Detection of God Classes in Object-Oriented Software Using Decision Tree

Sinan Eski, Feza Buzluca
Computer Engineering
Istanbul Technical University
{eski, buzluca}@itu.edu.tr

Abstract—**Bad smells, also known as code smells or disharmonies are characteristics of software that may indicate a code or design problem because of insufficient abstraction that makes software hard to evolve and maintain, and may trigger refactoring of code. Since these defects reduce understandability, flexibility and reusability of the system; adding new features, adapting to the changes, and finding design flaws are getting harder. In order to reduce software cost, especially in the maintenance, detection of bad smells is important. One of the most common bad smells is "God Class". A class is identified as God Class if it interacts with many other classes, and has a very high complexity and low cohesion. In this paper, a method that is based on decision tree is proposed for detecting God Classes. It is a supervised learning method and creates models to classify unseen samples based on training data. In this study, training data is formed as follows: software classes are samples, metrics are their attributes and tags show whether the class is a God Class or not. A database is prepared to train and validate our decision tree model, using information that is collected from open-source projects. The adaptable model can be calibrated during the development of a project by training it with detection results of earlier versions so that it can detect defects in later versions successfully. The model can be also prepared to work on software systems of a specific domain by training the model with a data collected from systems in the same domain. Empirical results of the study indicate that the decision tree method can be used successfully to detect God Classes.**

Keywords; Software Quality; Bad Smells, Disharmonies; God Classes; Decision Tree

I. INTRODUCTION

Software maintenance cost is typically more than 50% of the cost of the total software life cycle. Detection of bad smells in design during software development and maintenance is an important task in order to reduce development time and resource cost.

Software quality assessment is an important key factor to determine critical parts, because high-quality parts of software are less error-prone and easy to maintain. On the other hand, more complex parts with a lower quality level have the opposite situation. The low-level quality parts mostly have design-level defects, which show weaknesses in design and present symptoms that are called bad smells. Bad smells usually do not indicate bugs and they do not affect the program functioning; but may slow down the development or increase the risk of bugs or failures in the future. These parts need refactoring. At that point, the measurement of software quality is important to expose the good and bad parts of a software system.

Software design metrics are basic objects that can be used for the quality measurement. Metrics are the measurement values of software entities like lines of code or number of methods that could be extracted from source codes. In object-oriented software, in order to recognize the structure of classes and the relations among them, there are also high-level metrics to express concepts such as cohesion, coupling and complexity. As object-oriented software metrics give important evidence about design quality, they can help software engineers to determine critical low-quality parts, that should be considered firstly on maintenance and refactoring. Therefore, developers start refactoring from these low-quality parts to improve the internal quality of their product.

It is difficult to work with metrics to create certain rules for detecting bad smells because of their various types and distributions. Also, different metrics should be used together to create a model for quality assessment; but it is difficult to determine the roles, weights and thresholds of metrics in creating such a model. In this paper we propose a method for detecting the well-known bad smell called "God Class" [24], in designs of object-oriented systems by using decision tree. God classes have high complexity and incorporate more than one functionality that semantically belongs elsewhere; therefore they have negative impact on quality attibutes like maintainability, reusability and understandability. The decision tree can generate the rules that consist of multiple metrics by learning the properties of the systems that are examined in advance. The detection models that are created in a decision tree form can adapt to different types of software systems if they are trained with proper data. The user of the model does not need to be aware of the specific combinations of metrics or their thresholds.

The creation of the decision tree is based on metrics of software classes. In order to obtain the results and validate our method we analyze long-lived open source projects, which are in different sizes and domains. We prepare a database using these projects and save metric values of classes and their actual God Class information. We use these data as train and test sets according to three aspects: in the same project, among different releases of a project and among different projects. For the first aspect, we split data of a software system into train and test sets. Here we try to find out the answer of the following question. When we know some classes of a software system that are God Class or not, can we classify the rest of them? For the second aspect, we use data from a release of a software system as train set and then data obtained from one of the following releases as test set; because we want to discover whether the method can be used to detect defects in successor releases of a software system when we have information about defects in an earlier release of the same system. For the final aspect, we use data from different software systems for the train and test sets, because we want to find out whether a model that is created using data from a software system can be used to detect God Classes in other software systems.

In this study, we select metrics that are defined in CK suit [1], QMOOD [2] and in [13] because of their possible relation to God Classes. We give all metrics, which are explained in Section 3 as parameters to the C4.5 algorithm [19] that combines poroper metrics and creates decision tree models to find God Classes. The results show that this approach can be successfully used to find this common software bad smell.

The remainder of this paper is organized as follows: Related works are presented in the next section. Dataset and metrics that are used in empirical analysis are presented in Section 3. Section 4 shows the proposed method and the system architecture, and Section 5 shows the case studies and obtained results. Final section concludes the paper.

II. RELATED WORK

Webster [21] wrote the first book on “antipatterns” in object-oriented development; his contribution covers conceptual, political, coding, and quality-assurance problems. The code smells concept was introduced by Fowler and Beck [3], who defined 22 different kinds of smells and suggested where developers should apply refactorings. Later, other authors like Mäntylä [22] identified more smells and proposed their classifications.

Several approaches to specify and detect smells have been proposed in the literature. Travassos et al. [17] introduced manual inspections and reading techniques to identify code smells. Marinescu presented general approaches to find code smells [4] and after this work Marinescu et al. [5, 6, 7], presented metric-based detection strategies, which capture deviations from good design principles. These strategies consist of combining metrics with set operators and comparing their values to absolute and relative thresholds. Lanza and Marinescu introduced the detection strategies in a book [13].

Similarly to Marinescu, Munro [23] proposed metric-based heuristics to detect code smells; the detection heuristics are derived from template similar to the one used for design patterns. He also performed an empirical study to justify the choice of metrics and thresholds for detecting code smells.

Li and Shatnawi [8] work on relation between bad smells and software bugs.

Previous studies are generally based on predefined rules that are used to make boolean decisions on whether a class has a smell or not. In our approach we train our model and generate decision rules using data collected from different exemplary software systems. For different systems, the rules can be regenerated or adapted dynamically. Our approach neither relies on predefined rules nor needs some threshold values that are known in advance.

III. BACKGROUND

A. God Classes

God Class is one of the most common disharmonies, which do not satisfy identity harmony principles. Lanza and Marinescu [13] describe software identity harmony as follows. “Operations and classes should have a harmonious size i.e., they should avoid both size extremities. Each class should present its identity (i.e., its interface) by a set of services, which have one single responsibility and which provide a unique behavior data and operations should harmoniously collaborate within the class to which they semantically belong.”

A class is identified as God Class if it interacts with many other classes, and has a very high complexity and low cohesion opposite to the idendity harmony principle.

B. Studied Metrics

In order to detect God Classes; coupling, cohesion complexity and size-related metrics are selected because of the God Class definition. We collect 13 metrics that are defined in CK [1] metric suite, QMOOD [2] and [13]. The decision tree method creates the model dynamically according to the training data by selecting the specific combinations of metrics and their thresholds.

Studied metrics are separated and explained in the following three groups.

1) Coupling Metrics

ATFD (Access to Foreign Data): It is the number of classes whose attributes are directly or indirectly reachable from the investiggated class. Classes with a high ATFD value rely strongly on data of other classes and that can be the sign of the God Class defect.

CBO (Coupling Between Object Classes): It is the number of classes that a class is coupled to. It is calculated by counting other classes whose attributes or methods are used by a class, plus those that use the attributes or methods of the given class. Inheritance relations are excluded. As a measure of coupling CBO metric is related with reusability and testability of the class. More coupling means that the code becomes more difficult to maintain because changes in other classes can also cause changes in that class. God Classes tend to be having more connectivity to other classes.

RFC (Response For a Class): It is the number of the methods that can be potentially invoked in response to a public message received by an object of a particular class.

If the number of methods that can be invoked from a class is high, it is more complex and highly coupled to some methods. God Classes tend to invoke a lot of methods from other classes.

DIT (Depth of Inheritance Tree): It is the maximum length from the class being measured to the root of the inheritance tree. If it is not extended from any class this value is 0. If there are more than one ancestor branch it is the longest way to the root. A class that is located deeply in the inheritance hierarchy inherits a lot of properties from many ancestor classes. Inheritance is also another type of coupling. Deeper trees constitute greater design complexity, since more methods and classes are involved. It is difficult to to predict its behavior, design and maintain such a class; therefore it is likely to contain a defect.

2) *Cohesion Metrics*

TCC (Tight Class Cohesion): Methods that use at least one attribute of class in common are defined as strongly-connected. TCC shows the ratio of strongly-connected method couples to all method couples. Low values of TCC may indicate that the cohesion of the class is low because it is not well designed.

LCOM (Lack of Cohesion in Methods): It is a measure about how methods of a class are related to each other. Low cohesion means that the class implements more than one responsibility. A change request by either a bug or a new feature, on one of these responsibilities will result change of that class. Lack of cohesion also influences understandability and implies classes should probably be split into two or more subclasses. God Classes tend to be doing more irrelevant and complex tasks.

CAM (Cohesion Among Methods): It is a measure of cohesion based on parameter types of methods instead of attributes used by them. God Classes tend to have various unrelated responsibilities therefore they should have methods that have different type of parameters.

3) *Complexity and Size Metrics*

WMC (Weighted Methods per Class): It is the weighted sum of all class' methods an represents the complexity of a class. It is equal to number of methods, if the complexity is taken as 1 for each method. More complex classes tend to be a God Class. It is expected that complex classes have higher change rates because of bug fixing and refactoring activities.

LOC (Lines of Code): It is the number of all nonempty, non-commented lines of the body of the class and all of its methods. LOC is a measure of size and also indirectly related to the class complexity. Since God Classes are complex, their lines of code values tend to be high.

NOPA (Number of Public Attributes): The number of public attributes of the class.

NOAM (Number of Accessor Methods): The number of getter and setter methods of the class. These methods are used to access private or protected attributes of the classes.

NOA (Number of Attributes): The number of attributes of the class.

NOM (Number of Methods): The number of methods in a class.

NOPA, NOAM and NOA metrics are related to the number of attributes and NOM metric is related to the number of methods of a class. These metrics can be used to identify god classes, which are difficult to maintain because they contain too many attributes and methods [18].

C. *Studied Software Systems*

In this study the data set is extracted from the following open source projects.

ArgoUML [9] is a long-live UML modeling project.

JFreeChart[10] is an open source drawing library.

Xerces [11] is an Apache project that includes open source XML parser and related software.

These projects are selected since they are well-known long-lived projects.

IV. The God Class Detection Approach

In this study, we propose an approach for detecting the common design defect God Class. We create decision tree models for detecting God Classes using data about examined software classes. For training and evaluating the model, we prepare a database by extracting metric values and God Class information from open source projects.

In our approach, first we extract metric values of software classes from different open-source projects using iPlasma [16] and Equality [12]. Before we give the details of our detection method, we show some properties of software metrics to explain the necessary and benefits of the prposed method. Analysis on the metric values reveals properties of metrics that make it difficult to work with them to create certain rules. First observation is that the metric values are generally power-law distribution. The metric distributions of the Xerces 2.7 project are sperated into two groups and they are given in Figure 5 and Figure 6. In this figures x-axis is the value of the metric and y-axis is the number of classes with a specific metric value. Many classes have same or similar values. Furthermore, metrics have different intervals and various types. Some of them are integers between $[0,+\infty)$ (such as RFC) and some of them are floating numbers between [0, 1] (such as WOC). Interpretation of metrics also varies in detection of bad smells. While high values of some metrics (ATFD, WMC) indicate design defects, in contrast low values of some other metrics (TCC) are warning signals. Also, to detect design defects in a software system different metrics should be used together. To overcome these difficulties we benefit from artificial intelligence and use the decision tree. This method uses attributes, in our case metrics, together to classify software classes. The model is created in a training phase, so that certain thresholds of the metrics do not need to be given manually or know in advance, because they are determined and learned by the model.

A. Preparing the Database

We prepare a dataset for each software project using information that is collected from the classes. In the representation of data: software classes are samples, metrics are their attributes and statuses as “God Class” or “Normal Class” are their tags. We examine the software classes and tag them as God Class if they include this design defect. Defect-free classes are tagged as Normal Class. To assign the tags to classes, we use three different tools [16, 12, 18] as explained below.

We built dataset for each investigated project as follows:

1) We extract software metrics of each class and record them to the database.
2) We determine candidate classes that are possibly God Classes using three different tools [16, 12, 18].
   a) If all the three tools mark a class as a God Class we also tag it as a God Class without a further investigation.
   b) If all the three tools cannot agree upon the status of the class we analyze the source code manually and decide whether the detected candidate class is really a God Class or not.

B. Creating the Detection Model

In this study we use decision tree models to classify software classes as God Class and Normal Class. A decision tree is defined as a connected, acyclic, undirected graph, with a root node, zero or more internal nodes, and one or more leaf nodes. All nodes, except the root and the leaves, are internal nodes. Leaves show the tag of the objects and inner nodes include attributes and their outgoing edges present the rules to compare the object under test with the attributes of the train samples. The rules to examine objects are generated at the train phase by using train samples that are obtained from known projects. In the detection (test) pahase, starting from the root node, attributes of the test object are avaluated according to conditions defined in the inner nodes. Depending on the results of these evaluations the test object moves in the tree and finally reaches the leaf that shows the corresponding tag. Exemplary decision tree models that were created in our study for different software systems are shown in Figure 2, Figure 3 and Figure 4. In our case, leaves (rectangles) show the tag of the tested class, namely “1” represents God Class and “0” represents Normal Class. The inner nodes hold class metrics that take place in the comparison operations, and the edges show the conditions and the branches to move.

In this study, to generate decision trees we applied the C4.5 algorithm, which is developed by Ross Quinlan [19]. At each node of the tree, C4.5 chooses one attribute of the data that most effectively splits the set of samples into subsets so that the samples in each subset belon mostly to a specific category. The attribute with the highest normalized information gain (difference in entropy) is chosen for splitting the samples to branches. The C4.5 algorithm then recurses on the smaller sublists [20] until all the samples in a branch belong to the same category. J48 is one of the implementation of this algorithm in Weka [15]. It generates a pruned or unpruned decision tree. It uses pruning in order to get rid of over-fitting of decision tree, thus it creates smaller and less complex trees [14].

We used pruned version of the algorithm with 0.25 confidence factor in this empirical study. At the initial state we give all metric values and God Class information for all classes to the algorithm to construct the tree model. The algorithm, iteratively, selects proper metrics and related decision conditions (thresholds) to split the set class samples according to the tags (God Class or not). After iterations have been finished, the model is created in the form of a decision tree that includes the detection rules. Then we can use this model to predict God Classes in the validation (test) set that has unvisited class samples. We note that, the location of metrics in the tree and the comparison rules are determined in the training phase automatically and they can be adapted to different systems. The user of the model does not need to select these parameters.

Additionally decision tree method requires a short running time to create models. We measure the time that is taken to create the models on a computer that has 2.4 GHz Intel Core i5 processor, 8 GB RAM and Mac OS X operating system. In all experiments the models are built in less than 0.1 seconds.

C. Validation Methods

To validate our results holdout validation method is used in this article. It is a natural approach to split the available data into two non-overlapping parts: one for training and the other one for testing. The test data is held out and not used during training. Holdout validation avoids the overlap between training data and test data. Figure 1 shows this process.

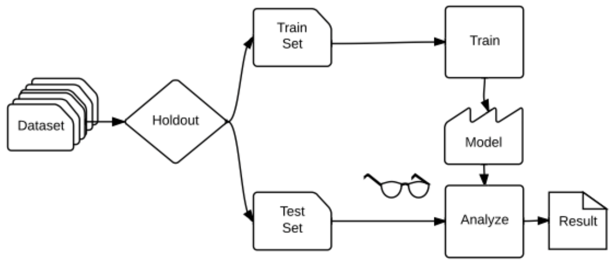

Figure 1. Holdout validation method

D. Measurements and evaluation

To evaluate our method we measure the accuracy, precision, recall and F-measure (F_1) values that are obtained in the experiments. The calculations are based on the status of the results as true positive (tp), true negative (tn), false positive (fp) and false negative (fn). The accuracy gives the portion of correctly classified classes in the whole data set. Precision is the number of software classes correctly classified as “God Class” divided by the total number of classes classified as “God Class”. The precision can also be calculated for “Normal classes”. In this case it will show the number of software classes correctly classified as “Normal Class” divided by the total number of classes classified as “Normal Class”. High precision means that the method generates more correct results than incorrect. Recall is the number of software classes correctly classified as “God Class” divided by the total number of real “God Classes”. Similarly it can also be defined for “Normal Classes”. High recall means that the method can detect most of the “God Classes” (or “Normal Classes”). The F-measure (F_1) enables us to consider the precision and the recall together. It can be interpreted as a weighted average of the precision and recall. The equations are given below:

		Actual	
		1	**0**
Predicted	**1**	**tp**	**fp**
	0	**fn**	**tn**

$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn}$$

$$\text{Precision} = \frac{tp}{tp + fp}$$

$$\text{Recall} = \frac{tp}{tp + fn}$$

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}.$$

If the validation dataset has unbalanced number of samples for God Classes and Normal Classes, accuracy could not be used solely to evaluate the model. In our dataset the number of “God Class” samples is less than “Normal Class” samples, and we aim to predict the “God Classes”. Therefore we consider the value of precision, recall and f-measure together to evaluate the model. For example if our dataset had 1000 samples and only 10 of samples were “God Class” and our model tagged all samples as “Normal” we would reach 99% accuracy, on the other hand God Class precision, recall and f-measure value would be all zero.

V. EXPERIMENTAL RESULTS

We evaluate the performance and practical usability of our approach in three different ways. First, we split one software system’s dataset into train and test sets. Here we want to discover, whether it is possible to train the model with a part of a program and then detect defects in the remaining part of the same program. Second, we use one software system’s different releases as train and test sets. This evaluation method is used to see if we can train the model in earlier development phases of the program and then use the model to detect defects in later releases. Third, we use one software system’s dataset as train and another software system’s dataset as test set. Here we try to train the model using data of a program and then to apply it to other programs to detect defects.

A. Investigating one software system

In this experiment Apache Xerces 2.7, Argo UML 2.4 and JFreeChart 0.9.21 projects are investigated. We use the holdout validation method by splitting dataset into two groups as train and test sets for each project. We randomly select 60% of samples from dataset for each tag, and put them into the train set. The remaining 40% of the dataset constitutes the test set. We create the decision tree model by using the train set, and then measure the success of the model using the test set.

The Xerces project includes 786 Normal and 55 God Classes. After splitting, train set contains 470 Normal Classes and 35 God Classes and test set includes 316 Normal and 20 God Classes. The model that is created after training for this project, is shown in Figure 2.

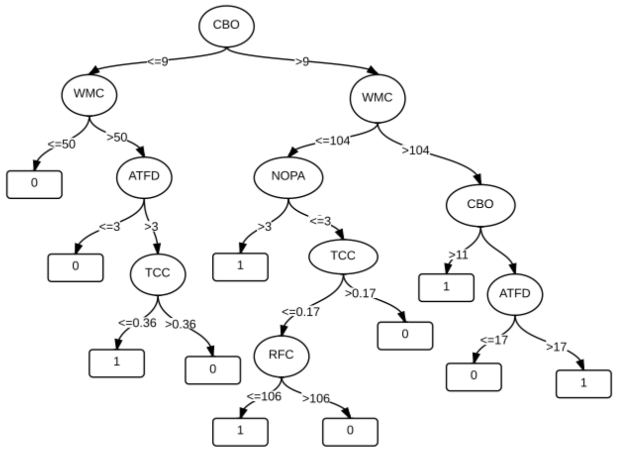

Figure 2. Decision tree model of the Xerces 2.7

When we apply the model on the test set, it detects 18 God Classes correctly. It classifies two of God Classes as Normal Class and tags four of Normal Classes as God Class. Precision, recall and f-measure results for the God Class, Normal Class and their weighted averages are given in Table I. Allthough the most important results are about the detection of the God Classes, to show the overall performance of the method we also give the results about the detection of the Normal Classes. Accuracy of the detection is 98.2% and God Class precision and recall values are 81.8% and 90% respectively. This means that we could find 82% of God Classes and 90% of marked classes are really God Classes.

TABLE I. XERCES 2.7 HOLDOUT VALIDATION RESULTS

	Precision	***Recall***	***F-measure***
God Class	0.818	0.900	0.857
Normal Class	0.994	0.987	0.990
Weighted Average	0.983	0.982	0.983

In the second experiment we run our method on the Argo UML 0.24 project that includes 1584 Normal and 30 God Classes. After splitting the samples into two groups, the train set contains 951 Normal Classes and 17 God Classes and the test set includes 633 Normal and 13 God Classes. The model that is created after training for this project, is shown in Figure 3.

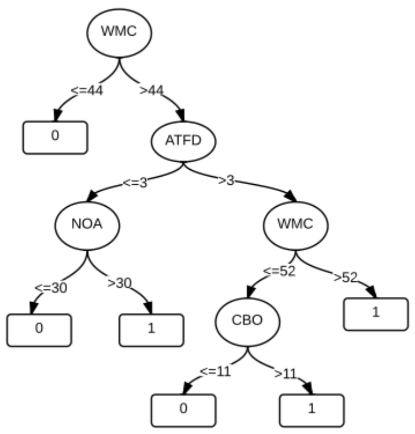

Figure 3. Decision tree model of the Argo UML 0.24

After we run our method on the test set, the results show that the model detects 10 of 13 God Classes correctly. The evaluation measurements are given in Table II. Accuracy of the detection is 99.5% and God Class precision and recall values are 100% and 76.9% respectively. The recall value shows that the classifier did not misclassify any Normal Class as God Class.

TABLE II. ARGOUML 0.24 HOLDOUT VALIDATION RESULTS

	Precision	***Recall***	***F-measure***
God Class	1.000	0.769	0.870
Normal Class	0.995	1.00	0.998
Weighted Average	0.995	0.995	0.995

We test our method also on the JFreeChart 0.9.21 project that includes 603 Normal and 40 God Classes. In this experiment the train set contains 363 Normal Classes and 23 God Classes and test set includes 240 Normal and 17 God Classes. The model, which is created after training for this project, is shown in Figure 4.

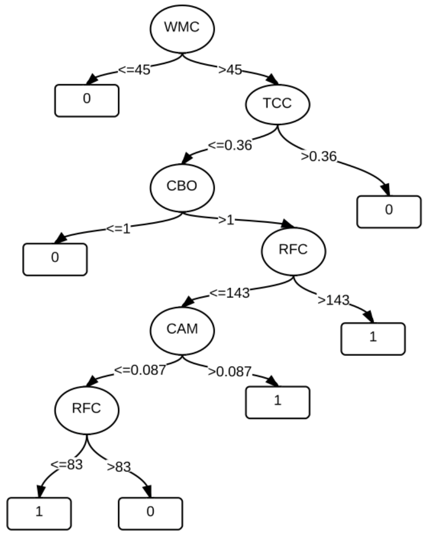

Figure 4. Decision tree model of the JFreeChart 0.9.21

The model detects 17 God Classes correctly. It classifies five Normal Classes as God Class. According to the results in Table III, we could find 100% of God Classes, and 77% of marked classes are really God Classes.

TABLE III. JFREECHART 0.9.21 HOLDOUT VALIDATION RESULTS

	Precision	***Recall***	***F-measure***
God Class	0.773	1.000	0.872
Normal Class	1.000	0.979	0.989
Weighted Average	0.985	0.981	0.982

For each software systems the decision tree creates dynamically different models according to the training set. After this group of experiments we can conclude that the decision tree model can be trained with a part of a software system and then it can be used to detect God Classes in the remainder of the same system.

B. *Investigating different versions of a software system*

In this experiment we investigate two different releases of the ArgoUML: 0.24 and 0.32. They contain 1584 and 1796 Normal Classes; 30 and 27 God Classes respectively. We trained our model using ArgoUML 0.24 and tested it using ArgoUML 0.32.

The results show that the model detects 22 God Class correctly. It tags five of God Classes as Normal Class and two of Normal Classes as God Class. Accuracy of the detection is 99.6% and Table IV shows the other results. We reach in detection of God Classes more than 91% and 81% success for the precision and recall results respectively.

TABLE IV. TRAIN SET: ARGOUML 0.24, TEST SET: ARGOUML 0.32

	Precision	***Recall***	***F-measure***
God Class	0.917	0.815	0.863
Normal Class	0.997	0.999	0.998
Weighted Average	0.996	0.996	0.992

We deduce that we can create a model with data from a particular release and use it in the successor releases of the software system to detect design defects. In some releases, the detection results of the model can be investigated and if it is necessary, the model can be retrained using data from the current release. In that way, the model can be adapted to the project and can generate more accurate results in later releases.

C. *Investigating different software systems*

In this experiment we investigate the detection capability of our approach if train and test software systems are different. We train our model using Xerces 2.7 and test it with JFreeChart 0.9.21 and ArgoUML 0.32. JFreeChart 0.9.21 contains 603 Normal and 40 God Class samples.

After an analysis we see that the model that was created using Xerces 2.7 finds 20 God Classes correctly, tags seven of God Classes as Normal Class and tags one Normal Class as God Class for the JFreeChart 0.9.21 project. On the other hand, for the ArgoUML 0.32 project, it finds 35 God Class correctly, tags five of God Classes as Normal Class and tags four of Normal Classes as God Class. The results are given in Table V and Table VI for the JFreeChart and ArgoUML projects respectively.

TABLE V. TRAIN: XERCES 2.7, TEST: JFREECHART 0.9.21

	Precision	*Recall*	*F-measure*
God Class	0.897	0.875	0.886
Normal Class	0.992	0.993	0.993
Weighted Average	0.986	0.986	0.986

TABLE VI. TRAIN: XERCES 2.7, TEST: ARGOUML 0.32

	Precision	*Recall*	*F-measure*
God Class	0.952	0.741	0.833
Normal Class	0.996	0.999	0.998
Weighted Average	0.995	0.996	0.995

The accuracy value is 98.6% for the JFreeChart 0.9.21 and 99.5 for the ArgoUML 0.32. The precision and recall values are greater than 89% and 74% respectively. These results encourage us to claim that decision tree model that is prepared with data from a software system can be used for detecting defects in other software systems. It is obvious that to support this idea more experiments on different software system should be conducted. Since, we here aim only to show the applicability of the approach we left additional experiments as future work.

VI. CONCLUSION

In this study, we propose the decision tree method for detecting the design defect that is called God Class in software systems. To validate our approach we investigated datasets that include object-oriented metrics of classes of open source software projects. Results from case studies verify our methodology and show that the God Classes in software systems can be successfully detected using decision trees that are created and trained with proper metrics. We validate our method in three different aspects to show its usability. First, we use some of classes of a program to train the model and then the method is applied to the rest of the same program to detect defects. Second, we create and train the decision tree with data from an earlier release of a software system and then use this model to detect defects in later releases of the same system. The model can also be retrained in some middle releases to adapt it to the characteristic of the software system. The third validation aspect is to train and to tests the model using totally different software systems. These experiments also gave reasonable results.

The proposed approach can reduce the maintenance cost by predicting the classes that are affected by God Classes and need refactoring. Decision tree is a supervised learning method and it selects the most significant metrics and their thresholds to predict God Classes according to training dataset. When we examined the models we saw that WMC, RFC, CBO, NOA, NOPA, ATFD, TCC, CAM were mostly selected metrics in our experiments.

In this study we trained the model to detect God Classes however the models can also be trained for detecting other types of bad smells using related metrics.As a future work, we want to apply this approach to predict other disharmonies like Data Class, Schizophrenic Class. In addition, to generalize the results we need to expand the datasets by investigating different software systems from several area and programming languages.

VII. REFERENCES

[1] S. Chidamber and C. Kemerer, “A Metrics Suite for Object-Oriented Design,” IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476- 493, June 1994.

[2] Bansiya, J., Davis, C., 2002: A Hierarchical Model for Object-Oriented Design Quality Assessment. IEEE Transaction on Software Engineering, vol. 28, no. 1, pp. 4-17, January.

[3] M. Fowler and K. Beck, "Bad Smells in Code", in Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2000, pp. 75-88.

[4] Marinescu, R. 2001: Detecting Design Flaws via Metrics in Object-Oriented Systems. TOOLS (39): 173-182.

[5] Marinescu, Radu, 2002: Measurement and Quality in Object-Oriented Design. Ph.D. Thesis, Department of Computer Science, "Politehinca" University of Timisora.

[6] Marinescu, Radu, 2004:. Detection Strategies: Metric Based Rules for Detecting Design Flaws. ICSM 2004: 350-359.

[7] Lanza, Michelle, Marinescu, Radu 2006: Object-Oriented Metrics in Practice. Springer.

[8] Li, Wei. Shatnawi, Raed (2007): An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. Journal of Systems and Software 80(7): 1120-1128.

[9] http://argouml.tigris.org/

[10] http://www.jfree.org/jfreechart/

[11] http://xerces.apache.org/index.html

[12] Erdemir, U., Tekin, U., Buzluca, F.. "E-Quality: A Graph Based Object Oriented Software Quality Visualization Tool". In Proceedings of the 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), pp..6–13, Virginia, USA Sep. 2011.

[13] Lanza, M., Marinescu, R., Object-Oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer, 2006.

[14] Drazin, S., and Montag, M., Decision Tree Analysis using Weka, University of Miami. from http://www.samdrazin.com/classes/een548/project2 report.pdf

[15] www.cs.waikato.ac.nz/~ml/weka/

[16] iPlasma: http://loose.upt.ro/iplasma/index.html

[17] 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 Object-Oriented Programming, Systems, Languages, and Applications, pages 47–56. ACM Press, 1999.

[18] Moha, N., Guéhéneuc, Y., Duchien, L., Le Meur, A., "DECOR: A Method for the Specification and Detection of Code and Design Smells", IEEE Transactions of Software Engineering, 2010.

[19] J. Ross Quinlan, C4.5: Programs for Machine Learning by. Morgan Kaufmann Publishers Inc., 1993

[20] Adhatrao, K., Gaykar, A., Dhawan, A., Jha, A., and Honrao V., "Predicting Students' Performance Using Id3 And C4.5 Classification Algorithms", International Journal of Data Mining & Knowledge Management Process (IJDKP) Vol.3, No.5, September 2013

[21] B. F. Webster. Pitfalls of Object Oriented Development. M & T Books, 1st edition, February 1995.

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

[23] 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.

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

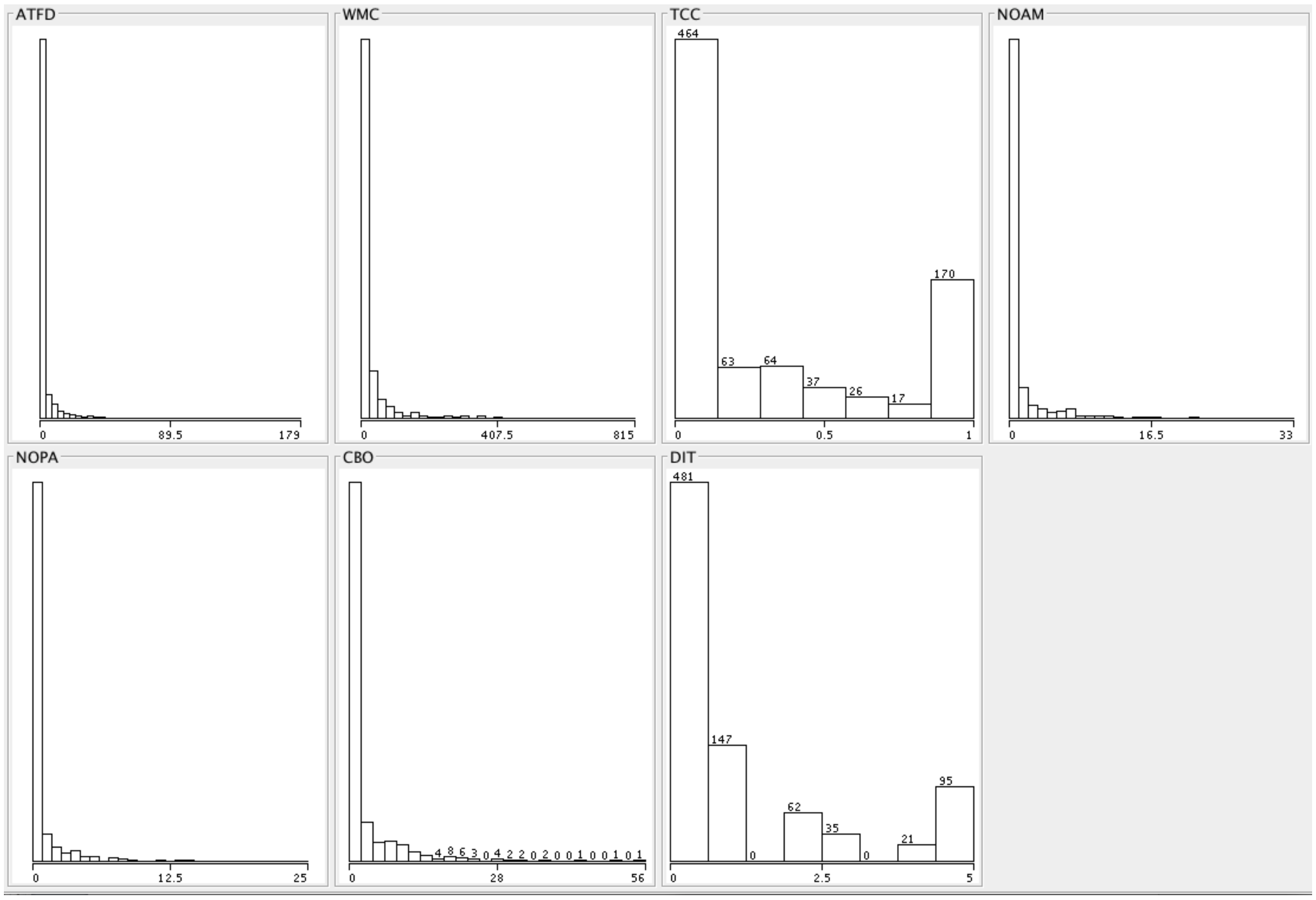

Figure 5. Metric distributions of the Xerces 2.7 (Part 1)

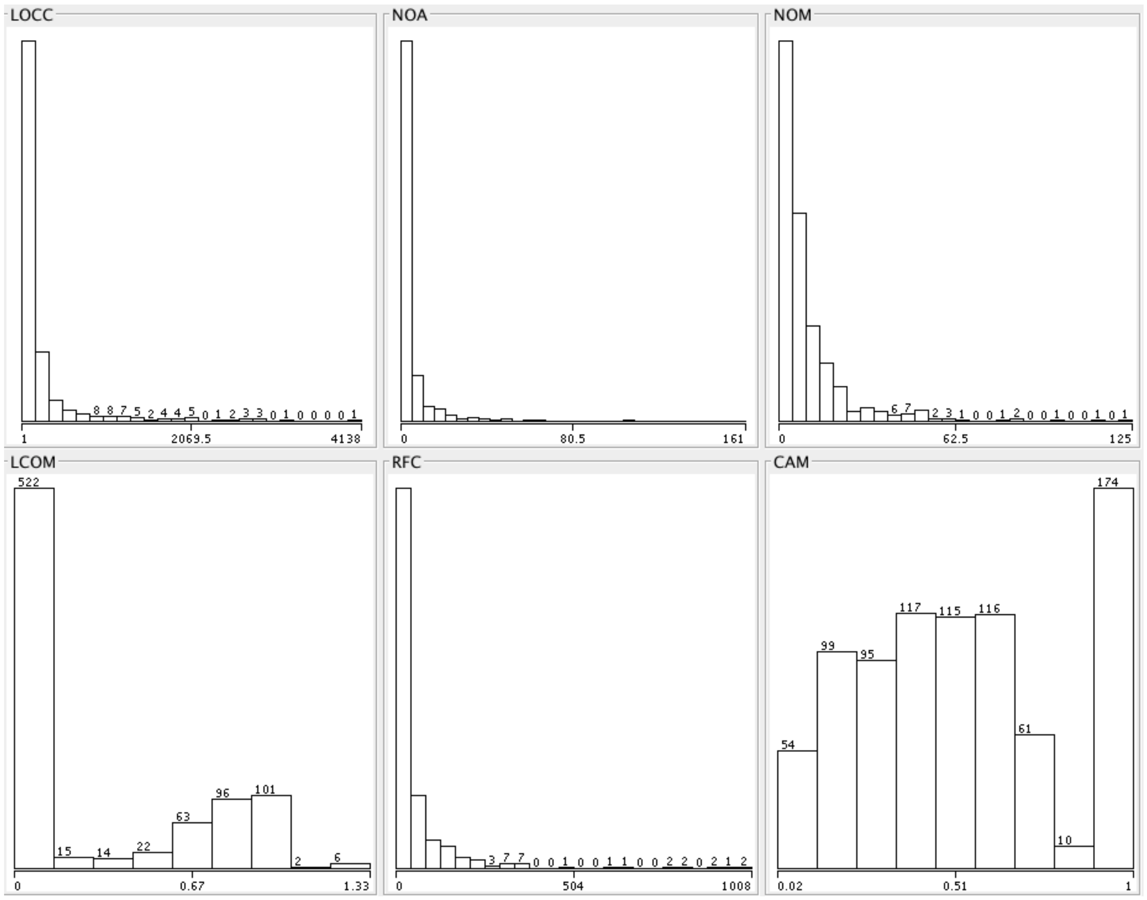

Figure 6. Metric distributions of the Xerces 2.7 (Part 2)