Towards Just-In-Time Refactoring Recommenders

Jevgenija Pantiuchina
Università della Svizzera italiana (USI)
Lugano, Switzerland
jevgenija.pantiuchina@usi.ch

Gabriele Bavota
Università della Svizzera italiana (USI)
Lugano, Switzerland
gabriele.bavota@usi.ch

Michele Tufano
College of William and Mary
Williamsburg, Virginia, USA
mtufano@cs.wm.edu

Denys Poshyvanyk
College of William and Mary
Williamsburg, Virginia, USA
denys@cs.wm.edu

ABSTRACT

Empirical studies have provided ample evidence that low code quality is generally associated with lower maintainability. For this reason, tools have been developed to automatically detect design flaws (*e.g.,* code smells). However, these tools are not able to *prevent* the introduction of design flaws. This means that the code has to experience a quality decay (with a consequent increase of maintenance/evolution costs) before state-of-the-art tools can be applied to identify and refactor the design flaws.

Our goal is to develop a new generation of refactoring recommenders aimed at preventing, via refactoring operations, the introduction of design flaws rather than fixing them once they already affect the system. We refer to such a novel perspective on software refactoring as just-in-time refactoring. In this paper, we make a first step towards this direction, presenting an approach able to predict which classes will be affected in the future by code smells.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

KEYWORDS

refactoring, code smells

ACM Reference Format:
Jevgenija Pantiuchina, Gabriele Bavota, Michele Tufano, and Denys Poshyvanyk. 2018. Towards Just-In-Time Refactoring Recommenders. In *ICPC'18: 26th IEEE/ACM International Conference on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196321.3196365

1 INTRODUCTION

In the software life-cycle, change is the rule rather than the exception. A key point for sustainable program evolution is to tackle software complexity, making sure that the source code exhibits a high quality thus easing maintenance activities. Indeed, several studies provide evidence that low code quality is associated with lower productivity and more effort for developers [4, 7].

While there is an agreement on the need for software quality, modern IDEs (Integrated Development Environments) only provide basic support for developers to fight low code quality. For this reason, tools have been developed in industry and academia to detect design flaws and, in some cases, to remove them via refactoring (see [8]). For example, code smell detectors are able to identify specific types of design flaws hindering code maintainability. While extensive empirical studies showed the high accuracy of these detectors, they all suffer from a common limitation: they are not able to *prevent* the introduction of design flaws, but only able to identify flaws once they already affect the system. This means that the code has to experience a quality decay (with a consequent increase of maintenance/evolution costs) before these tools can be applied to identify and refactor those design flaws.

Our long term vision is to develop a new generation of refactoring recommendation systems able to *prevent* the introduction of design flaws. We refer to such a novel perspective on software refactoring as **just-in-time refactoring**. In this paper, we make a first step in that direction by presenting COSP (**CO**de **S**mell **P**redictor), an approach to predict classes that are likely to be affected by code smells in the near future. COSP can be used to prioritize code components for refactoring; similarly to how bug prediction techniques can be used to focus testing activities on code components that are more likely to be buggy, COSP can point developers to locations of code that warrant refactoring. COSP takes a change history of a system with its latest version as an input and classifies all its classes as becoming smelly or not in the near future, also indicating types of code smells that will likely affect the classes.

We present a preliminary evaluation of COSP on six systems. Our results show that predicting classes that will be affected by code smells is far from trivial. This is due to the fact that out of hundreds of classes in a system, only one or two will become smelly in the near future, and the real challenge is to identify them without flooding developers with a high number of false positives. We deal with this problem by embedding a confidence level in COSP that provides indications about the probability that a class would actually become smelly in the future. In this way, the developer can decide to only receive a warning when COSP has a high confidence in its recommendation. With the highest confidence, COSP identifies classes that will become smelly with a precision ~75%.

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196365

2 CODE SMELL PREDICTOR

COSP predicts whether a given class will be affected by a specific type of code smell within t days. The type of code smell to predict as well as the threshold t can be customized. In our work, we use COSP to predict *God* and *Complex Classes*. A *God Class* has been defined as a procedural-style design class with many different responsibilities, which monopolizes the application's logic of the system. A *Complex Class* is characterized by extremely high complexity of its code, generally assessed using the cyclomatic complexity [5].

COSP exploits a machine learning approach, *i.e.,* the Weka implementation of Random Forest [1], to identify a series of rules which would discriminate classes likely to be affected by a specific smell type (ST) within t days. The Random Forest builds a collection of decision trees with the aim of solving classification-type problems, where the goal is to predict values of a categorical dependent variable from one or more continuous and/or categorical predictor variables. When used to classify previously unseen instances, the Random Forest provides as output the probability that an instance belongs to each of the possible categories of the dependent variable.

Table 1: Class quality metrics used by COSP

Metric	Description
CBO	Coupling Between Object classes: measures the dependencies a class has [2]
DIT	Depth of Inheritance Tree: the length of the path from a class to its farthest ancestor in the inheritance tree [2]
NOC	Number Of Children (direct subclasses) of a class
NOF	Number Of Fields declared in a class
NOPF	Number Of Public Fields declared in a class
NOSF	Number Of Static Fields declared in a class
NOM	Number Of Methods in a class
NOPM	Number Of Public Methods in a class
NOSM	Number Of Static Methods in a class
NOSI	Number Of Static Invocations of a class
WMC	Weighted Methods per Class: Sums the cyclom. comp. of methods in a class [2]
RFC	Response For a Class: The number of methods in a class plus the number of remote methods that are called recursively through the entire call tree [2]
ELOC	Effective Lines Of Code: The lines of code excluding blank lines and comments
LCOM	Lack of Cohesion Of Methods: A class cohesion metric based on the sharing of local instance variables by the methods of the class [2]

2.1 Predictor Variables

We use the metrics described in Table 1 as predictor variables.

Given d_p the date in which the developer wants to use COSP to predict which classes in her system are likely to be affected by ST within t days, we use the 14 metrics to characterize the quality of each class C from three different perspectives:

Current code quality. The value of the 14 metrics measured on C at the prediction date d_p.

Historical code quality trend. As observed by Tufano *et al.* [9], smelly classes exhibit metric trends over time that are substantially different as compared to clean classes. For example, the cohesion of classes that will become God Classes generally decreases over 100 times faster than the cohesion of classes that will preserve their high quality [9]. For this reason, we compute the value of the 14 quality metrics on C in all commits performed until d_p in which C has been added or modified. Then, for each metric M, we compute the regression slope line fitting the M's values for C across the identified commits. The slope of a line describes its steepness and, in our case, can highlight, for example, a continuing degradation of some quality aspects (*e.g.,* a high positive slope for the WMC metrics indicates a strong increase in complexity for C over time).

Recent code quality trend. For each metric M we also compute for C its recent code quality trend, meaning the slope of the regression line fitting the M's values for C when only focusing on the last l changes that C has been subject to. We consider $l = 10$ (*i.e.,* at most the last ten changes impacting C). If C has been subject to p changes before d_p and $p < l$, then we set $l = p$ when computing the recent slopes for C. The idea behind using the recent slopes is that a class could have had a very long change history, mostly characterized by high quality code, but with recent worrying trends of some quality metrics. The 14 recent slopes lead to the total 42 features (14+14+14) used by the Random Forest algorithm to identify classes that would potentially become smelly.

2.2 Training the Classifier

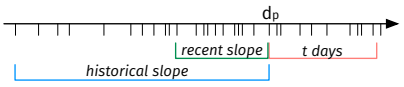

Figure 1: Building the training set with historical data

Fig. 1 depicts how data from a versioning system can be used to build the COSP's training set. In particular, Fig. 1 represents changes (commits) performed over the history of a software system and impacting (*i.e.,* adding or modifying) a specific class C, that will be used as a data entry for the training set. We start by checking out the system's snapshot at date d_p, making sure that this date is at least t days foregoing the last commit available in the repository. In fact, while the past history of C (*i.e.,* the time period preceding d_p) is needed to compute the 42 predictor variables, knowing its future (*i.e.,* the time period following d_p) is needed to verify whether the C has been affected by ST within t days. For each class C, which is used to build the COSP's training set, we need to provide:

(1) The values of 42 predictor variables. In the example depicted in Fig. 1 this would include: (i) the value for the 14 quality metrics measured for C on d_p; (ii) the historical slope for each metric computed by considering all the commits impacting C and performed before d_p (blue time period); and (iii) the recent slopes obtained when only considering the last l commits impacting C (green).

(2) Its classification (affected or not by ST within t days). The training set must report whether C will be affected by ST within t days, meaning between d_p and $d_0 + t$. To classify a class as affected or not by ST, it is possible to use one of the smell detectors proposed in the literature (*e.g.,* [6]), or manually annotated datasets.

Note that, given the versioning system of software project, the training set can be built by computing the above information for all its classes and across several different commits (*i.e.,* different d_p). The only constraints to keep in mind are: (i) if the class C is already affected by ST at date d_p, C cannot be included in the training set, since COSP should learn to predict classes that will be affected by ST in the future (and not the classes that already represent a maintainability problem); (ii) for the reason previously explained, d_p date must be at least t days foregoing the last commit available in the repository; (iii) at least two commits must have impacted the C class before d_p, otherwise it is not possible to compute the predictor variables exploiting the regression lines slopes.

Also, for each smell type ST one is interested in predicting, a different training set is needed since the categorical dependent variable changes. While it should be possible to build a classifier

considering different types of code smells at once (*e.g.,* predicting if within the next t days the C class will (i) remain clean, (ii) be affected by *God Class*, or (iii) be affected by *Complex Class*), we preferred to start tackling the problem in its simplest formulation (*i.e.,* smell-specialized models, predicting if within the next t days the C class will be affected by a specific type of smell ST or not).

Once the training set is built, it can be used to train the classifier. Since some of the features we considered are likely to correlate (*e.g.,* recent and historical slopes for the same metric), we perform a *correlation-based* feature selection process [3] implemented in the `CfsSubsetEval` Weka class. Also, when training the model we check the distribution of training set samples across the two categories (*i.e.,* will become smelly or not). Since the number of classes that will become smelly in the near future is much smaller than those that remain clean [9], we apply a spread sub-sample re-balancing technique, meaning that given x, the number of classes that will become smelly present in the training set, we randomly select x classes that will remain clean to balance the training set.

The output of the training stage is a Random Forest classifier, represented by a collection of decision trees composed by yes/no questions that split the training sample into gradually smaller partitions that group together cohesive sets of data (*i.e.,* those having the same value for the dependent variable).

2.3 Using COSP to Predict Code Smells

Once the classifier is built, `COSP` can automatically predict in a given set of classes (*e.g.,* an entire system) those classes that are likely of being affected by ST within the next t days. To do that, `COSP` starts by computing the value of the 42 predictor variables for each given class. Then, the Random Forest classifier is used to categorize each class. Important to highlight here is that the Random Forest not only assigns each class to one of the two possible categories (*i.e.,* 0: affected or 1: not by ST within t days), but it also provides the probability that the class belongs to the predicted category. For example, it can predict that the class will become smelly with a probability of 0.9. We exploit this indication as a *confidence level* for `COSP`. In particular, given a scenario in which the developer uses `COSP` to prioritize classes for refactoring, she can start by only inspecting the ones predicted by `COSP` as becoming smelly with a confidence level (probability) of 1.0. The same applies in a scenario in which a project manager wants to be alerted when the quality of some classes is likely to become noticeably worse in the near future, without receiving too many false alarms, however.

3 PRELIMINARY EVALUATION

Table 2: The six systems used in our evaluation

System	KLOC	#Commits	#Testing Commits	Become Complex Class	Become God Class
bazel	46	14,660	2,596	1,030	942
druid	34	5,702	827	105	149
fast json	50	2,684	493	3	22
guava	13	4,650	1,079	115	415
picasso	13	1,002	107	14	23
realm	128	7,868	395	145	230
overall	**284**	**36,566**	**5,497**	**1,412**	**1,781**

3.1 Study Design

The *goal* is to assess the performance of `COSP` in identifying classes that could become smelly in the near future. We instantiate `COSP` to predict two types of smells (*Complex Class* and *God Class*) and we set $t = 90$. Future work will investigate different values of t.

The *context* is represented by the history of the six systems listed in Table 2. These systems are all hosted on GitHub and have been randomly selected from the list of most popular repositories. The only filtering criterion we used was to exclude repositories having less than one thousand commits in their change history.

Assuming D to be the length of the change history of a system S, we split it into three parts. Commits falling in the first 25% of D were skipped and used as the history needed to compute the predictor variables based on slopes. Commits from the 25% to the 75% of D are used to build the training set for S. This means that for each commit c falling in this time period we:

(1) Checkout c and compute the 14 quality metrics for all classes.

(2) Analyze the history preceding c to compute the 28 slope-based variables for each class. Having skipped the first 25% of D ensures that we have enough change history for most of classes.

(3) Check whether each class in c will become a *Complex Class* (or a *God Class*) within t days from c. To do that, we run an implementation of the *DECOR* smell detector based on the original rules defined by Moha *et al.* [6]. The choice of using *DECOR* is driven by the fact that it is a state-of-the-art smell detector having a high accuracy in detecting smells [6].

Note that if a class is already affected by the considered code smell type in c, it is not included in the training set.

Finally, the remaining 25% of commits (excluding those performed in the last t days) are used for testing. In particular, we simulate the scenario in which the developer runs `COSP` on a given system snapshot (commit) and wants to predict which classes will become smelly within t days. Note that with this experimental design we are training a different model for each system by using part of its history for training and part for testing. Future work will be devoted to experiment with cross-project prediction.

Table 2 shows for each system the number of commits used for testing and the total number of classes in the testing commits that will become smelly after $t = 90$ days. For example, let us assume to only have one system with two testing commits c_1 and c_2 and that, of all classes in c_1, two become smelly after t days while of all those in c_2 none becomes smelly after t days. The total number of classes that become smelly in the testing commits will be two.

We report precision and recall achieved by our approach both overall and when considering classifications having five different confidence levels going from 1 to 0.6 at steps of 0.1. We do not consider classifications with a confidence lower than 0.6 since a probability close to 0.5, with only two possible categories is basically random guessing. Also, we are particularly interested in the precision and recall when predicting a class as becoming smelly, since these would be the alerts given to the developers to prioritize refactoring operations. We compare the performance of our approach with two baselines built on top of the ELOC (size) and the WMC (complexity) quality metrics. Given the set of n classes identified as becoming smelly in the future by `COSP` when run on a given commit c, we descendingly sort the classes in c based on their ELOC (first baseline) and WMC (second). Then, we select the top n classes in the two ranked lists and we assume that those, being the largest and most complex in the system, are the ones going to become smelly in the future. In other words, we compare the

recall and precision of COSP with that of the two baselines when providing the developer with a list of classes of the same size n candidate to become smelly (thus, the same effort is required by the developer for their analysis). Complete data about our study is available at https://github.com/ICPC2018/COSP.

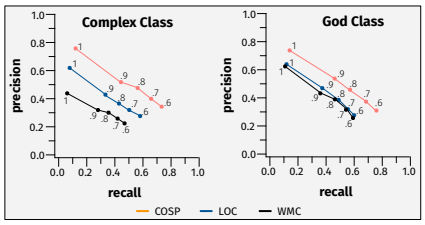

Figure 2: Results achieved with different confidence levels.

3.2 Discussion of the Results

We present the precision and recall of COSP when categorizing classes as becoming smelly (from now on, *BS*) or not (*NS*). Performance is good when considering the *NS* class (precision ranging across systems between .98 and 1.00, recall between .84 and .97), the accuracy of COSP strongly drops in the *BS* class, where it achieves a good recall (from .74 to 1.00) but with a lower precision (from .08 to .48). As previously said, the precision in the *BS* class is what we are really interested in, since we envision COSP as a way to point out to developer where to focus her refactoring attentions.

For this reason, we now focus our attention on the *BS* to see what happens when COSP reports classes as becoming smelly. The red line in Fig. 2 shows the precision and recall achieved by COSP in the *BS* class when only considering classifications having a confidence level higher or equal than λ, with λ going from 1.0 to 0.6 at steps of 0.1. The results in Fig. 2 summarize the overall performance of COSP when considering the six systems as a single dataset. As previously explained, we also compare the results achieved by COSP with two baselines based on the LOC (blue line) and the WMC (black) metrics. Fig. 2 shows that: (i) COSP substantially outperforms the baselines; (ii) when the confidence level decreases, the COSP precision strongly goes downhill, making it less useful in a real scenario; (iii) with the maximum confidence level (1.0), the precision is ∼.75, highlighting as three out of four classes indicated as becoming smelly, will actually become smelly within t days. However, there is a high price to pay in terms of recall, not going over ∼.13. Table 3 reports the results achieved on each system with the maximum confidence level (no classes were classified as becoming *God Class* in fast json).

Table 3: Results when confidence equals 1.0

System	COSP		ELOC baseline		WMC baseline	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
Complex Class						
bazel	0.78	0.09	0.68	0.08	0.46	0.06
druid	0.59	0.34	0.29	0.16	0.26	0.15
fast json	1.00	0.33	0.00	0.00	1.00	1.33
guava	0.75	0.10	0.81	0.11	0.56	0.08
picasso	1.00	0.07	1.00	0.07	1.00	0.07
realm	1.00	0.19	1.00	0.19	0.64	0.12
overall	**0.76**	**0.12**	**0.62**	**0.10**	**0.44**	**0.07**
God Class						
bazel	0.65	0.06	0.62	0.06	0.45	0.04
druid	0.70	0.29	0.56	0.29	0.66	0.27
fast json	-	0.00	-	0.00	-	0.00
guava	0.70	0.15	0.63	0.14	0.59	0.13
picasso	0.61	0.61	0.61	0.61	0.61	0.61
realm	0.97	0.30	0.90	0.28	0.92	0.27
overall	**0.75**	**0.13**	**0.67**	**0.12**	**0.63**	**0.11**

COSP only represents the very first solution paving the way to more research in the field of predicting classes that, in future, will hinder code maintainability. Indeed, the achieved results clearly show that, while the COSP's precision is acceptable in the highest confidence scenario, much more research is needed to create techniques able to match this level of precision with a good recall.

3.3 Threats to Validity

Constructs validity. We relied on *DECOR* rules to detect smells. We are aware that our results can be affected by the presence of false positives/negatives. We never mixed training and testing, and we only balanced the training set, without modifying the testing data.

External validity. We only experimented with COSP on six systems, however, this still accounted for 5,497 testing commits. Also, cross-project prediction was not considered.

4 RELATED WORK

The most relevant works are those proposing techniques to verifying whether the current version of a code component is affected by code smells. We refer the interested reader to the recent survey by Sharma and Spinellis *et al.* [8] overviewing these techniques. COSP clearly tackles a related, but different, problem, being complementary to those techniques. Indeed, a developer could use a code smell detector to identify refactoring opportunities and remove the code smells currently affecting the system. Then, she could start using COSP to *prevent* the future introduction of new smell instances.

5 CONCLUSION

We presented COSP, the first approach available to predict classes that will become smelly in future, thus providing the developers with information useful to prevent their introduction. While COSP represents a tangible step towards just-in-time refactoring recommenders, much more work is needed in experimental domain as well as efforts to overcome its limitations.

ACKNOWLEDGMENT

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the JITRA projects (No. 172479), and of the Swiss Group for Software Engineering (CHOOSE).

REFERENCES

[1] L. Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32.
[2] Shyam R. Chidamber and Chris F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng* (1994), 476–493.
[3] M. Hall. 1998. *Correlation-based feature selection for machine learning.* Technical Report. University of Waikato.
[4] F. Khomh, M. Di Penta, Y. Guéhéneuc, and G. Antoniol. 2012. An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-proneness. *Empirical Softw. Engg.* 17, 3 (June 2012), 243–275.
[5] T. McCabe. 1976. A complexity measure. *IEEE Trans. on Softw. Eng.* 4 (1976), 308–320.
[6] N. Moha, Y. Gueheneuc, L. Duchien, and A. Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Trans. Softw. Eng.* 36, 1 (Jan. 2010), 20–36.
[7] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia. 2017. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* (Aug 2017).
[8] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (2018), 158 – 173.
[9] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. 2015. When and Why Your Code Starts to Smell Bad. In *ICSE '15 (ICSE '15).* 403–414.