IJASRM

International Journal of Advanced Scientific Research and Management, Volume 3 Issue 11, Nov 2018

www.ijasrm.com

ISSN 2455-6378

Analysis of different approaches for code smell detection

S.James Benedict Felix[1] and Dr.Viji Vinod[2]

[1]Research Scholar, Bharathiar University,
Coimbatore, Tamil Nadu, 641 046, India.

[2]Professor & Head, Department of Computer Applications,
Dr.MGR Educational and Research Institute University,
Chennai, Tamil Nadu, 600 095, India.

Abstract

Bad smells are signs of latent problems in codes. Bad smells reduce the design value of software, so the codes are hard to analyze, understand, test or reprocess. Code-Smells represent design situations that can concern the maintenance and evolution of software. They make a system difficult to progress. Some code smells cannot be detected by using program analysis alone. In such cases, software metrics are adopted to help identify tools. However, the choice of suitable quality metrics is challenging due to the absence of accord to identify some code-smells based on a set of symptoms and also the high calibration effort to determine manually the thresholds value for each metric. In this paper, we discussed about detecting code smells using various approaches.

Keywords: *Machine Learning Approach, GUI-based approach, Textual Based Approach, A Bayesian Approach, Parallel Search Approach, Genetic Algorithm-based approach.*

1. Introduction

There exist many approaches to specify and detect code smells. Most of these approaches are manual or based on rules. Although these approaches improved the state of the art and of the practice in smell detection, to the best of our knowledge, none is able to handle the inherent uncertainty of the detection. Quality programs are easy to understandable, analyzable, modifiable, testable, maintainable and reusable. To the best of our knowledge, all previous approaches require expert knowledge and interpretation of the smell for their implementation. They focus on identifying one smell at a time, while some smells share similar characteristics, and exclude classes that are not identical to the smell (given some thresholds). Various technologies are used for detecting code smells.

2. The Benefits of a Code Refactoring

Code refactoring is the process of reforming the existing program code. Refactoring gets better nonfunctional attributes of the software and can improve source code maintainability. It improves code readability and decreases complexity. It creates an additional expressive internal architecture or object model to progress extensibility. There are two general categories of benefits to the activity of refactoring.

a) **Maintainability**. It is very easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp. This might be achieved by decreasing large monolithic routines into a set of individually concise, well-named, single-purpose methods. It may be accomplished by a class by moving a method or by removing misleading comments.

b) **Extensibility**. It is easier to expand the capabilities of the application if it uses identifiable design patterns, and it provides some flexibility where none may have existed before (M. Fowler, 1999).

3. Various approaches for Code Smell Detection

Many code smell detection tools have been developed providing different results, because smells can be subjectively interpreted, and hence detected, in different ways. Here, we discussed various approaches such as

- Machine learning approach
- GUI-based approach
- A Textual based Approach
- A Bayesian Approach
- Parallel Search Approach
- Genetic Algorithm based Approach

3.1 Machine Learning Approach

The application of machine learning to the code smell detection problem requires a formalization of the input and output of the learning algorithms, and a selection of the data to be analyzed and the algorithms to use in the experimentation. A huge set of object-oriented metrics, covering dissimilar aspects of software design, have been computed on a huge repository of heterogeneous software systems. A set of code smells to detect has been recognized, representing the dependent variables. For each and every code smell, a set of example instances have been manually assessed and labeled as correct or incorrect (affected or not by a code smell). The selection and labeling phase is an important role in machine learning techniques. Our approach selects the example instances by applying stratified random sampling on many projects, by the results of a set of pre-existing code smell detection tools and rules, called *Advisors*. This methodology guarantees a homogeneous selection of instances on different projects and prioritizes the labeling of instances with a higher chance of being affected by a code smell. The selected instances are used to train a set of machine learning algorithms, to execute experiments evaluating the performance of different algorithms and to search for the best setting of their parameters.

Some of the principal steps of Machine learning approach,

1. A Collection of a huge repository of heterogeneous software systems.
2. Extract a large set of object-oriented metrics from systems at class, method, package and project levels.
3. Choice of tools, or rules, for their detection; they are called *Advisors* in the following.
4. Application of the chosen Advisors on the systems, recording the results for each class and method.
5. Labeling: following the output of the Advisors, the reported code smell candidates are manually evaluated, and they are assigned different degrees of gravity.

The manual labeling is used to train supervised classifiers, whose performances (e.g., precision, recall, and learning curves) will be compared to find the best one (F. Arcelli Fontana et al, 2013).

3.2 GUI-based Approach

The aim of Bad smells detection is to address the software quality. For that purpose, the bad smell is to be defined and searched in the source code. Bad smells are commonly grouped into two types: (i) Internal and (ii) External.

Internal bad smells are obtained from the source code and provide information to improve software development. With our approach, we can get internal bad smells from source code through a reverse engineering process. Internal bad smells are structural characteristics of source code that may indicate a code or design problem. Internal bad smell has 22 different kinds of smells, being useful to enhance software's internal quality through refactoring process. Fowler specified different types of code smells, like:

Duplicated Code: means that the same code structure appears in more than one place;

Feature Envy: means that a method is in the wrong place since it is more tightly united to the other class than to the one where it is currently located;

God Class: It means class that tends to complete too much work;

Large Class: refers to classes that have too many instance variables or methods;

Considering different types of bad smells, we aim to detect them and discuss some of the relevant problems which we have to face for their automatic detection in interactive systems. To achieve that purpose adequate metrics must be specified and calculated.

External bad smells are defined in relation to running software. In concerns GUIs, external bad smells can be used as usability indicators. However, external bad smells are not accessible from source code analysis, rather through user's feedback (Brad A.Myers, 1999).

3.3 Textual Based Approach

The textual-based approach for detecting smells in the source code, coined as TACO (Textual Analysis for Code smell detection), has been instantiated for detecting the Long Method smell and has been evaluated on three Java open source projects. The results indicate that TACO is able to detect between 50% and 77% of the smell instances with a precision ranging between 63% and 67%. In addition, the results show that TACO identifies smells that are not identified by approaches based on the solely structural information. We evaluate the accuracy of TACO in detecting Long Method smell instances in three software systems, namely Apache Cassandra1, Apache Xerces2 and Eclipse Core3. Besides the analysis of the accuracy of TACO, we also compare the proposed approach with a structural-based technique, namely DECOR. In order to evaluate the accuracy of the experimented techniques, we compare the set of Long Method instances identified by a specific technique with the set of instances manually identified in the object system. Details on how these smells have been manually identified can be found in the paper by Palomba et al. Then, we measure the accuracy of the experimented techniques by using three widely-

adopted Information Retrieval (IR) metrics, namely recall, precision, and F-measure. In addition, we also measure the overlap between TACO and DECOR by measuring the smell instances identified by both the technique (TACO ∩ DECOR), the instances identified by TACO only (TACO \ DECOR) and the instances identified by DECOR only (DECOR \ TACO). The use of textual analysis is actually useful to avoid the identification of many false positive candidates, but also to detect instances of Long Method that the structural technique is not able to detect.

3.3.1 The proposed code smell detection process

Figure 1 depicts the main steps used by TACO in order to compute the probability of a code component being affected by a smell, which are (i) Textual Content Extractor, (ii) IR Normalization Process, and (iii) Smell Detector.

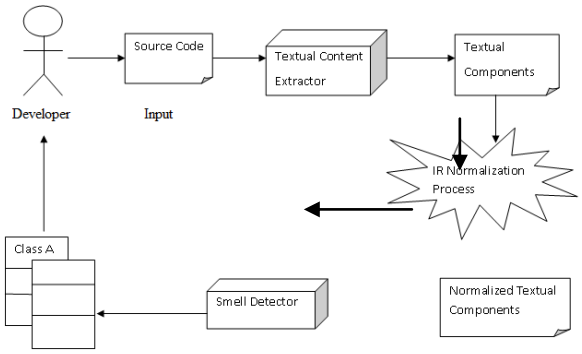

Fig 1: TACO: The proposed code smell detection process

Content (Textual) Extractor: Beginning from the set of code artifacts composing the software project under analysis. The first step is responsible for the extraction of the textual content characterizing each code component by selecting only the textual elements actually needed for the textual analysis process.

Information Retrieval Normalization Process: Comments and Identifiers of each component are firstly normalized by using a typical Information Retrieval (IR) normalization process. Therefore, the terms hold in the source code are transformed by applying the following steps: (i) separating composed identifiers using the camel case splitting which splits words based on capital letters, numerical digits and underscores; (ii) reducing to lower case letters of extracted words; (iii) eliminating special characters, programming keywords and common English stop words; (iv) stemming words to their original roots via Porter's stemmer. Then, the normalized words are weighted using the term frequency - inverse document frequency (tfidf) schema, which reduces the relevance of too generic words that are contained in most source components.

Then the normalized textual content of each code module is then separately evaluated by the Smell Detector, which applies various heuristics to recognize target smells. The detector relies on Latent Semantic Indexing (LSI), an extension of the Vector Space Model (VSM). LSI uses Singular Value Decomposition (SVD) to cluster code components according to the associations among words and among code components (co-occurrences). After that, the creative vectors (code components) are projected into a reduced k space of concepts to limit the effect of textual noise. For the choice of size of the decreased space (k) we used the heuristic proposed by Kuhn et al. that granted good quality results in many software engineering applications, i.e., k = (m_n) 0:2 where m indicates the size of vocabulary and n indicates the number of documents (code components in our case).

Finally, the textual similarity among software components is measured as the cosine of the angle between the corresponding vectors. The similarity values are then united in different ways, according to the type of smell we are interested in, to gain a probability that a code section is actually smelly. For detection purpose, we convert such a probability in a real value in the set ftrue, falseg to indicate whether a given code component is affected or not by a specific smell.

3.4 A Bayesian Approach

BBNs (Bayesian Beliefs Networks) have been successfully used to model uncertainty in fields as diverse as risk management, medicine, and computer science. We propose to use BBNs to specify and detect smells.

A Bayesian Belief Network is a directed, acyclic graph that represents a probability distribution. In this graph, a node is identified by random variable *Xi*. A directed edge between two nodes indicates a probabilistic dependency from the variable denoted by the parent node to that of the child. Thus, the structure of the network represents the every node *Xi* in the network is only conditionally dependent on its parents. Each node *Xi* in the network is associated with a conditional probability table that species the probability distribution of all of its probable values, for every possible combination of values of its parent nodes. A quality analyst requires two pieces of information to build a BBN: the structure of the network, in the form of arcs and nodes (causal relations), and the conditional probability tables describing the decision processes between each node. By structuring the network, the quality analyst makes ensure that the decision process is

valid. The conditional probabilities can be learned using historical data or entered directly by the analysts when data is missing. The structure make ensures the qualitative validity of the approach while proper conditional tables (learned or entered manually) ensure that the model is well calibrated and is quantitatively valid (N. Fenton and M. Neil, 2007).

3.4.1 Comparison with other Techniques

There are many techniques capable of modeling uncertainty. The two most popular groups of techniques are statistical models and machine learning models. Both groups rely on the availability of historical data to correctly predict a phenomenon with certainty. However, these types of models must be trained on large amounts of tagged data to be effective (each datum describing the inputs and correct outputs). In the context of smell detection, organizations rarely keep track of past detected smells and there is no public database containing instances of smells. Consequently, these techniques are not easily and directly applicable to smell detection. Furthermore, they use black-box processes not suitable for quality analysts who want to encode their knowledge in the process. BBNs can work with missing data and allow quality analysts to specify explicitly the decision process. When data is unavailable or must be adapted to a different context, an analyst can encode her judgment into the model. In the context of smell detection, this structuring is important because there are usually only a few instances of smells in a program; hence, a database of smell instances would be generally too small for other types of models while the literature contains many analysts' judgments on smells, which can be used to structure BBNs (R. G. Cowell et al, 2007).

3.5 Parallel Search Approach

There are some motivations behind the use of parallel computing for the design and implementation of parallel metaheuristics. Firstly, parallel search approach permits speeding up the search process by reducing the search time. Secondly, the getting solutions may be extensively improved. Cooperative metaheuristics have been established to explore the fitness landscape more efficiently on different problems such as the defect identification problem. This is recognized by portioning the search space and then swapping information between the various search methods which permits examining the search space proficiently. Thirdly, the uses of different metaheuristics concurrently in solving a particular problem decrease the sensitivity to the parameter values. Certainly, every search method would be launched with a special parameter value set which is dissimilar from the others ones. Thus, the search process would work according to several various parameter value sets which may augment the *robustness* of the obtained results. Finally, parallel distributed metaheuristics permits handle the problematic of scalability. Several problems actually involve a very large number of decision variables (called large-scale problems), a huge number of objectives (called many objective problems), a huge number of constraints (called highly constrained problems), etc. Parallel distributed metaheuristics can represent one possible remedy to tackle such problems. Designing parallel metaheuristics contain different existing models. It follows the following three hierarchical levels:

• ***An Algorithmic level*:** In this parallel model, independent or cooperating self-contained metaheuristics are used. It can be identified as a *problem-independent inter-algorithm parallelization*. If the different metaheuristics are independent, the search will be correspondent to the sequential execution of the metaheuristics in terms of the quality of solutions. However, the cooperative model will modify the behavior of the metaheuristics and enable the development of the quality of solutions.

• ***Iteration level*:** In this level, metaheuristic iterations are parallelized. It is *a problem-independent intra-algorithm parallelization*. The metaheuristic behavior is not modified. The main goal of this level is speeds up the algorithm by reducing the search time. In fact, the iteration cycle of metaheuristics on large neighborhoods for trajectory-based metaheuristics or large populations for population-based metaheuristics requires many computational resources.

• ***Solution level*:** In this model, the parallelization process manipulates a single solution from the search space. It is a *problem-dependent intra-algorithm parallelization*. In general, evaluating the objective function(s) or constraints for a particular generated solution is almost always the *most costly* operation in metaheuristics. In this model, the metaheuristic behavior is not altered. The main goal is the speed up of the search (W. Kessentini et al, 2014).

3.6 Genetic Algorithm Based Approach

Genetic Algorithms (GAs) are evolutionary algorithms motivated by the Darwinian theory of natural evolution. They simulate the progress of species emphasizing the law of survival of the nearly-best to resolve optimization problems. Thus, these algorithms start from a set of initial individuals (i.e. solutions), and to use naturally inspired evolution mechanisms to receive new and possibly enhanced solutions which gives the good approximation of the optimum for the problem under examination. The Genetic

Algorithms depends on the following three key factors: (i) an individual representation used to encode a answer to the problem; (ii) a fitness function which is a mean to evaluate the value of a given individual; and (iii) transform operators which are used to produce new nearest solutions starting from existing ones. Generally, GA proceeds using the prior elements, as follows: first it randomly generates an initial population, and then it executes crossovers and mutations on the fittest elements of this population until the chosen number of generation is reached. (Salim Kebir et al., 2016)

```
G_A(numofGenerations : Int) : Population
Begin
i = 0;
p = initialPopulation();
while i < numofGenerations do
        p' = SELECT(p);
        CROSSOVER(p0);
        MUTATE(p0);
        p'= p0;
        i =i + 1;
end while
return p;
End
```

Genetic Algorithm

3.6.1 Individuals

In our approach, individuals are composed of two elements:

- **Genotype**: The genotype which is an ordered variable-length sequence of refactoring including necessary parameters. When the sequence of refactoring is executed, it performs these changes and produces a modified version of the source code model.
- **Phenotype**: The phenotype is obtained by performing the sequence of refactoring to the initial source code model in the order that is given in the genotype.

This representation has the following key benefits. First, use of a source code model as a phenotype to present the component-based software design enables good computation of bad smells detection rules. Second, we give the possibility to the genotype to hold invalid refactoring.

3.6.2 Fitness function

This function is evaluated on an individual by (i) running the succession of refactoring operations hold in its genotype and (ii) evaluating the detection rules on the resulting source code model contained in its phenotype.

3.6.3 Change operators

In iteration, the Genetic Algorithm starts by choosing chromosomes. After that offspring is generated by applying crossover on each pair to produce two new chromosomes. Then, the mutation is applied to each chromosome in the current population with a given probability. The following three operators are used for implemented each of these.

- **Selection**: All of the population selected chromosomes will form a mating pool for the crossover and mutation.
- **Crossover**: In our approach, we adopt the one-point crossover operator which conceptually operates on two genotypes at a time and generates offspring by cutting each of the two parent chromosomes into two subsets of genes. Then two new chromosomes are created by interleaving the two subsets.
- **Mutation**: Our mutation operator either replaces a randomly chosen refactoring operation by a new one or randomly inserts/deletes a new refactoring operation to the genotype. A mutation is chosen by the user as an attribute of the GA.

4. Conclusion

Removing more number of bad smells is believed to improve the quality of the software. The above approaches are detected code smells using different methodologies. In this paper, we discussed various approaches in a different manner to find detection of code smells. Numbers of tools has been built for the detection of code smells and were validated on a different open source software system. Among all these approaches code smells are identified parallel manner using Parallel Search. In future work, we are planning to provide Genetic Algorithm and Particle Swarm Optimization based framework for code smell detection.

References

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

[2] F. Arcelli Fontana, M. Zanoni, A. Marino, M. V. 2013. Mantyla; "Code Smell Detection: Towards a Machine Learning-Based Approach". 29th IEEE International Conference on Software Maintenance (ICSM).

[3] Xiaodong Li, Xin Yao. 2012. Cooperatively coevolving particle swarms for large scale optimization. IEEE Transactions on Evolutionary Computation 16(2): 210–224.

[4] P. Siarry and Z. Michalewicz. 2008. Advances in Metaheuristics for Hard Optimization (Natural Computing Series). New York, NY, USA: Springer.

[5] Brad A.Myers. 1999. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-backs. School of Computer Science, Carnegie Mellon University.

[6] R. G. Cowell, R. J. Verrall, and Y. K. Yoon. 2007. Modeling operational risk with Bayesian networks. In Journal of Risk andInsurance. 74(4):795-827.

[7] N. Fenton and M. Neil. 2007. Managing Risk in the Modern World: Applications of Bayesian Networks. Technical report, London Mathematical Society.

[8] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni. 2014. "A cooperative parallel search-based software engineering approach for code-smells detection." IEEE Transactions on Software Engineering. pp. 841–861.

[9] Salim Kebir, Isabelle Borne, Djamel Meslati. 2016. "A Genetic Algorithm for Automated Refactoring of Component-Based software". Institute for Computer Science, Social-Informatics and Telecommunications Engineering. pp. 221-228.