RESEARCH-ARTICLE

Enhancing Defect Prediction with Static Defect Analysis

HAO TANG, Key Lab of High Confidence Software Technologies, Ministry of Education, Beijing, China

TIAN LAN, Key Lab of High Confidence Software Technologies, Ministry of Education, Beijing, China

DAN HAO, Key Lab of High Confidence Software Technologies, Ministry of Education, Beijing, China

LU ZHANG, Key Lab of High Confidence Software Technologies, Ministry of Education, Beijing, China

Enhancing Defect Prediction with Static Defect Analysis

Hao Tang, Tian Lan, Dan Hao, Lu Zhang
Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University,
Beijing, 100871, China
{townhall,1201214079,haodan,zhanglucs}@pku.edu.cn

ABSTRACT

In the software development process, how to develop better software at lower cost has been a major issue of concern. One way that helps is to find more defects as early as possible, on which defect prediction can provide effective guidance. The most popular defect prediction technique is to build defect prediction models based on machine learning. To improve the performance of defect prediction model, selecting appropriate features is critical. On the other hand, static analysis is usually used in defect detection. As static defect analyzers detects defects by matching some well-defined "defect patterns", its result is useful for locating defects. However, defect prediction and static defect analysis are supposed to be two parallel areas due to the differences in research motivation, solution and granularity.

In this paper, we present a possible approach to improve the performance of defect prediction with the help of static analysis techniques. Specifically, we present to extract features based on defect patterns from static defect analyzers to improve the performance of defect prediction models. Based on this approach, we implemented a defect prediction tool and set up experiments to measure the effect of the features.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics - Product metrics; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages - Program analysis

General Terms

Debugging

Keywords

Defect, predictive model, code feature, defect pattern, static defect analyzer, machine learning

1. INTRODUCTION

Internetware'15, November 06 2015, Wuhan, China.
©2015 ACM. ISBN 978-1-4503-3641-3/15/11 ...$15.00
DOI: http://dx.doi.org/10.1145/2875913.2875922

As a software paradigm for the Internet, Internetware supports the development, operation and quality assurance for software applications on the Internet. Software quality assurance is one of the challenges in the Internetware paradigm [23]. Since human activities rely increasingly on software applications, especially Internet applications, low-quality applications usually contains software defects which may bring immeasurable losses. Therefore, finding and fixing software defects is an important task for software quality assurance. Some quality assurance mechanisms can help developers to fight against some software defects in the applications. These mechanisms include software testing, verification, software defect prediction and static defect analysis.

Besides software quality assurance, software development needs to take both development progress and development cost into account. An unsolved software defect within the code may result in a "defect snowball", that is to say, the future development may need to deal with more and more induced defects. If developers can find defects as early as possible, the subsequent maintenance cost can be saved a lot. Periodical testing is considered a way to improve software quality, to assure development progress, and to save development cost. However, if the code is not sufficiently tested, software quality can be lowered by potential unsolved software defects. But excessive testing can also hinder development progress and raise development cost. A reasonable testing period requires developers to be aware of the current status of software quality. The number and the distribution of defects are indicators of software quality. A lot of empirical research work shows that the distribution of software defects follows Pareto's Law ("80/20 principle"). The law means that most defects locate in a small number of software modules [10]. Through software defect prediction techniques, it is easier for project managers, developers and testers to be concentrated on potential defective modules, so as to optimize resource assignment, improve work efficiency and assure software quality.

The software defect prediction area aims to improve the predicting performance in both precision and recall. Most defect prediction techniques build machine learning models about the relationship between code features and software defects. Selecting appropriate features is critical for performance improvement of machine learning models. For example, line of code (LOC) of a module is a code feature. Static defect analysis is another approach to find defects in the code so as to assure software quality. It uses pre-defined defect patterns to locate defects. For Java language, there are two mature static defect analyzers: Findbugs [15] and PMD [32]. A fraction of their built-in defect patterns are shown in Appendix A and B. For example, the first defect patterns `HE_INHERITS_EQUALS_USE_HASHCODE` in Appendix A indicates a defect-prone case, that is to say, a class inherits $equals(Object)$ from an abstract superclass, and $hashCode()$ from $java.lang.Object$. Therefore, the class is very likely to violate the invariant that equal objects must have equal hashcodes.[1]

The research purpose of this paper is to improve the performance of software defect prediction models. Our approach is combining the two approaches, namely defect prediction and static defect analysis, by adding static defect features to the original code feature set. More specifically, this paper covers these three aspects:

1. introducing software defect prediction and static defect analysis;
2. combing software defect prediction and static defect analysis techniques by extracting defect pattern features to improve the predictive performance;
3. implementing a tool and evaluating its performance on different feature sets and predictive models.

This paper continues as follows. Section 2 presents existing research related to ours. Section 3 presents our defect prediction algorithm using static defect analyzers. Section 4 is about the design and the implementation of our tool based on this algorithm. Section 5 is conclusion and future work.

2. RELATED WORK

To better understand our work, we discuss the current research progress of both software defect prediction and static defect analysis.

Software defect prediction. Since Akiyama [1] firstly addressed software defect prediction in 1971, it has still been one of the hottest topics in the software engineering area. On one hand, software defect prediction techniques comes in two flavors: static and dynamic, depending on whether code execution is needed. Static defect prediction mainly utilizes code features to predict defects. Dynamic defect prediction predicts defects based on the distribution of defects in different software life-cycle phases. On the other hand, software defect prediction has two research goals: defect density prediction and defect-prone module prediction. Defect density prediction is a typical regression problem which is usually solved by regression models. It aims to predict the number of defects in a module. Defect-prone module prediction uses binary classification models to classify a module into defect-prone or non-defect-prone one. It aims to identify defect-prone modules.

Our research work is based on static defect-prone module prediction. According to prediction models, there are two static prediction techniques: statistical prediction technique and machine learning prediction technique. In early years, researchers focused on extracting features about code size and complexity. Predictive models predict the number or distribution of potential defects by quantitatively analyzing the correlation of these features and code defects. With the prevalence of machine learning techniques, the above features are included in the training phase of machine learning.

Statistical prediction technique focuses on the code size [21] and the code complexity [13, 22, 35]. 1. Code size features include lines of code (LOC), lines of executable code, the number of documents, etc [1]; 2. Software volume features [13] is to identify measurable properties of software using the concepts such as volume and mass; 3. Lipow feature [21] is used to indicate the correlation between defect numbers and lines of executable code; 4. Takahashi feature [35] uses documentation information; 5. McCabe cyclomatic complexity feature [22] measures the complexity of control-flow graphs.

With the prevalence of object-oriented languages, researchers built object-oriented code features [5, 17]. Chidamber and Kemerer [5] proposed CK object-oriented features suite using object-oriented features such as inheritance and encapsulation. Empirical study shows that CK feature suite works well in predicting defects [17]. Besides CK suite, MOOD feature suite is another object-oriented feature suite [14]. Other types of features (e.g., code changing features [11]) can also improve the predicting performance a lot. Code-changing features assume that the more frequent the code changed, the code is more likely to contain defects. An empirical study [11] shows that recent modified modules are more likely to contain defects. Common code-changing features include line insertion, deletion and modification, and the number of commits, and the number of developers. More complex features include Code Churn features [8, 11, 25, 26, 27, 28, 29]. Rahman et al. [29] compared the code-changing features to simple code features and found the former perform better than the latter. Other features related to code runtime properties and software development also work well. Nagappan et al. [26] used Windows system data to predict defect. A lot of work uses bug reports (e.g. stack-trace) to guide software defect prediction and localization [37, 39].

During these years, machine learning techniques have become the main technology of the software defect prediction area [2, 3, 6, 12, 19, 24]. These techniques focus on mining knowledge and training data of the history projects to predict the defects in new projects. Empirical studies show that the predictive results of machine learning are better than that of code review [34, 36]. Software defect prediction usually utilize classification algorithms to predict defect-prone modules rather than regression algorithms to predict defect-prone density. Predicting defect-prone modules is a binary classification problem. The common techniques for this problem are naive bayes, logistic regression, decision tree, random forests, artificial neural network, support vector machine, and so on. Naive bayes is the simplest solution and sometimes performs better than complex algorithms [19, 24]. Logistic regression is also simple and it is utilized by a large amount of defect prediction work as the baseline to compare with other algorithms [19, 29, 31, 40]. Based on the union of decision trees, random forests are considered the best classifiers [16, 19]. Artificial neural network is an expressive model, but its training phase is time-consuming. Support vector machine performs well in processing high dimension data such as text classification and hand-writing recognition. Its model does not perform better than naive bayes and logistic regression in predicting software defects [19].

Static defect analysis. Developers often use static defect analyzers to check defects in the code so as to ensure software quality [33]. Different from software defect prediction, static defect analysis does not require the history data of the project. It uses predefined defect patterns to locate defects in the code. Static analysis can help developers to identify and tackle defects as early as possible without running the code, which can ensure software quality. Some existing automated static analysis tools are implemented to be used as self-checking tools before code review. Developers do not need to inspect source code details and learn specialized knowl-

[1]FindBugs Bug Descriptions:
http://findbugs.sourceforge.net/bugDescriptions.html

edge during self-checking phase.

Most static defects analyzers firstly analyze control-flow, data-flow, function calls of the code. Then, they use some pre-defined defect patterns to match the static analyzing results. These patterns usually include array out-of-bounds, uninitialized local variables, accessing uninitialized memory locations, null pointer exceptions, buffer overflow, memory leaking, double free error, dead locks and so on. Defect patterns can also be extracted by machine learning techniques. PR-Miner [20] identifies implicit programming rules by mining frequent item-set. The code violating these rules is considered relevant to software defects.

So far, there are hundreds of software static defect analyzers for a variety of programming languages and defect patterns. To deal with multiple programming languages, researchers tried to design a pervasive feature set. But it is proved to be a very difficult work. Cross-language software static defect analyzers can only deal with very limited types of defects. The common tools include: (1) *C*: Splint [4], Frama-C [7]; (2) *C++*: cppcheck [38], cpplint [9]; (3) *Java*: Findbugs [15], PMD [32]; (4) *Cross-language*: HP Fortify static code analyzer [2] (help developers to check security issues about C/C++, Java, JSP, and several other languages).

However, static defect analyzers have two major problems: (1) *low recall*: no software defect prediction tools are able to detect all defects in the code. Existing tools check the code against some specific defect patterns, but they are unable to recognize other types of defects. So the recall of these tools is limited. (2) *high false alarm*: some code snippets may be recognized as defects because they match some patterns, but these snippets are not true defects. Therefore, software defect prediction can have plenty of false alarms, which actually reduce the usability of the tool.

3. APPROACH

This section proposes a simple approach to enhance the performance of software defect prediction by static defect analysis. Our work aims to recognize whether a module is defect-prone or not.

Although static defect analysis and defect prediction vary in research purposes and techniques that they leverage, they can both find the possibly existing defects or point out whether a software module contain defects. Static defect analysis detects potential defects by matching some pre-defined defect patterns with the source code. Each of the potential defects usually corresponds to one or several lines of source code. Therefore, static defect analyzers support granularity level of lines. Defect prediction computes a set of features and utilizes the history data concerning about software defect distribution. By software defect models generated by statistical learning or machine learning, it is able to predict whether a software module is defect-prone or to predict the defect distribution. Defect prediction supports granularity level of methods, classes or files.

In recent years, researchers come to realize the similarity between static analysis and defect prediction. Rahman and Devanbu [30] proposed the AUCECL value compare static defect analyzers and defect prediction techniques. They also proposed to use the result of one technology to improve the result of the other by priority-reordering. They pointed out that the results of static defect analyzers can be improved by defect prediction, but defect prediction enhanced with static defect analysis does not work well. But Rahman and Devanbu [30] only prioritized the results of defect prediction by simply using the results of static defect analyzers, and they did not look deeply into the results of static defect analyzers. In this sense, we proposed to extract the effective features from the results of static defect analyzers, and we proposed to add these features into the original features from defect prediction in order to enhance the predicting performance.

[2]FORTIFY STATIC CODE ANALYZER: http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/

3.1 Machine-learning-based Defect Prediction

The basic work-flow of software defect prediction based on machine learning techniques is shown in Figure 1. Some important details of the work-flow include processing history data, building predictive models, and processing input data and making prediction:

- *Processing history data.* History data are transformed into the training set. The training set includes module features and labelled data. Our approach considers two feature sets: code features and static defect features. Defect reports indicate whether a module contains defects. We will explain how to use defect report as labelled data in Section 3.2.
- *Building predictive model.* An appropriate predictive model should be able to learn the training set completely and avoid overfitting. We should examine the properties of the training set carefully before predictive model selection. We will evaluate the models and training sets in Section 3.3.
- *Processing input data and making prediction.* The features and labelled data are processed from new versions or new projects. Selected predictive models input these data to produce predictive results.

3.2 Feature Engineering

The key to improve the performance of defect prediction by static analysis is to extract effective static defect features from static analyzing results. Specifically, two feature sets were collected in our approach: code features and static defect features.

Code features. Our approach use 14 code features: LCOM3, NPM, DAM, MOA, MFA, CAM, IC, CBM, AMC, CA, Ce, Max(CC), Avg(CC), LOC. They are shown in Table 1.

Static defect features. We focus on Java language. Our approach selects defect patterns from two well-known Java static checkers: Findbugs and PMD. Researchers have summarized numerous defect patterns in the two tools. Because of the excessive number of defect patterns, our approach only summarize defect patterns in the targeted projects and remove unfound patterns. (See Section 3.2 for the detail)

This paper proposes an approach by designing static defect features based on defect patterns. A static defect feature can be of different levels of granularity (one pattern, a type of patterns, or the whole set of patterns). The occurrence number of one pattern (or a type of patterns, or the whole set of patterns) is used as the value of the corresponding static defect feature.

If we treat each defect pattern in Findbugs and PMD as a static defect feature, the occurrence number for each found pattern is the corresponding feature value. The value is set to zero when the corresponding pattern is not found.

Assume that Findbugs and PMD contains six defect patterns (a, b, c, d, e, f) and the input project has four modules (A, B, C, D).

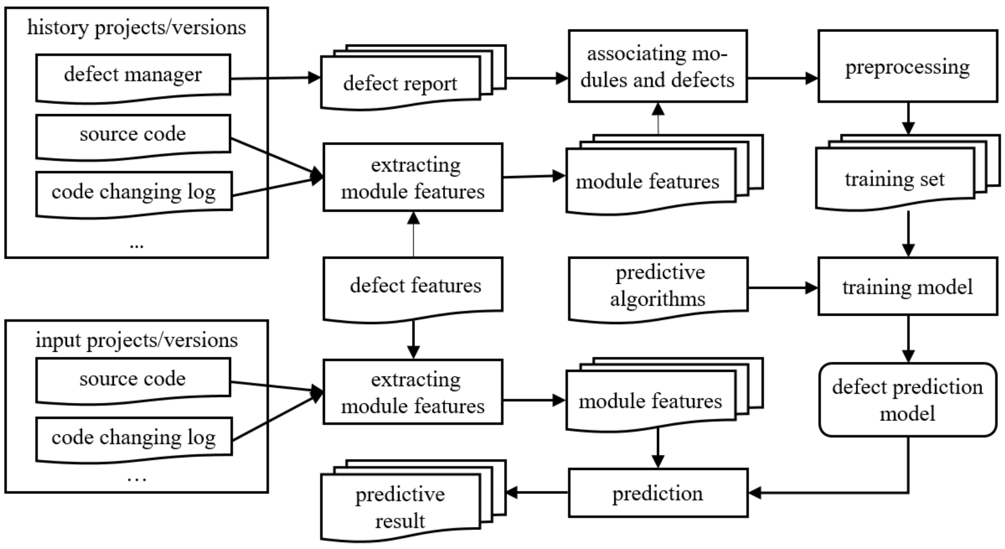


Figure 1: software defect prediction based on machine learning techniques

Applying static analysis to the input project derives the occurrence number of each defect pattern in each module, as shown in Figure 2.

However, some defect patterns (e.g., bad programming style and format) are not very relevant to the real defects, which may have a bad effect on the performance of the tool. One solution is to remove irrelevant defect patterns. For the example in Figure 2, assume that e and f are about bad programming style, and that only a, b, c, and d can be included in the feature set. The resulting feature matrix is shown in Table 3.

In these software defect prediction techniques, especially the machine learning based techniques, the more features do not mean the better. Too many features may cause overfitting of the training set which results in poor predictive performance. Feature selection is a key to tackle overfitting problem.

Feature combination is another approach other than feature selection. Feature combination combines multiple defect patterns into a static defect feature (the feature value is just the total occurrence number of these patterns). One way is to sort the patterns by their precision. Patterns can be divided into three groups: high precision, middle precision, and low precision. All patterns from static defect analyzers can even be regarded as one group.

In defect patterns a, b, c, d of Table 3, the precision of a and b is high and that of c and d is low, ab, cd. See Table 4.

In real applications, developers may select and combine some patterns from the whole set of the defect patterns to improve the performance of the predictive model.

3.3 Predictive Model Selection

Besides feature engineering mentioned in Section 3.2, defect prediction performance also depends the selection of predictive models. This paper considers two classification algorithms in the machine learning area as the predictive models. They are logistic regression and random forests.

- *Predictive Models Based on Logistical Regression.* Logistical regression is a classical machine learning algorithm. Due to its simplicity, efficiency, and good performance, it is the most commonly-used and basic software defect prediction technique, and it is usually used as the baseline [19, 29, 31, 40].
- *Predictive Models Based on Random Forests.* Random forests integrate the ability of multiple decision trees. It has good performance in the software defect prediction area, even better than support vector machine (SVM) and other supervised learning techniques [16, 19]. The random forests model first construct multiple decision trees (e.g., by ID3 algorithm) as weak classifiers. The random forests model combines these weak classifiers to a strong classifier. It has good performance such as high precision, fast learning, and reducing overfitting possibility.

4. IMPLEMENTATION

This section introduces the implementation of the defect prediction tool enhanced with static analysis. We show the whole architecture as well as its two modules.

4.1 Architecture

This tool consists of a feature extraction module (processing data) and a machine learning module (building predictive models). It is a subset of the work-flow shown in Figure 1.

4.2 Feature Extraction Module

The feature extraction module extracts both code features and static analysis features. Code features are extracted by ckjm[3] which is frequently used in the software defect prediction area. Static analysis features are extracted by Findbugs and PMD. The results are organized as feature matrices.

4.3 Machine Learning Module

The machine learning module of this tool calls APIs of Weka[4] (a data mining software in Java) in three phases:

[3]ckjm (Chidamber and Kemerer Java Metrics): http://www.spinellis.gr/sw/ckjm/

[4]Weka: http://www.cs.waikato.ac.nz/ml/weka/

Table 1: Code features for a class

Feature	Description
LCOM3	*Lack of Cohesion in Methods*. $LCOM3 = (m - \sum_p m_p)/a)/(m-1)$. m and a are the number of methods and that of properties, respectively. m_p is the number of methods using property p.
NPM	*Number of Public Methods*.
DAM	*Data Access Metric*. The ratio of the number of private and protected properties by the total number of properties.
MOA	*Measure of Aggregation*. The number of data declarations of which types are classes that are defined by developer.
MFA	*Measure of function Abstraction*. The ratio in the number of methods inherited by a class and the total number of methods accessible by members in the class.
CAM	*Cohesion Among Methods in Class*. The summation of the intersection of parameters of method with the maximum independent set of all types of parameter in class.
IC	*Inheritance Coupling*. The number of super classes coupled with this class. The coupling is through methods that are inherited from a parent class but not re-defined by its subclass.
CBM	*Coupling Between Methods*. The number of inherited methods.
AMC	*Average Method Complexity*. E.g., Java's bytecode length.
CA	*Afferent couplings*. The number of classes calling this class.
Ce	*Efferent couplings*. The number of classes called by this class.
Max (CC)	*Maximum McCabe*. The maximum McCabe of methods in this class.
Avg (CC)	*Average McCabe*. The average McCabe of methods in this class.
LOC	*Lines of Code*.

1. **Data input.** Before feeding data into Weka, the feature matrices from the previous module must be transformed into the ARFF data format used by Weka.

2. **Training predictive models.** This tool use *BaseRF.model* which is a normal random forest predictive model in Weka by default. The training set may include different history data (e.g. Ant or jEdit), which depends on the concrete experimental setting.

3. **Prediction.** The project data still needs to be transformed into the ARFF data format. The resulting report is in *csv* form. A snippet of the resulting report is shown in Figure 2. The first column contains the module names. The second column contains the defect-prone possibility of each module. The third column contains the classifying results. We set 0.5 as the default threshold value, that is to say, a module is classified as an defect-prone one if its defect-prone possibility is greater than 0.5.

Table 2: Static defect feature matrix for the all defect patterns

	a	b	c	d	e	f
A	4	1	2	0	1	0
B	2	1	0	1	1	0
C	1	3	1	1	0	1
D	0	2	1	0	0	0

Table 3: Static defect feature matrix for reduced defect patterns

	a	b	c	d
A	4	1	2	0
B	2	1	0	1
C	1	3	1	1
D	0	2	1	0

Table 4: Static defect feature matrix for combined defect patterns

	ab	cd
A	5	2
B	3	1
C	4	2
D	2	1

5. EXPERIMENTAL EVALUATIONS

To evaluate the performance improvement of our approach, we performed experiments on history data of open source projects.

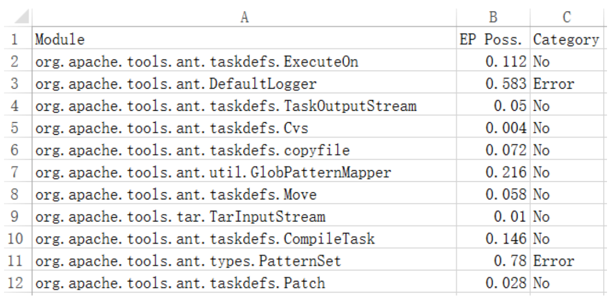
5.1 Experimental Setup

Our experiments used two Java open source projects: Apache Ant and jEdit. Ant is a software tool for automating software build processes. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java or non Java applications. We used five versions of Ant: ant-1.3, ant-1.4, ant-1.5, ant-1.6 and ant-1.7. jEdit is a cross platform text editor. jEdit-4.0, jEdit-4.1 and jEdit-4.2 are used in our experiments. Table 5 shows the basic information of these versions of projects.

In these experiments, we set the predictive granularity (i.e. a software module) as a Java class. A module is considered "defect-prone" if there is a defect report of this module within six months after code check-in. We obtained the defect report data from Marian Jureckzo Datasets [18] in the tera-PROMISE Repository [5]. The defect data sets of Ant and jEdit have the number of defects in each module.

Table 5 lists the total lines of code of each project (test source code excluded), the number of defect-prone modules, the number of all modules, and the radio of defect-prone modules to all modules. It is

[5]tera-PROMOSE: http://openscience.us/repo



	A	B	C
1	Module	EP Poss.	Category
2	org.apache.tools.ant.taskdefs.ExecuteOn	0.112	No
3	org.apache.tools.ant.DefaultLogger	0.583	Error
4	org.apache.tools.ant.taskdefs.TaskOutputStream	0.05	No
5	org.apache.tools.ant.taskdefs.Cvs	0.004	No
6	org.apache.tools.ant.taskdefs.copyfile	0.072	No
7	org.apache.tools.ant.util.GlobPatternMapper	0.216	No
8	org.apache.tools.ant.taskdefs.Move	0.058	No
9	org.apache.tools.tar.TarInputStream	0.01	No
10	org.apache.tools.ant.taskdefs.CompileTask	0.146	No
11	org.apache.tools.ant.types.PatternSet	0.78	Error
12	org.apache.tools.ant.taskdefs.Patch	0.028	No

Figure 2: A snippet of results

Table 5: Basic information of benchmarks

Project	**LOC**	**Defect-prone modules**	**Modules**	**Defect-prone radio**
ant-1.3	37699	20	125	16.00%
ant-1.4	54195	40	178	22.47%
ant-1.5	87047	32	293	10.92%
ant-1.6	113246	92	351	26.21%
ant-1.7	208653	166	745	22.28%
jEdit-4.0	144803	75	306	24.51%
jEdit-4.1	153087	79	312	25.32%
jEdit-4.2	170683	48	367	13.08%
Total		*552*	*2677*	*20.62%*

not hard to see that both Ant and jEdit have a considerable amount of code. ant-1.7, jEdit-4.1 and jEdit-4.2 all contain more than 150 KLOC. And there are totally 2677 modules and 552 defect-prone ones. Only 20.62% of modules are defect-prone in the all nine projects. The maximal ratio only reaches 26.21% (ant-1.6), while the minimal ratio even reaches 10.92% (ant-1.5). Obviously, imbalanced distribution of defect-prone and non-defect-prone module indicated by these data follows Pareto's Law ("80/20 principle") that we have mentioned above.

Feature engineering. As we mentioned in Section 3.2, we used two feature sets in our experiments, namely code features and static defect features. We have listed the code features in Table 1. Marian Jureckzo Datasets [18] also provide the code features of each module. Static defect features are the union set of all defect patterns summarized by checking all the versions of Ant and jEdit in the experiments using Findbugs and PMD. In our experiments, Findbugs detects 115 defect patterns, and PMD detects 184 defect patterns. See Appendix A and B for the details of these defect patterns. The non-reduced static defect features and the reduced static defect features are included in different experiments. Both of them are of Java class level granularity. The non-reduced static defect features form a feature vector. Each component of the vector indicates the number of occurrence of the corresponding defect pattern in a Java class. As we mention above, high dimensional features result in overfitting which may have bad effect on predictive performance. Our experiments reduced the whole features into two dimensions representing the total occurrence number of the all defect patterns from Findbugs and PMD, respectively.

5.2 Results

In this section, we investigate how much performance enhancement the defect predictive models gains from the newly-added static defect features.

We did within-version experiments and cross-version experiments, and we evaluated them on two predictive models (logistic regression and random forests). The within-version experiment was validated by 10-folds cross-validation. In the cross-version experiment, the last version is regarded as testing set, while the other versions are the training set, which we believe is close to real applications because developers may use such kind of tool to predict defects by older versions.

Within-version Experiments. Our within-version experiments were to use the data set from one version of a project. The data set was divided into a training set and a testing set. More specifically, the experiments used 10-folds cross-validation method which divided the data set into 10 folds. Each round of the cross-validation selected one fold from these 10 folds as the testing set and the others as the training set. After 10 rounds, all folds are selected as the testing set. Within-version experiments can validate the perform of the predictive model and give an insight on how the predictive model will generalize to a more realistic application.

Our within-version experiments used 5 versions of Ant (1.3, 1.4, 1.5, 1.6, 1.7) and 3 versions of jEdit (4.0, 4.1, 4.2) as experimental data. We built predictive models for each version using logistic regression and random forests, respectively. We used three sets of features: only code features, code features with reduced static defect features, and code features with non-reduced static defect features.

The results of within-version experiments are shown in Table 6 and Table 7. Table 6 is the results by logistic regression, and Table 7 is the results by random forests. As we can see, by adding reduced static defect features, the performance of predictive models improves a lot, which indicates that static defect features play an positive role. However, when using non-reduced static defect features, the performance of predictive models is unstable. That is because of overfitting caused by high dimensional features.

Cross-version experiments. Cross-version experiments use history project data as the training set and new project data as the testing set to evaluate the performance of predictive models. Although 10-folds validation is a common approach in the machine learning area, researchers and developers care more about the "future " in software engineering. Therefore, cross-version experiments are more close to applications in real situations.

We did 4 cross-version experiments shown in Table 8.

Table 8: Cross-version experiments setting

No.	*Predictive model*	*Training set*	*Testing set*
1	Logistic regression	ant-1.3, 1.4, 1.5, 1.6	ant-1.7
2	Logistic regression	jEdit-4.0, 4.1	jEdit-4.2
3	Random forests	ant-1.3, 1.4, 1.5, 1.6	ant-1.7
4	Random forests	jEdit-4.0, 4.1	jEdit-4.2

Cross-version experimental results are shown in Table 9. Table 9 shows the performance enhancement in cross-version defect prediction by adding the static defect features. Except for experiment 4 in Table 9, the other three experiments perform better with the static defect features added. Non-reduced static defect features have superior predictive performance in experiment 1 and 3, but their performance is unstable in experiment 2 and 4. That is because Ant has 4 versions which lowers the fitting ability of models, whereas jEdit only have 2 versions.

These experiments indicate the predictive performance using reduced and non-reduced static defect features, respectively. Reduced static defect features improve the predictive models in most experiments, except for a little degradation in some cases. Non-reduced static defect features have unstable perfomance. High dimensional features make the model overfitting. In summary, reduced static defect features have the best predictive performance.

6. CONCLUSION AND FUTURE WORK

Since 1970s, software defect prediction is one of the hottest topics of software engineering area. In this paper, we introduce the history and current research work on this topic, and then we introduce

Table 6: Within-version experimental results (by logistic regression)

Project Version	Defect-prone Rate	Non-reduced Static Defect Features			Reduced Static Defect Features			Code Features Only		
		Precision	Recall	F-Measure	Precision	Recall	F-Measure	Precision	Recall	F-Measure
ant-1.3	16.0%	38.1%	40.0%	0.39	47.1%	40.0%	**0.43**	35.7%	25.0%	0.29
ant-1.4	22.5%	32.8%	50.0%	**0.40**	28.6%	15.0%	0.20	28.6%	15.0%	0.20
ant-1.5	10.9%	26.2%	35.5%	**0.30**	37.5%	19.4%	0.26	30.8%	12.9%	0.18
ant-1.6	26.2%	37.2%	38.0%	0.38	58.7%	40.2%	**0.48**	54.4%	33.7%	0.42
ant-1.7	22.3%	42.2%	42.2%	0.42	69.0%	36.1%	**0.47**	69.0%	36.1%	**0.47**
jEdit-4.0	24.5%	40.0%	40.0%	0.40	65.9%	38.7%	**0.49**	66.7%	37.3%	0.48
jEdit-4.1	25.3%	39.8%	46.8%	0.43	66.1%	51.9%	0.58	68.9%	53.2%	**0.60**
jEdit-4.2	13.1%	25.3%	39.6%	0.31	67.9%	39.6%	**0.50**	61.5%	33.3%	0.43

Table 7: Within-version experimental results (by random forests)

Project Version	Defect-prone Rate	Non-reduced Static Defect Features			Reduced Static Defect Features			Code Features Only		
		Precision	Recall	F-Measure	Precision	Recall	F-Measure	Precision	Recall	F-Measure
ant-1.3	16.0%	0.0%	0.0%	0.00	50.0%	30.0%	**0.38**	41.7%	25.5%	0.31
ant-1.4	22.5%	40.0%	10.0%	0.16	33.3%	15.0%	0.20	38.9%	17.5%	**0.24**
ant-1.5	10.9%	100.0%	3.2%	0.06	43.8%	22.6%	**0.30**	33.3%	12.9%	0.19
ant-1.6	26.2%	68.1%	51.1%	0.58	68.8%	59.8%	**0.64**	67.1%	57.6%	0.62
ant-1.7	22.3%	72.0%	43.4%	0.54	68.1%	46.4%	**0.55**	65.8%	44.0%	0.53
jEdit-4.0	24.5%	65.2%	40.0%	0.50	64.0%	42.7%	**0.51**	60.4%	42.7%	0.50
jEdit-4.1	25.3%	73.1%	48.1%	0.58	72.2%	49.4%	**0.59**	67.3%	44.3%	0.53
jEdit-4.2	13.1%	73.3%	22.9%	0.35	65.4%	35.4%	**0.46**	63.0%	35.4%	0.45

Table 9: Cross-version experiment results

Experiment	Non-reduced Static Defect Features			Reduced Static Defect Features			Code Features Only		
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	Precision	Recall	F-Measure
1	43.0%	48.2%	**0.46**	59.6%	31.9%	0.42	72.2%	23.5%	0.36
2	27.7%	58.3%	0.38	44.6%	68.8%	**0.54**	45.3%	60.4%	0.52
3	65.5%	43.4%	**0.52**	61.4%	42.2%	0.50	68.2%	34.9%	0.46
4	41.3%	64.6%	**0.50**	40.5%	62.5%	0.49	41.7%	62.5%	**0.50**

static defect analysis. However, defect prediction and static defect analysis are still considered two parallel areas.

In this paper, we propose to combine static defect analysis with software defect prediction by enhancing the latter one with the techniques of the former. We further propose to add the static defect features to the code feature set. We implemented a tool to build predict models based on this proposal. We experimentally evaluate our approach on a number of history data sets of Java open source projects, and our experimental results demonstrate that the predictive model of our approach outperforms the one using only code features.

We believe that there will be a lot of future work on top of software defect prediction. In the near future, we plan to investigate new static defect analysis tools with higher precision features, and we plan to select and combine the defect patterns more carefully so as to improve the performance. We also plan to make our tool more flexible to train and predict on more kinds of predictive models.

Acknowledgements

This work was supported by the High-Tech Research and Development Program of China under Grant No.2015AA01A202 and the National Natural Science Foundation of China under Grant No.61 421091, 61529201, and 61272157.

7. REFERENCES

[1] F. Akiyama. An example of software system debugging. In *Proceedings of the International Federation of Information Processing Societies Congress*, volume 71, pages 353–359. New York: Springer Science and Business Media, 1971.

[2] C. Catal. Software fault prediction: A literature review and current trends. *Expert systems with applications*, 38(4):4626–4636, 2011.

[3] C. Catal and B. Diri. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.

[4] R. Chansler, R. Bryant, R. Bryant, R. Canino-Koening, F. Cesarini, E. Allman, K. Bostic, and T. Brown. The architecture of open source applications. 2011.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[6] B. Clark and D. Zubrow. How good is the software: a review of defect prediction techniques. *Software Engineering Symposium, IEEE Computer Press*, 2001.

[7] P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti. Experience report: Ocaml for an industrial-strength static analysis framework. In *ACM Sigplan Notices*, volume 44, pages 281–286. ACM, 2009.

[8] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *7th IEEE*

Working Conference on Mining Software Repositories, pages 31–41. IEEE, 2010.

[9] A. M. Dutko. *The Relational Database: A New Static Analysis Tool?* PhD thesis, Cleveland State University, 2011.

[10] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000.

[11] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[12] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[13] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[14] R. Harrison, S. J. Counsell, and R. V. Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, 24(6):491–496, 1998.

[15] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[16] Y. Jiang, B. Cukic, and T. Menzies. Fault prediction using early lifecycle data. In *The 18th IEEE International Symposium on Software Reliability*, pages 237–246. IEEE, 2007.

[17] M. Jureczko. Significance of different software metrics in defect prediction. *Software Engineering: An International Journal*, 1(1):86–95, 2011.

[18] M. Jureczko and L. Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 9. ACM, 2010.

[19] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[20] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 2005.

[21] M. Lipow. Number of faults per line of code. *IEEE Transactions on Software Engineering*, (4):437–439, 1982.

[22] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.

[23] H. Mei and X. Liu. Internetware: An emerging software paradigm for internet computing. *Journal of computer science and technology*, 26(4):588–599, 2011.

[24] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[25] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering*, pages 181–190. IEEE, 2008.

[26] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, pages 284–292. IEEE, 2005.

[27] T. Nakamura, L. Hochstein, and V. R. Basili. Identifying domain-specific defect classes using inspections and change history. In *Proceedings of the ACM/IEEE 28th international symposium on Empirical software engineering*, pages 346–355. ACM, 2006.

[28] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 86–96. ACM, 2004.

[29] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[30] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 424–434. ACM, 2014.

[31] F. Rahman, D. Posnett, and P. Devanbu. Recalling the imprecision of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 61. ACM, 2012.

[32] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *In Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 245–256. IEEE, 2004.

[33] C. Sadowski, J. van Gogh, C. Jaspan, E. Soederberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *Proceedings of the International Conference on Software Engineering*, 2015.

[34] F. Shull, V. Basili, B. Boehm, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, M. Zelkowitz, et al. What we have learned about fighting defects. In *Proceedings of 8th IEEE Symposium on Software Metrics*, pages 249–258. IEEE, 2002.

[35] M. Takahashi and Y. Kamayachi. An empirical study of a model for program error prediction. In *Proceedings of the 8th International Conference on Software Engineering*, pages 330–336. IEEE Computer Society Press, 1985.

[36] A. Tosun, A. B. Bener, and R. Kale. Ai-based software defect predictors: Applications and benefits in a case study. In *Proceedings of the 22th Innovative Applications of Artificial Intelligence Conference*, pages 1748–1755, 2010.

[37] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 181–190, 2014.

[38] D. Worth, C. Greenough, and L. Chin. A survey of c and c++ software tools for computational science. *Science and Technologies Facilities Council*, pages 1–38, 2009.

[39] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei. A survey on bug-report analysis. *SCIENCE CHINA Information Sciences*, 58(2):1–24, 2015.

[40] T. Zimmermann, R. Premraj, and A. Zeller. Predicting

defects for eclipse. In *International Workshop on Predictor Models in Software Engineering*, pages 9–9. IEEE, 2007.

APPENDIX

A. FINDBUGS DEFECT PATTERNS IN OUR EXPERIMENTS

BAD_PRACTICE: HE_INHERITS_EQUALS_USE_HASHCODE,OS_OPEN_STREAM,DE_MIGHT_IGNORE,RR_NOT_CHECKED,DM_EXIT,SR_NOT_CHECKED,NP_CLONE_COULD_RETURN_NULL,ES_COMPARING_PARAMETER_STRING_WITH_EQ,NP_TOSTRING_COULD_RETURN_NULL,NP_EQUALS_SHOULD_HANDLE_NULL_ARGUMENT,SE_COMPARATOR_SHOULD_BE_SERIALIZABLE,UI_INHERITANCE_UNSAFE_GETRESOURCE,NM_CLASS_NAMING_CONVENTION,RV_RETURN_VALUE_IGNORED_BAD_PRACTICE,NM_SAME_SIMPLE_NAME_AS_SUPERCLASS,CN_IMPLEMENTS_CLONE_BUT_NOT_CLONEABLE,SE_BAD_FIELD,NM_METHOD_NAMING_CONVENTION,EQ_COMPARETO_USE_OBJECT_EQUALS,SE_TRANSIENT_FIELD_NOT_RESTORED,BC_EQUALS_METHOD_SHOULD_WORK_FOR_ALL_OBJECTS,SE_NO_SERIALVERSIONID,SE_INNER_CLASS,ES_COMPARING_STRINGS_WITH_EQ,SE_BAD_FIELD_STORE,HE_EQUALS_USE_HASHCODE,CN_IDIOM,DMI_RANDOM_USED_ONLY_ONCE,CN_IDIOM_NO_SUPER_CALL
CORRECTNESS: RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE,MF_CLASS_MASKS_FIELD,EQ_SELF_USE_OBJECT,INT_BAD_COMPARISON_WITH_NONNEGATIVE_VALUE,SA_LOCAL_SELF_ASSIGNMENT_INSTEAD_OF_FIELD,EC_UNRELATED_TYPES,NM_VERY_CONFUSING,INT_BAD_COMPARISON_WITH_INT_VALUE,UWF_NULL_FIELD,UR_UNINIT_READ,SA_FIELD_SELF_ASSIGNMENT,RV_ABSOLUTE_VALUE_OF_RANDOM_INT,NP_UNWRITTEN_FIELD,RpC_REPEATED_CONDITIONAL_TEST,NP_NULL_ON_SOME_PATH_EXCEPTION,NP_ALWAYS_NULL,UWF_UNWRITTEN_FIELD,SA_LOCAL_SELF_COMPARISON,NP_NULL_ON_SOME_PATH,NP_NULL_PARAM_DEREF,NP_CLOSING_NULL
EXPERIMENTAL: OBL_UNSATISFIED_OBLIGATION_EXCEPTION_EDGE,OBL_UNSATISFIED_OBLIGATION
I18N : DM_DEFAULT_ENCODING
MALICIOUS_CODE:EI_EXPOSE_STATIC_REP2,FI_PUBLIC_SHOULD_BE_PROTECTED,EI_EXPOSE_REP2,MS_PKGPROTECT,DP_CREATE_CLASSLOADER_INSIDE_DO_PRIVILEGED,MS_EXPOSE_REP,MS_FINAL_PKGPROTECT,MS_CANNOT_BE_FINAL,EI_EXPOSE_REP,MS_OOI_PKGPROTECT,MS_SHOULD_BE_FINAL
MT_CORRECTNESS: LI_LAZY_INIT_STATIC,ESync_EMPTY_SYNC,DC_DOUBLECHECK,UG_SYNC_SET_UNSYNC_GET,IS2_INCONSISTENT_SYNC,UW_UNCOND_WAIT,WA_NOT_IN_LOOP,STCAL_STATIC_SIMPLE_DATE_FORMAT_INSTANCE,STCAL_INVOKE_ON_STATIC_DATE_FORMAT_INSTANCE,NN_NAKED_NOTIFY,LI_LAZY_INIT_UPDATE_STATIC
PERFORMANCE: DMI_BLOCKING_METHODS_ON_URL,SIC_INNER_SHOULD_BE_STATIC,DM_STRING_CTOR,UPM_UNCALLED_PRIVATE_METHOD,DM_STRING_VOID_CTOR,DMI_COLLECTION_OF_URLS,SS_SHOULD_BE_STATIC,URF_UNREAD_FIELD,DM_GC,WMI_WRONG_MAP_ITERATOR,DM_NUMBER_CTOR,UUF_UNUSED_FIELD,DM_BOOLEAN_CTOR,DM_BOXED_PRIMITIVE_TOSTRING,SBSC_USE_STRINGBUFFER_CONCATENATION,BX_BOXING_IMMEDIATELY_UNBOXED,DM_BOXED_PRIMITIVE_FOR_PARSING
STYLE: UWF_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD,RV_DONT_JUST_NULL_CHECK_READLINE,NP_LOAD_OF_KNOWN_NULL_VALUE,UCF_USELESS_CONTROL_FLOW_NEXT_LINE,RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE,BC_UNCONFIRMED_CAST,REC_CATCH_EXCEPTION,NP_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD,DMI_HARDCODED_ABSOLUTE_FILENAME,SF_SWITCH_NO_DEFAULT,ICAST_IDIV_CAST_TO_DOUBLE,DLS_DEAD_LOCAL_STORE_SHADOWS_FIELD,SF_SWITCH_FALLTHROUGH,UUF_UNUSED_PUBLIC_OR_PROTECTED_FIELD,DLS_DEAD_LOCAL_STORE,RCN_REDUNDANT_NULLCHECK_OF_NULL_VALUE,IM_BAD_CHECK_FOR_ODD,ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD,IM_AVERAGE_COMPUTATION_COULD_OVERFLOW,EQ_DOESNT_OVERRIDE_EQUALS,NP_DEREFERENCE_OF_READLINE_VALUE,BC_VACUOUS_INSTANCEOF,URF_UNREAD_PUBLIC_OR_PROTECTED_FIELD

B. PMD DEFECT PATTERNS IN OUR EXPERIMENTS

Basic Rules: CollapsibleIfStatements,EmptyIfStmt,ReturnFromFinallyBlock,AvoidMultipleUnaryOperators,JumbledIncrementer,EmptyWhileStmt,BooleanInstantiation,AvoidUsingOctalValues,UnnecessaryConversionTemporary,EmptySynchronizedBlock,BrokenNullCheck,EmptyStatementNotInLoop,UnnecessaryReturn,EmptyFinallyBlock,AvoidThreadGroup,EmptyCatchBlock,ForLoopShouldBeWhileLoop,UnnecessaryFinalModifier,MisplacedNullCheck,AvoidUsingHardCodedIP,UselessOverridingMethod,OverrideBothEqualsAndHashcode,UnconditionalIfStatement,EmptyStaticInitializer,EmptyInitializer
Braces Rules: WhileLoopsMustUseBraces,ForLoopsMustUseBraces,IfStmtsMustUseBraces,IfElseStmtsMustUseBraces
Clone Implementation Rules: ProperCloneImplementation,CloneThrowsCloneNotSupportedException
Code Size Rules: NcssMethodCount,ExcessiveParameterList,NcssConstructorCount,ExcessiveMethodLength,CyclomaticComplexity,TooManyFields,ExcessivePublicCount,NPathComplexity,TooManyMethods,NcssTypeCount,ExcessiveClassLength
Controversial Rules: DefaultPackage,AvoidUsingVolatile,BooleanInversion,DoNotCallGarbageCollectionExplicitly,AtLeastOneConstructor,UnnecessaryParentheses,AvoidFinalLocalVariable,CallSuperInConstructor,AssignmentInOperand,AvoidUsingShortType,UnnecessaryConstructor,DataflowAnomalyAnalysis,OnlyOneReturn,UnusedModifier,NullAssignment,DontImportSun
Coupling Rules: LooseCoupling,ExcessiveImports
Design Rules: ConfusingTernary,UseNotifyAllInsteadOfNotify,AvoidConstantsInterface,ConstructorCallsOverridableMethod,AvoidInstanceofChecksInCatchClause,EmptyMethodInAbstractClassShouldBeAbstract,NonThreadSafeSingleton,UncommentedEmptyMethod,SimplifyBooleanExpressions,UseSingleton,CompareObjectsWithEquals,PreserveStackTrace,UnsynchronizedStaticDateFormatter,TooFewBranchesForASwitchStatement,AbstractClassWithoutAbstractMethod,SingularField,UnnecessaryLocalBeforeReturn,NonCaseLabelInSwitchStatement,SimplifyConditional,AvoidReassigningParameters,ReturnEmptyArrayRatherThanNull,AvoidProtectedFieldInFinalClass,UseLocaleWithCaseConversions,SwitchStmtsShouldHaveDefault,NonStaticInitializer,MissingBreakInSwitch,AvoidDeeplyNestedIfStmts,AccessorClassGeneration,SwitchDensity,AvoidSynchronizedAtMethodLevel,OptimizableToArrayCall,FinalFieldCouldBeStatic,PositionLiteralsFirstInComparisons,ClassWithOnlyPrivateConstructorsShouldBeFinal,CloseResource,UseCollectionIsEmpty,SimpleDateFormatNeedsLocale,AssignmentToNonFinalStatic,SimplifyBooleanReturns,IdempotentOperations,UncommentedEmptyConstructor,ImmutableField
Finalizer Rules: FinalizeDoesNotCallSuperFinalize,FinalizeShouldBeProtected
Import Statement Rules: UnusedImports,DontImportJavaLang,DuplicateImports,ImportFromSamePackage
J2EE Rules: DoNotUseThreads,DoNotCallSystemExit,UseProperClassLoader
Java Logging Rules: MoreThanOneLogger,AvoidPrintStackTrace,SystemPrintln
JavaBean Rules: MissingSerialVersionUID,BeanMembersShouldSerialize
JUnit Rules: UseAssertSameInsteadOfAssertTrue,UseAssertNullInsteadOfAssertTrue,TestClassWithoutTestCases
Migration Rules: JUnit4TestShouldUseTestAnnotation,ReplaceEnumerationWithIterator,ShortInstantiation,ReplaceVectorWithList,IntegerInstantiation,JUnit4TestShouldUseBeforeAnnotation,ReplaceHashtableWithMap,LongInstantiation,ByteInstantiation
Naming Rules: ClassNamingConventions,BooleanGetMethodName,VariableNamingConventions,SuspiciousConstantFieldName,LongVariable,MethodNamingConventions,AvoidFieldNameMatchingMethodName,AvoidFieldNameMatchingTypeName,ShortMethodName,ShortVariable,NoPackage,AbstractNaming,SuspiciousEqualsMethodName
Optimization Rules: SimplifyStartsWith,AvoidArrayLoops,MethodArgumentCouldBeFinal,UseArrayListInsteadOfVector,UseArraysAsList,UseStringBufferForStringAppends,UnnecessaryWrapperObjectCreation,LocalVariableCouldBeFinal,AddEmptyString,AvoidInstantiatingObjectsInLoops
Security Code Guidelines: MethodReturnsInternalArray,ArrayIsStoredDirectly
Strict Exception Rules: AvoidCatchingNPE,DoNotThrowExceptionInFinally,SignatureDeclareThrowsException,AvoidCatchingThrowable,DoNotExtendJavaLangError,AvoidThrowingNullPointerException,AvoidRethrowingException,AvoidThrowingRawExceptionTypes,ExceptionAsFlowControl
String and StringBuffer Rules: ConsecutiveLiteralAppends,UnnecessaryCaseChange,InefficientStringBuffering,AvoidStringBufferField,StringInstantiation,UseStringBufferLength,StringToString,UselessStringValueOf,AvoidDuplicateLiterals,InefficientEmptyStringCheck,UseIndexOfChar,UseEqualsToCompareStrings,AppendCharacterWithChar,InsufficientStringBufferDeclaration
Type Resolution Rules: UnusedImports,CloneMethodMustImplementCloneable,SignatureDeclareThrowsException,LooseCoupling
Unused Code Rules: UnusedFormalParameter,UnusedLocalVariable,UnusedPrivateField,UnusedPrivateMethod