Code Bad Smell Detection through Evolutionary Data Mining

Shizhe Fu, Beijun Shen
School of Software
Shanghai Jiao Tong University
Shanghai, China
bjshen@sjtu.edu.cn

***Abstract*—The existence of code bad smell has a severe impact on the software quality. Numerous researches show that ignoring code bad smells can lead to failure of a software system. Thus, the detection of bad smells has drawn the attention of many researchers and practitioners. Quite a few approaches have been proposed to detect code bad smells. Most approaches are solely based on structural information extracted from source code. However, we have observed that some code bad smells have the evolutionary property, and thus propose a novel approach to detect three code bad smells by mining software evolutionary data: duplicated code, shotgun surgery, and divergent change. It exploits association rules mined from change history of software systems, upon which we define heuristic algorithms to detect the three bad smells. The experimental results on five open source projects demonstrate that the proposed approach achieves higher precision, recall and F-measure.**

Keywords—bad smell detection; data mining; software evolutionary history

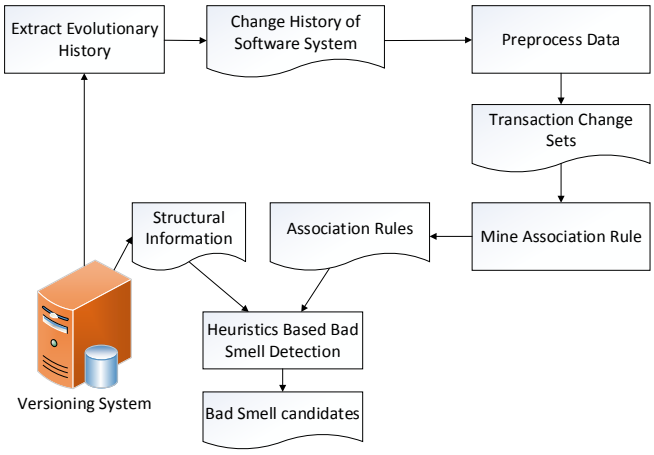
I. INTRODUCTION

As software systems evolve, they may become increasing complex, and the complexity will be accumulated from various sources, such as violation of coding principle and the pressure to meet deadline. In order to conquer these problems, Fowler [1] introduced "bad smell" to denote symptoms of poor design and implementation choices. When a software system is faced with severe complexity and bad quality, refactoring actions should be taken to improve the quality without altering the external behavior. Among all the steps of refactoring, bad smell detection is the very first one. So it is highly crucial to detect bad smells precisely and efficiently.

Quite a few approaches have been put forward to find bad smells automatically. Most of them are based on code structural information extracted from a snapshot of software systems. For example, Marinescu [2] proposed "detection strategies" to detect bad smell via identifying symptoms characterizing bad smells and metrics to measure such symptoms. The metrics are extracted from structural information of the software system, such as WMC (Weight Methods per Class), NOM (Number of Methods), CBO (Coupling Between Objects). Then, detection rules are defined using thresholds on such metrics.

We argue that despite code structural information, the evolutionary data is also a highly important source of bad smell symptoms. On the one hand, some bad smells' definition contain characteristic of evolutionary histories. For instance, in order to detect parallel inheritance (every time you make a subclass of one class, you also have to make a subclass of another), we will look through the evolutionary history to find the candidate class, where the addition of a subclass for it often triggers the addition of a subclass for another class [1, 3]. In this way, we can detect parallel inheritance without knowing the specific structural information of the code. On the other hand, the bad smells detected on evolutionary data are usually most long-standing in the software system.

Fig. 1. Our approach



In this paper, we propose a novel approach to detect bad smells through evolutionary data mining, as shown in Figure 1. It consists of four steps. 1) First, we extract change history of the target software system from revision control system. 2) A series of data preprocessing is performed to get transaction change sets. 3) We perform association rule mining on the change sets using frequent pattern algorithms such as Apriori or FP-growth. The output is a set of rules indicating couplings between different code entities. 4)

978-1-4673-7899-4/15/$31.00 ©2015 IEEE

Heuristic algorithms are defined according to bad smells definition in literatures. They combine association rules and structural information to find bad smell candidate automatically.

This paper makes the following major contributions: It is the first approach that exploits evolutionary data mining to detect code bad smells with code structural information in aid. We perform comprehensive evaluations on five open source projects. The results show that our approach can detect duplicated code, shotgun surgery and divergent change with higher precision, recall, and F-measure. And it proves that evolutionary data is useful in the detection of code bad smells.

The rest of paper is organized as follows. Section II gives an overview of the related work. Section III describes association rule mining of our approach. Section IV presents heuristics based smell detection algorithms using association rules. Section V reports experiments on our approach. Section VI concludes the paper.

II. Related Work

All the researches on bad smell detection originate from the description of bad smells [1-3]. Fowler defined 22 bad smells and also provided refactoring operations to remove them. Webster presented pitfalls in object-oriented development, and provided over eighty helpful summaries on how to avoid potential problems. Riel defined more than 60 guidelines to rate the integrity of a software design. Brown described 40 anti-patterns for which he also provided heuristics to detect them.

Some approaches have been proposed to detect bad smells in software systems. Travassos [4] proposed manual inspections and reading techniques to detect bad smells. Simon [5] introduced a metric-based visualization tool to discover bad smells, using distance based cohesion measure. Emden [6] developed jCOSMO, a code smell browser, to detect and visualize smells in Java source code. Marinescu [2] proposed a metric-based approach to detect bad smells, where he uses detection strategies to formulate metric-based rules that capture deviations from good design. Munro [7] designed templates to describe bad smells systematically. Based on these templates, he proposed metric-based heuristics to detect bad smells. He also performed an empirical study to justify the choice of metrics and thresholds for detecting smells.

In addition to metric-based approaches, there are some other peculiar approaches to detect bad smells. Khomh [8] proposed a novel approach to detect bad smells using Bayesian Belief Network. Instead of just telling user whether or not a code component is affected by a bad smell, the approach calculates the impact probability. Moha [9,10] proposed Décor, which uses a Domain-Specific Language (DSL) for specifying smells in high abstraction level, where software engineers can manually define the specification for bad smells detection using the taxonomy and vocabulary, and, if needed, the context of the analyzed systems.

Specifically, for the detection of duplicated code, most proposed techniques focus on the detection of syntactic or structural similarity of source code. Kamiya [11] introduced CCFinder, which divides a program into lexemes and compare the token sequences to find matches between two subsequences. For shotgun surgery and divergent change, Rao [12] presented an approach based on Design Change Propagation Probability (DCPP) matrix. The DCPP is an n*n matrix where A_{ij} represents the probability that a design change on the artifact i requires a change also to the artifact j. A row of the DCPP matrix that contains high values for an artifact means that there is high probability that a change have impact on more than one artifact, where an instance of shotgun surgery is detected. If a column in the matrix contains high values for a particular artifact, an instance of divergent change is detected.

In summary, most of the approaches available in the literature are solely based on structural information extracted from a single snapshot of software systems. As far as we know, there has not much research on the use of evolutionary data for code smells detection.

III. Association Rule Mining

Association rule mining is a popular and well researched method for discovering interesting relations between variables in large databases [13]. In this paper, what we are mostly interested in this part is to find out when a particular source code entity (class, function or field) is changed (modified, added or deleted), what other entities are also changed. This relation can reflect couplings between different source code entities [14,15].

Fig. 2. Couplings between entities

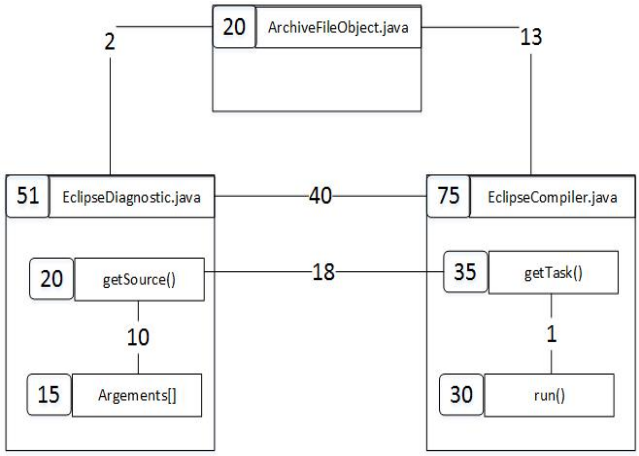


Figure 2 is an example of couplings between different source code entities in org.eclipse.jdt.internal.compiler.tool of Eclipse. Every file and entity are shown with their changed times. Here, EclipseDiagnostic.java and EclipseCompiler.java are changed together 40 times, while EclipseDiagnostic.java has been changed 51 times, which means there are only 11 times EclipseDiagnostic.java being changed without EclipseCompiler.java being changed. In light of this, the coupling between EclipseCompiler.java and EclipseDiagnostic.java is strong. Similarly we can observe that the coupling between EclipseDiagnostic.java and ArchiveFileObject.java is weak. These couplings between

different entities of the source code can only be discovered obviously through evolutionary history mining.

There are three steps of mining association rules from a software system. Firstly, we extract evolutionary history from revision control system. Secondly, we need to preprocess the data to get transaction change sets. Finally, frequent pattern mining algorithms are used to discover association rules [18].

A. Extract Evolutionary History

There are quite a few revision control systems like GIT, SVN and CVS. Most software development teams nowadays use them to store the evolutionary histories [16-18]. What we want to do in this step is to extract the change logs of evolutionary histories from revision control systems. The logs extracted are mostly in file change level, which is not enough for further mining and detection. Some revision control systems can provide line differences of the change in a commit, through which we still cannot get the relations between the change and the affected entities.

In order to get changes in suitable granularity, the change logs are analyzed. For a pair of consecutive snapshots, we use code analysis tool to compare the source code of these two snapshots, and extract the changes between them in the following form:

(commit_id, entity_type, entity_name, change_type)

Commit_id is the ID of the commit. Entity_type is the type of entity, which can be the one of the values: class, function, and field. Entity_name is the name of the changed entity. Change_type is one of the values: added, deleted, and modified.

B. Preprocess Data

Once all the change items are extracted, they need to be preprocessed according to our purpose. From the original definition by Agrawal [13], association rule mining is defined as:

Let $I=\{i_1,i_2,...,i_n\}$ be a set of n binary attributes called items. Let $D=\{t_1,t_2,\ldots,t_m\}$ be a set of transactions called the database. Each transaction in D has a unique transaction ID and contains a subset of the items in I. A rules is defined as an implication of the form $X=>Y$ where $X, Y \in I$, and $X \cap Y = \varnothing$. The sets of items X and Y are called antecedent and consequent of the rule respectively [19, 20].

In the context of bad smell detection, the concept of transaction will be the sets of files, which are changed in the revision control system by a single author within a short time frame. In this step, we need to construct transactions of change histories. All the changes in a commit are treated as in the same transaction. Then we set the time T_{fr} as the frame of a transaction. All the change of a file within T_{fr} is added to the transaction. The time frame is reset to T_{fr} once a related change is added to the transaction. We repeat this process until there are no more changes to the file happening within T_{fr}.

C. Apply Frequent Pattern Mining Algorithms

Association rule mining is an unsupervised machine learning technique for finding sets of items that happen frequently enough among the transactions in a database [22-24]. Concretely we use frequent pattern mining to find recurring sets of items. The strength of an association rule is determined by its support and confidence:

$$Support = \frac{X \bigcup Y}{T} \quad (1)$$

$$Confidence = \frac{X \bigcup Y}{X} \quad (2)$$

where X is the antecedent of the rule, Y is the consequent of the rule, and T is the total number of change sets extracted from last step. It is not easy to find all frequent patterns efficiently because the performance can be exponential with respect to the number of items in D. So we use FP-growth as our mining algorithm. FP-growth uses FP-tree as a compact data structure to encode the database, which can accelerate the mining process. Each node of FP-tree stands for a frequent item in D, except for the root node which represents null. In FP-tree, a path from root to a node represents a collection of transactions in D, each of which contains all the items on the path. FP-growth finds frequent patterns by a depth first approach of recursively mining a pattern of increasing cardinality from the FP-tree. The process decomposes the FP-tree into smaller FP-tree which represents a partition of D.

IV. Heuristics Based Bad Smell Detection

The association rules mined from evolutionary history data are the main input to our heuristics based bad smell detection algorithms. In addition, we will use static code analysis tool to help get some code structural information that is necessary for further detection. In this paper, we mainly focus on detecting the following bad smells: duplicated code, shotgun surgery, and divergent change. The reason why we choose these bad smells is that they have evolutionary characteristic. Meanwhile, these bad smells have been proved to be harmful to software systems. There are several existed approaches having been devised to detect them, however all of these approaches only use code structural information extracted from a single snapshot of software systems. The precision, recall, and F-measure of these approaches are low.

In the following we describe the bad smells to detect individually, and present our heuristics based algorithms for detecting them.

A. Duplicated Code

Software systems that show the same or similar code structure in more than one place are affected by duplicated code [19]. Duplicated Code is a potentially serious problem which can affect the maintainability and comprehensibility of the software system. All of existing approaches develop static code analysis tool to compare each two of the source code of the system to get a complete list of duplicated code.

But we argue that this is not practical in software development. That is not only because finding all the duplicated is time-consuming, but also some found duplicated code is stable which means it is not often changed. We should concentrate on the "dynamic" duplicated code that often happen in the system evolutionary history, and is more harmful to software quality.

Let s be a piece of code that appears several times in the software development process. If in a considered time, s appears more than α times, we define it as an instance of duplicated code.

There are two types of duplicated code considered as most harmful [19]. One is the duplicated code within a class, another is the duplicated code in two sibling subclasses. Because those two types of duplicated code can cause more labor force in software development. So, the duplicated code discussed in our context are those that occur frequently in the software development process, which wastes labor force.

Based on the discuss above, for the duplicated code within a class or sibling subclasses, our approach will walk through the mined association rule sets, and find the frequent sets that contain the same class or sibling subclasses, then, we will use CCFinder to compare the source code in the set. If similar or identical code change is found for more than α times, a duplicated code candidate is reported. In the experiment, α will be tuned on a training data set from a different software system to get the highest F-measure, then it will be used in the testing data set of the target software system.

B. Shotgun Surgery

According to Fowler, shotgun surgery exists when a change to the class triggers many little changes to several other class. This bad smell makes developer hard to correctly locate the needed changes and may even miss some changes since they are all over the place [27, 29].

Let t be a class under consideration. If changedTogether(t) > β, the class is detected as shotgun surgery, where changedTogether(t) means the number of classes that has at least one function changed with other functions.

According to the definition of shotgun surgery, we argue that if a class has shotgun surgery, it must have at least one function that is often changed together with several other functions of other classes. In order to detect this smell, we walk through the mined association rules to find the class having at least one function that is changed with other functions changed in more than β different classes. Parameter β will be tuned automatically in the similar method as α.

C. Divergent Change

Fowler defines divergent change as when a class is changed in different ways for different reasons. A class that is affected by divergent change has low cohesion [28, 30, 31]. This is harmful in software development because any change to handle a variation should change a single class, and all the typing in the new class should express the variation.

Let u be a class under consideration. If numOf(u) >γ, the class is detected as divergent change, where numOf(u) means the size of functions in u that changes together, and do not change together with other functions in other sets.

Based on the discuss above, we scan all the mined rules to find those between functions within the same class. If all the functions in the set change together, and do not change together with other functions in other sets, we identify it as a candidate. The minimum size of the set would be γ. The tuning of γ is the same as duplicated code.

V. Experiments

The purpose of these experiments is to be a proof of concept demonstrating the applicability of our approach based on evolutionary data mining for bad smell detection. The goal of our study is to improve the quality of programs by improving the bad smell detection. The quality focus is to provide a more complete and accurate approach to detect bad smells compared to the existing static code analysis methods.

The context of the study consists of five software projects, namely Eclipse, jUnit, Guava, Closure-Compiler, and maven. Eclipse is an integrated development environment which can be used to develop applications in many programming languages. JUnit is a unit testing framework for the Java programming language. Guava is a project of Google which contains several Google's core libraries that Google relies on in their Java-based projects. Closure Compiler is a compiler for JavaScript which makes JavaScript download and run faster. Maven is a software project management and comprehension tool.

Table I shows the summary of these projects including evolutionary duration and the size of the project. The chosen projects are mainly written in Java, which can reflect the main concept of object-oriented programming. And their entire histories of development are stored in revision control systems from which we could extract related information. Moreover, the evolutionary history lengths of chosen projects vary from 5 years to 13 years, and the commit counts vary from 2000 to 20000.

TABLE I. Summary of Projects

Project	Duration	#commits
Eclipse	Jun 2001 ~ Oct 2014	21523
jUnit	Dec 2001 ~ May 2014	2030
Guava	Oct 2009 ~ Nov 2014	2751
Closure Compiler	Oct 2010 ~ Aug 2014	4970
Maven	Jan 2004 ~ Nov 2014	10025

A. Research Questions

We present the results of some experiments aimed to answer the following research questions:

- RQ1: Does our approach perform well in detecting bad smells? This research question is to evaluate the

accuracy of detecting the three types of bad smells: duplicated code, shotgun surgery, and divergent change.

- RQ2: Is our detection approach based on evolutionary data mining better than or complement to other state-of-art approach? This research focuses on comparing our approach to other approaches based on static code analysis. The result of this research will show the usefulness of evolutionary data in detecting bad smells.

B. Experimental Design

In order to answer RQ1, we apply our approach on the selected open source software systems. We will start from a snapshot not too long ago in the development process of software system. For example, given the evolutionary history data of Maven from September 2003 to October 2014, we may select a snapshot in January 2011. The reason why we select a snapshot not far away from now is that we want to simulate that developers use our approach in the development process. Then, our detection approach is applied on the remaining history of the software system to see whether or not our approach truly find the bad smells that affect the system frequently. Table II shows the detection and validation duration of five projects used in our experiment.

TABLE II. DETECTION AND VALIDATION PERIOD OF PROJECTS

Project	Detection Duration	Validation Duration
Eclipse	Jun 2001 ~ Nov 2010	Dec 2010 ~Oct 2014
jUnit	Dec 2001 ~ Nov 2010	Dec 2010 ~ May 2014
Guava	Oct 2009 ~ May 2012	Jun 2012 ~ Nov 2014
Closure Compiler	Oct 2010 ~ Jun 2012	Jul 2012 ~ Aug 2014
Maven	Jan 2004 ~ Jan 2010	Feb 2010 ~ Nov 2014

In order to evaluate the accuracy of our approach, we need a set of benchmark data so that we can compare our approach to it. Unfortunately, as far as we know there is no such data set available. So we have to manually build the set on our own. Three Master students from Shanghai Jiao Tong University were invited to manually detect the bad smells. They analyzed the snapshot of systems, and looked for bad smells according to their definitions, not aware of our detection approach in advance. And other three master students verified whether the found bad smells are correct or not. The verified bad smells forms the complete set.

After we get the set of bad smells detected from our approach, we can evaluate our detection approach using recall and precision, which are well-known metrics in information retrieval and pattern recognition.

$$precision = \frac{TP}{TP+FP} \quad (3)$$

$$recall = \frac{TP}{TP+FN} \quad (4)$$

Where TP represents the number of true positive bad smells detected, FP represents the number of false positive bad smells detected, FN represents the number of false negative bad smells detected. And then we use F-measure which is the harmonic mean of precision and recall to combine precision and recall.

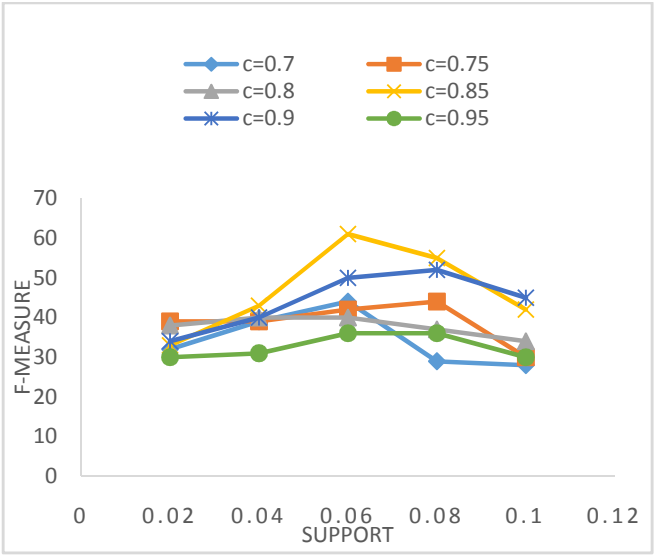
$$F\text{-}measure = 2 * \frac{precision * recall}{precision + recall} \quad (5)$$

In order to answer RQ2, we apply bad smells detection approaches on the same snapshot of the system previously chosen when answering RQ1. The compared approaches we chose are all based on structural information extracted from source code. As for duplicated code, we compare our approach against SonarQube (http://www.sonarqube.org). It is an open platform to manage code quality, and covers the seven axes of code quality, including duplications. As for shotgun surgery, we compare our approach with Décor [10], while as for divergent change, we compare our approach with jDeodorant [30]. Furthermore, we compare our approach against DCPP (Design Change Propagation Probability), which is a tool to detect divergent change and shotgun surgery solely based on structural information. DCPP starts by building an n*n matrix, where n is the number of classes in the system. The matrix is called design change propagation probability. Entry Aij in DCPP represents the probability that a change in the class i triggers a change to the class j. All the tools chosen are the best current detection approaches of the three bad smells .

C. Parameters Tuning

To apply our approach, there are five parameters need to set, including the support and confidence of association rule mining, α of duplicated code, β of shotgun surgery, and γ of divergent change. We performed the calibration on a different software system, tomcat, since it has a long evolutionary history.

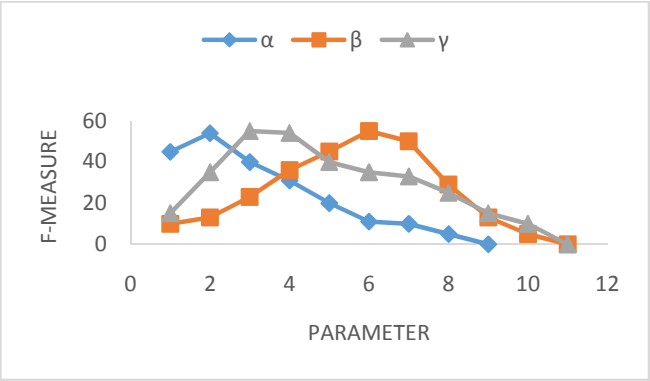
Fig. 3. Calibration of support and confidence



In order to calibrate support and confidence, we varied the value of support from 0.02 to 0.08 with the step of 0.02, and confidence 0.7 to 0.95 with the step of 0.05. And we tried all the combination of support and confidence, then calculated the F-Measure of the combination. Figure 3 is the result of our calibration, where we can see that F-Measure gets the maximum value when support equals 0.06 and confidence equals 0.85.

As for α, we varied the value of α from 1 to 9 with the step of 1, and calculated the F-Measure accordingly. Figure 4 shows the result. The maximum value of F-Measure is obtained with α equaling 2. As for β, we varied the value of β from 1 to 11 with the step of 1, and the maximum value of F-Measure is obtained with β equaling 6. As for γ, we varied the value of γ from 1 to 11 with the step of 1, and the maximum value of F-Measure is obtained with γ equaling 3.

Fig. 4. Calibration of α, β, γ



In summary, Table III presents the values of the chosen parameters using in the following experiment.

TABLE III. VALUES OF PARAMETERS

Parameter	Value
support	0.06
confidence	0.85
α of duplicated code	2
β of shotgun surgery	6
γ of divergent change	3

D. Results Discussion

In this section, we will report and discuss the results of our experiment to answer the two research questions in subsection A.

1) Experimental Results on Duplicated Code

For duplicated code, Table IV shows the experimental result of duplicated code detection in terms of precision, recall and F-Measure. OP stands for our approach, SQ stands for SonarQube. We can observe from Table IV that our approach achieves both high precision and recall, thus its F-Measure is also high. SonarQube (SQ) achieves even higher recall than our approach, but its precision rate is very low. The reason is that SonarQube simply scans all the source code of the snapshot and gets all the code pieces that are similar or identical. But in fact, some of these code pieces are never or seldom changed in the evolutionary history which is not considered harmful compared to those frequently changed duplicated code. SonarQube finds much more duplicated code than our approach where most of them are not harmful, and just make it impractical for developer to fix the most urgent bad smells. So, SonarQube achieves low F-measure in detection of duplicated code.

TABLE IV. DUPLICATED CODE DETECTING RESULTS BY OUR APPROACH AND SONARQUBE

Project	Precision		Recall		F-Measure	
	OP	SQ	OP	SQ	OP	SQ
Eclipse	90%	20%	80%	100%	84.7%	33.3%
jUnit	80%	15%	90%	100%	84.7%	26.1%
Guava	90%	8%	60%	95%	72%	14.8%
Closure Compiler	75%	10%	85%	100%	79.7%	18.1%
Maven	90%	5%	80%	100%	84.7%	9.5%

2) Experimental Results on Shotgun Surgery

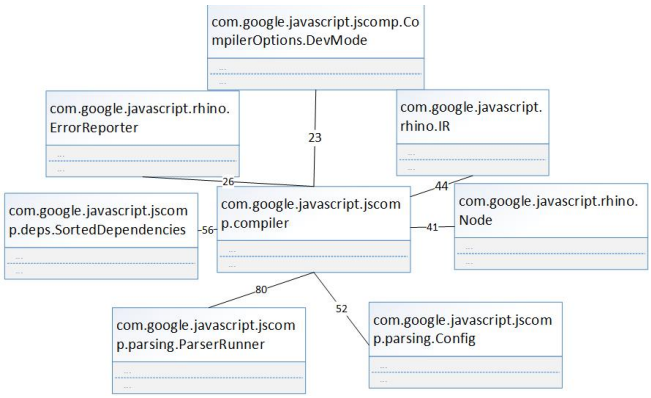
As for shotgun surgery, unlike duplicated code, there is only several candidates that we manually found. Table V shows the result of detecting shotgun surgery on five software systems using our approach and Décor. We can see that our approach manages to detect all the shotgun surgery smells. There is only one false positive in our approach on Maven. DE stands for Décor, which is a bad smell detection tool solely based on static code analysis. We cannot find any candidates in the software systems using Décor. It is highly difficult to detect shotgun surgery solely based on analyzing the structural information of a software system from a single snapshot.

TABLE V. SHOTGUN SURGERY DETECTING RESULTS BY OUR APPROACH AND DECOR

Project	#candidates	Precision		Recall		F-Measure	
		OP	DE	OP	DE	OP	DE
Eclipse	2	100%	0%	100%	0%	100%	0%
jUnit	0	-	-	-	-	-	-
Guava	1	100%	0%	100%	0%	100%	0%
Closure Compiler	1	100%	0%	100%	0%	100%	0%
Maven	3	75%	0%	100%	0%	85.7%	0%

Figure 5 shows an example of shotgun surgery found by our approach in Closure Compiler. The bad smell exists in the class com.google.javascript.jscomp.compiler. Our approach detected association rules between this class and other 29 functions which belong to seven classes. The association between com.google.javascript.jscomp.compiler and other seven classes is shown with lines marked with change times. Such a smell can do harm to software systems during its evolution, and needs to be refactored.

Fig. 5. A shotgun surgery instance in closure compiler



3) *Experimental Results on Divergent Change*

As for divergent change, Table VI presents the result of our approach and jDeodorant. Our approach achieves both higher precision and recall rate, and thus its F-Measure can achieve higher than 50%. JD stands for jDeodorant, which is solely based on code structural information. jDeodorant can also detect some of the divergent change instances, but its precision, recall and F-Measure are lower than our approach. These results show that our approach performs better than static analysis of single snapshot of a software system.

TABLE VI. DIVERGENT CHANGE DETECTING RESULTS BY OUR APPROACH AND JDEODORANT

Project	Precision		Recall		F-Measure	
	OP	JD	OP	JD	OP	JD
Eclipse	50%	35%	70%	40%	58.3%	37.3%
jUnit	60%	20%	50%	20%	54.5%	20%
Guava	100%	40%	100%	50%	100%	44.4%
Closure Compiler	90%	15%	80%	20%	84.7%	17.1%
Maven	100%	20%	80%	40%	88.9%	26.7%

4) *Experimental Results on Our Approach vs. DCPP*

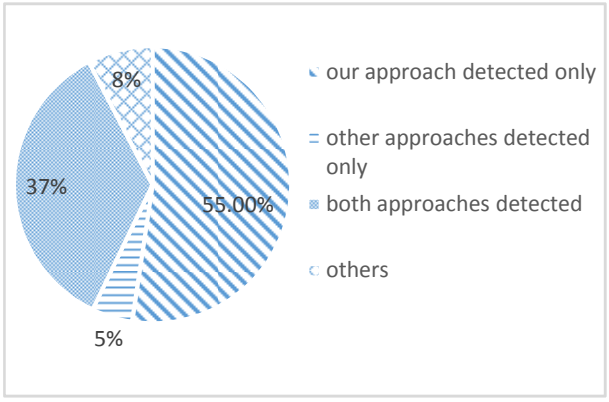
Table VII shows the result our approach and DCPP for detection of divergent change and shotgun surgery. The results show that our approach outperforms DCPP.

TABLE VII. DIVERGENT CHANGE & SHOTGUN SURGERY DETECTING RESULTS BY OUR APPROACH AND DCPP

Project	Precision		Recall		F-Measure	
	OP	DCPP	OP	DCPP	OP	DCPP
Eclipse	60%	20%	50%	40%	54.5%	26.7%
jUnit	50%	50%	40%	30%	44.4%	37.5%
Guava	80%	55%	70%	40%	74.7%	46.3%
Closure Compiler	80%	50%	70%	60%	74.7%	54.5%
Maven	70%	40%	50%	40%	58.3%	40%

Figure 6 presents a concrete instance of divergent change in Eclipse detected by our approach. This instance exists in org.eclipse.jdt.api.ui.internal.preferences of the Eclipse project. The class is used to organize controls and keys within a property/preference page. In this class, our approach detected four sets of functions that change divergently during the evolutionary history, which is separated by dashed lines in the figure. The first set of functions is related to the GUI widgets control in the preference page. The second set of functions control the state change of the preferences page. The third set of functions is used to get values from preferences page. The fourth set of functions is related to the update of the page. These four set of functions conduct different responsibilities, and they are changed independently from each other as detected by our approach.

Fig. 6. A divergent change instance in Eclipse



5) *Experimental Results on Complementarity between Our Approach and Other Static Code Analysis*

Figure 7 shows the complementarity between our approach and other static code analysis approaches for detecting duplicated code, shotgun surgery and divergent change. From the figure we can see that although our approach performs better than other approaches, there are still 5% instances of bad smells that our approach cannot detect while other approaches can do. This result reveals that structural information extracted from a single snapshot of software system can be a complement to our approach. So combing the two techniques properly can yield an even better approach for code bad smells detection.

6) *Experimental Results on Comparison between Different Periods*

Table VIII presents the comparison between the smells in detection period and the smells that reoccur in validation period. The results show that for duplicated code, most of them would reoccur in the later development of the software system. While for shotgun surgery and divergent change, almost every one of the smells detected in detection period would reoccur in validation period. These bad smells that reoccur frequently can cost a lot of labor expense, and make the development progress slow. Moreover, ignoring these bad smells can make the software system become more and more complex, where the software system may grow out of control, making it harder for developers to understand them, to fix bugs, and to add new features. The results show that

our approach can truly detect the most urgent bad smells that would occur in later development of the software system if not refactored immediately.

Fig. 7. Complementarity between our approach and others

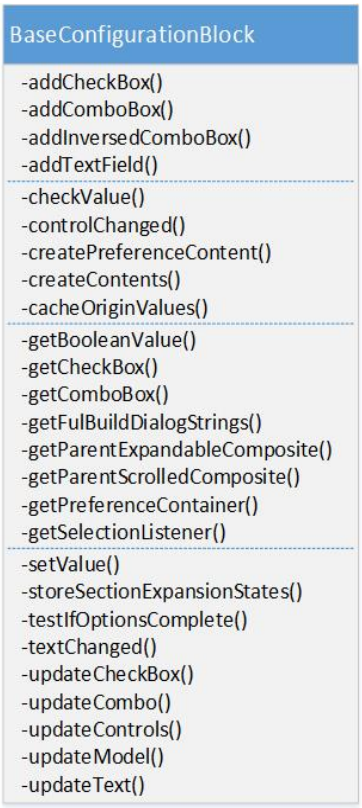


TABLE VIII. COMPARISONS BETWEEN DIFFERENT PERIODS

Project	#smells in detection period(duplicated code/shotgun surgery/divergent change)	#smells reoccur in validation period(duplicated code/shotgun surgery/divergent change)
Eclipse	40/2/10	23/2/5
jUnit	50/0/4	33/0/4
Guava	33/1/2	14/0/2
Closure Compiler	51/1/6	37/1/4
Maven	66/3/7	40/3/3

In summary, for RQ1, our detection approach based on evolutionary data mining achieves high accuracy for detecting duplicated code, shotgun surgery and divergent change. Its F-measure value is higher than 50%. In fact, this result is quite expected, because the bad smells we tried to detect all have the property of being detected from evolutionary history. Through some concrete instances of bad smells detected by our approach, the experiments show that our approach can detect the truly harmful bad smells that exist in the software systems.

For RQ2, our approach all performs better than other detection tools which are solely based on the structural information extracted from a single snapshot of software systems. And we also find out that static code analysis can complement to our approach. So combining evolutionary data mining and static code analysis is a good way to further improve the detection performance of bad smells.

VI. CONCLUSION AND FUTURE WORK

In this paper, we proposed a novel approach to detect three different code bad smells by mining evolutionary history of the software systems from revision control system. The bad smells we chosen to detect are all evolutionary history related: duplicated code, shotgun surgery, and divergent change. We use frequent pattern mining to extract association rules from revision control system. For each code bad smell, we explain in detail the property of the bad smell in terms of evolutionary history. And based on the association rules mined, heuristics based algorithms are proposed to detect the target bad smells.

We evaluated our approach on five open source projects through three experiments. (1) We compare the detected results to our manually built benchmark data. The results show that our approach achieves precisions between 50% and 100%, recalls between 60% and 100%, and F-measure between 58% and 100%. (2) We also compare our approach to other competitive approaches using structural information extracted from source code. The results show that our approach performs better other approaches in terms of history change related bad smells, while structural information based approach can be a good complement to our approach. (3) Furthermore, we apply our approach on the remaining evolutionary history of the five projects to see how the detected bad smells now can affect the future development of a software systems. The results show that our approach can truly detect harmful bad smells that occur frequently in the evolutionary history of the software systems.

In the future, we will validate our approach on different types of software systems. Also, we plan to combine our approach with static code analysis technique to improve the precision and recall of bad smell detection. Furthermore, we will research on search based automated refactoring to remove bad smells.

ACKNOWLEDGMENT

This research is supported by 973 Program in China (Grant No. 2015CB352203) and National Natural Science Foundation of China (Grant No. 61472242).

REFERENCES

[1] M. Fowler, Refactoring: Improving the Design of Existing code. Addison-Wesley, 1999.

[2] 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.

[3] A. J. Riel, Object-Oriented Design Heuristics. Addison-Wesley, 1996.

[4] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to

increase software quality," in Proceedings of the 14th Conference on ObjectOriented Programming, Systems, Languages, and Applications. ACM Press, 1999, pp. 47–56..

[5] F. Simon, F. Steinbr, and C. Lewerentz, "Metrics based refactoring," in Proceedings of 5th European Conference on Software Maintenance and Reengineering. Lisbon, Portugal: IEEE CS Press, 2001, pp. 30–38.

[6] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02). IEEE CS Press, Oct. 2002.

[7] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in Proceedings of the 11th International Software Metrics Symposium. IEEE Computer Society Press, September 2005.

[8] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in Proceedings of the 9th International Conference on Quality Software. Hong Kong, China: IEEE CS Press, 2009, pp. 305–314.

[9] M. Mantyla. Bad Smells in Software - a Taxonomy and an Empirical Study. PhD thesis, Helsinki University of Technology, 2003.

[10] 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.

[11] T. Kamiya, S. Kusumoto, and K. Inoue, "CCfinder: a multilinguistic token-based code clone detection system for large scale source code," Transactions on Software Engineering, no. 4, 2002.

[12] A. Rao and K. Reddy, "Detecting Bad Smells in Object Oriented Design Using Design Change Propagation Probability Matrix", in International MultiConference of Engineers and Computer Scientists, 2008.

[13] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, 1993, pp. 207–216.

[14] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting, ser. IWPSE '07. New York, NY, USA: ACM, 2007, pp. 31–34.

[15] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution ofbad smells in object-oriented code," in International Conference on the Quality of Information and Communications Technology (QUATIC). IEEE, 2010, pp. 106–115.

[16] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," in Proceedings of the International Workshop on Refactoring Tools. ACM, 2011, pp. 33–36.

[17] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in ICSE '04: Proceedings of the 26th International Conference on Software Engineering, 2004, pp. 563–572.

[18] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," IEEE Transactions on Software Engineering, vol. 30, no. 9, pp. 574–586, 2004.

[19] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in Proceedings of the International Conference on Software Engineering, 2010.

[20] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and R. Daniel, "Using concept analysis to detect co-change patterns," in Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting, ser. IWPSE '07. ACM, 2007, pp. 83–89.

[21] A. Mockus and L.G. Votta, "Identifying Reasons for Software Changes Using Historic Databases,"Proc. Int'l Conf. Software Maintenance (ICSM 2000),pp. 120-130, Oct. 2000.

[22] A. Michail, "Data Mining Library Reuse Patterns Using Generalized Association Rules,"Proc. Int'l Conf. Software Eng.,pp. 167-176, 2000.

[23] I. H. Witten and E. Frank. Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann, 1st edition, October 1999.

[24] J. Sayyad-Shirabad, T.C. Lethbridge, and S. Matwin, "Mining the Software Change Repository of a Legacy Telephony System,"Proc. Int'l Workshop Mining Software Repositories (MSR 2004), pp. 53-57, 2004.

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

[26] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting, ser. IWPSE '07. New York, NY, USA: ACM, 2007, pp. 31–34.

[27] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "The use of development history in software refactoring using a multiobjective evolutionary algorithm," in Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, ser. GECCO' 13. ACM, 2013, pp. 1461–1468.

[28] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in European Conference on Software Maintenance and ReEngineering. IEEE, 2012, pp. 411– 416.

[29] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in International Conference on Software Maintenance (ICSM). IEEE, 2012, pp. 306–315.

[30] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "JDeodorant: identification and application of extract class refactorings," in Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. ACM, 2011, pp. 1037–1039.

[31] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in Reverse Engineering (WCRE), 2010 17th Working Conference on, 2010, pp. 119–128.