified by several people, thus reducing the degree of subjectiveness. The LANDFILL's datasets can also be used in empirical studies on smells. Indeed, often such studies are performed on smell instances automatically identified by tools [5], [6], thus possibly introducing errors in the data due to the tool's imprecision (*e.g.,* detecting a smelly instance which may not be affected by any code smell). The availability of the LANDFILL's datasets can move such studies to the new level, where the code smell data are manually built and verified by multiple annotators. Last, but not least, the platform can be used in academic context for teaching refactorings that usually start with identifying code smells.

Structure of the paper. Section II reports the process we adopted to build the first dataset currently available in LANDFILL, while Section III describes the support provided by the LANDFILL web-platform to increase and improve the available datasets. Finally, Section V concludes the paper.

II. BUILDING THE CODE SMELL DATASET

This section describes the process we adopted to contribute the first smell dataset in LANDFILL. Our dataset includes instances of the following five code smells:

- *Divergent Change*: this smell occurs when a class is changed in different ways for different reasons. The example reported by Fowler in his book on refactoring [4] helps understand this smell: *If you look at a class and say, "Well, I will have to change these three methods every time I get a new database; I have to change these four methods every time there is a new financial instrument", you likely have a situation in which two classes are better than one* [4]. Thus, this kind of smell clearly triggers *Extract Class* refactoring opportunities.
- *Shotgun Surgery*: a class is affected by this smell when a change to this class (*i.e.,* to one of its fields/methods) triggers many little changes to several other classes [4]. The presence of a Shotgun Surgery smell can be removed through a *Move Method/Field* refactoring.
- *Parallel Inheritance*: this smell occurs when *"every time you make a subclass of one class, you also have to make a subclass of another"* [4]. This could be a symptom of design problems in the class hierarchy that can be solved by redistributing responsibilities among the classes through different refactoring operations, *e.g., Extract Subclass*.
- *Blob*: a class implementing several responsibilities, having a large number of attributes, operations, and dependencies with data classes [3]. The obvious way to remove this smell is to use *Extract Class* refactoring.
- *Feature Envy*: as defined by Fowler [4], this smell occurs when *"a method is more interested in another class than the one it is actually in"*. For instance, there can be a method that frequently invokes accessor methods of another class to use its data. This smell can be removed via *Move Method* refactoring operations.

A. Manual Validation Methodology

We looked for instances of these five smells in the 20 open source projects listed in Table I. Besides listing the projects' name, Table I also reports for each of them some size attributes (*i.e.,* number of classes and KLOC), the number of identified instances for each smell, and the git snapshot-id of the system version on which the oracle has been built. Concerning the latter, we split the history of the twenty projects in two equal parts, and build the oracle on a snapshot falling in the middle. For instance, given the history of *Apache Ant* going from January 2000 to January 2013, we selected a system snapshot s from June 2006. This was done since some of the existing code smells detection approaches exploit historical information (see [9]). Thus, it is important for such techniques to have sufficient history (the first half of each system in our case) from which to extract the needed information.

For each of the twenty selected snapshots, one of the authors manually identified instances of five considered smells. Starting from the definition of the five smells reported in literature [3], [4], the author manually analyzed the source code of each snapshot, looking for instances of those smells. Clearly, for smells characterized by how the code components evolve over time (*e.g., Parallel Inheritance*), he also analyzed the changes performed by developers on different code components. This process took approximately four weeks of work. Then, another author validated the produced oracle, to verify that all affected code components identified by the first author were correct. Only six of the smells identified by the first author were classified as false positives by the second author. An open discussion about these six instances has been performed between the two authors, by looking again together into the smell definitions, the source code, and the change-history information at hand. After this discussion, two of the six smells were classified as false positives (and thus removed from the oracle).

Note that, while additional verification performed by the second author does not ensure that the defined oracle is complete (*i.e.,* it includes all affected components), it increases our degree of confidence in the correctness of the identified smell instances and reduces the subjectiveness of our dataset.

B. Threats and Possible Imprecisions

As every manually built dataset, the oracles we produced and included in LANDFILL may contain errors and omissions, mainly due to the subjectiveness of discriminating smelly and

TABLE I
SNAPSHOTS CONSIDERED FOR THE SMELL DETECTION.

Project	git snapshot	Classes	KLOC	Divergent Change	Shotgun Surgery	Parallel Inheritance	Blob	Feature Envy
Apache Ant	da641025	846	173	0	0	7	8	8
Apache Tomcat	398ca7ee	1,284	336	5	1	9	5	3
jEdit	feb608el	316	101	4	1	3	5	10
Android API (framework-opt-telephony)	b3a03455	223	75	0	0	0	13	0
Android API (frameworks-base)	b4ff35df	2,766	770	3	1	3	18	17
Android API (frameworks-support)	0f6f72e1	246	59	1	1	0	5	0
Android API (sdk)	6feca9ac	268	54	1	0	9	10	3
Android API (tool-base)	cfebaa9b	532	119	0	0	0	0	0
Apache Commons Lang	4af8bf41	233	76	1	0	6	3	1
Apache Cassandra	4f9e551	826	117	3	0	3	2	28
Apache Commons Codec	c6c8ae7a	103	23	0	0	0	1	0
Apache Derby	562a9252	1,746	166	0	0	0	9	0
Eclipse Core	0eb04df7	1,190	162	1	1	8	4	3
Apache James Mime4j	f4ad2176	250	280	1	0	0	0	9
Google Guava	e8959ed0	153	16	0	0	0	1	2
Aardvark	ff98d508	103	25	0	1	0	1	0
And Engine	f25236e4	596	20	0	0	0	0	1
Apache Commons IO	c8cb451c	108	27	1	0	1	2	1
Apache Commons Logging	d821ed3e	61	23	2	0	2	2	0
Mongo DB	b67c0c43	22	25	1	0	0	3	0

Blob	org.data.User	
Blob	org.action.Send	
Parallel Inheritance	org.data.User	org.data.Student
Parallel Inheritance	org.ui.UserForm	org.ui.StudentForm
Divergent Change	org.action.Manager	

Fig. 1. Example of `csv` file accepted by LANDFILL to import a dataset.

non-smelly instances. We alleviated this threat by involving two people in building the oracle, thus reducing the level of subjectiveness. However, we cannot exclude that such manual analysis could have potentially missed some smells, or else identified some false positives. We also rely on the contributions from the research community to help us improving the quality of the dataset via the LANDFILL platform.

III. EVOLVING THE DATASET

This section describes how the LANDFILL platform can be used to create, share, and improve code smell datasets.

A. Creating a New Dataset

LANDFILL allows anyone to create a code smell dataset. After registering, a user can upload a dataset to LANDFILL following two simple steps:

1) *Create the dataset.* The user provides (i) the name of the dataset, (ii) the list of authors, (iii) a description of the process adopted to create it. Note that the process adopted by the authors could be totally different from the one we adopted in the building of the first dataset we contributed to LANDFILL. The goal of this description is to provide the users of the dataset with clear information on how the dataset has been built, catching possible pitfalls in the adopted process.
2) *Dataset upload.* After having created the dataset instance, the user can add new systems to it by providing (i) the name of the system, (ii) the snapshot for which the user wants to insert smell instances, (iii) an archive file containing the source code of the smells of the system and (iv) an `csv` file having a specific format in which she can specify the kind of code smells considered in the dataset (*e.g., Blob* and *Feature Envy*). In particular:
   - each line of the `csv` file represents a smell instance;
   - the first column of each line specifies the smell kind the instance refers to (*e.g., Blob*);
   - columns from $\#2$ to $\#n$ in each line represent the $n-1$ code components involved in that specific instance of code smell (*e.g.,* a single class in column $\#2$ following the "*Blob*" string in column $\#1$ indicates the class affected by the *Blob* smell).

   An example of `csv` file accepted by LANDFILL is shown in Fig. 1. Lines reporting instances of *Blob* and *Divergent Change* smells just list the class affected by these smells (*e.g.,*`org.data.User` is a *Blob* class). Instead, since the *Parallel Inheritance* smell affects pairs of classes (see the definition in Section II), each of its instances reports the affected pair of classes. Note that, depending on the smell kind, an affected code component can also be something different from a class (*e.g.,* in case of *Feature Envy* the affected code component is a method).

B. Improving Existing Datasets

Any registered user can contribute to LANDFILL by improving the quality of existing datasets. Fig. 2 depicts the page used by LANDFILL to visualize an existing dataset. Given a dataset reporting the instances of a smell type T on a software system S, a registered user can:

1) *Add a missing instance of T in S.* This can easily be done by pushing the button on top of the screen (see Fig. 2) and inserting the complete identifier of the smelly code component (*e.g.,* org.Landfill.AddMissingInstance).
2) *Vote on the smell instances reported in the dataset.* This is possible thanks to the like/dislike buttons appearing when moving the mouse over a smell instance (see left side of Fig. 2). On the one side, positive scores are useful to easily identify the smell instances for which there is a high agreement among the LANDFILL's users. On the other side, negative scores are used to identify "likely

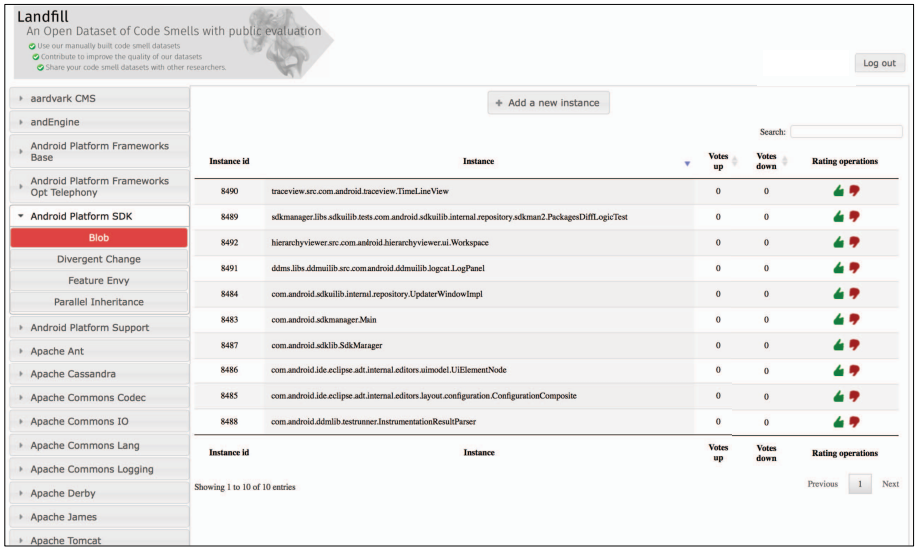

Fig. 2. LANDFILL: Visualizing an existing dataset.

wrong" instances. To facilitate the analysis of classes and identifying the possible presence of smells, a user can explore class's source code by clicking on its name. Users can also leave comments to the dataset, for example to explain the rationale behind a positive/negative evaluation they gave to a smell instance. LANDFILL visualizes the percentage of positive ratings obtained by each dataset as one of its properties. In this way, it is easy for researchers interested in reusing code smells datasets to quickly identify the most "reliable" ones.

IV. PROGRAMMATIC ACCESS TO THE PLATFORM

LANDFILL provides a set of APIs that support the programmatic access to its datasets. In particular, it is possible to (i) retrieve the list of all datasets available in the platform, obtaining for each of them the list of authors, the name of the dataset, the considered smells, and the percentage of positive ratings it has; (ii) get the list of code smell instances in a specific system (belonging to a dataset and a specific type,*e.g.*, Blob classes); (iii) download the source code of the classes of interest; (iv) send positive/negative ratings for specific smell instances. The complete APIs specification is available at http://www.sesa.unisa.it/tools/landfill.jsp.

V. CONCLUSION

We presented LANDFILL, a web-based platform aimed at promoting the collection and sharing of code smell datasets in the software engineering and MSR research community in particular. We contributed to LANDFILL with the first dataset containing 243 instances of five smell types from 20 open source projects. Such a dataset has been already used to validate our history-based smell detection approach [9].

LANDFILL is already available to everyone willing to contribute to it. In the future, we are planning on extending our platform to support other types of datasets, and also to host the results of experiments performed using LANDFILL's datasets.

REFERENCES

[1] 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 *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*. IEEE Computer Society, 2011, pp. 181–190.

[2] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *Software Engineering, IEEE Transactions on*, vol. 40, no. 7, pp. 671–694, July 2014.

[3] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. John Wiley and Sons, 1998.

[4] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[5] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*. IEEE Computer Society, 2009, pp. 75–84.

[6] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[7] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*. IEEE Computer Society, 2004, pp. 350–359.

[8] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "DECOR: a method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[9] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering (to appear)*.

[10] ——, "Detecting bad smells in source code using change history information," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 268–278.

[11] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[12] S. Vaucher, F. Khomh, N. Moha, and Y. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes," in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*. IEEE Computer Society, 2009, pp. 145–154.