Improve the Performance of Mobile Applications based on Code Optimization Techniques using PMD and Android Lint

Man D. Nguyen[1], Thang Q. Huynh[2] and Hung T. Nguyen[2]

[1]International School, Duy Tan University, Vietnam
mannd@duytan.edu.vn
[2]School of Information and Communication Technology of HUST, Vietnam
thanghq@soict.hust.edu.vn, hungnt@soict.hust.edu.vn

Abstract. Analyzing, testing, and optimizing source code are techniques that improve software quality and the performance of features and energy consumption of systems. Source code analysis includes analyzing the source code of an application and checking aspects to detect potential problems based on previous experience. In this paper, we investigate the rules and techniques of analyzing and optimizing Java source code by using PMD and Android lint. An automatic code-analyzing and code refactoring tool is developed with a set of rules based on the Eclipse Refactoring API (plug-in) to get optimized code that consumes less energy and improves performance for Android applications. The optimized code was tested in real environments with positive results. It reveals that programmer could use these techniques and support tool for developing Android applications with high quality source code and reliable and performance.

Keywords: Static testing, code analyzing, optimized code, code refactoring, Android testing, PMD, Android lint.

1 Introduction

Analyzing and optimizing source code are techniques that improve the software quality and increase the performance of a system. Source code inspection and review are static testing techniques used for enhancing quality and reliability of applications. Code analysis includes reviewing and analyzing the source code in order to uncover the defects and potential problems within an application based on previous experience [1, 2, 3]. Static testing is a form of software testing that can be applied in each software development phase where the software has not been executed yet. It is generally not a detailed test because mainly checks for the sanity of the code, algorithms, or documents. A developer who wrote the code, in isolation, can use this type of testing to improve the quality of the code and product [22].

Energy efficiency can have a significant influence on user experience of mobile devices. Although energy is consumed by hardware, software optimization plays an important role in saving energy, so that the developers have to participate in the optimization process. The source code is the interface between the developer and hardware resources. According to Manotas and et al. [25] developers do not understand how the software engineering decisions they make affect the energy consumption of their applications and lack the tool support to help them make decisions or change their code to improve its energy usage. Developing energy efficient mobile applications is an important goal for the developers as energy usage can directly affect the usability of a mobile device. But developers lack guidance as to how to improve the energy efficiency of their implementation and which practices are most useful. There are existing studies and tools that can help developers to analyze, improve, and optimize their applications. Such as, Shuai Hao and et al. proposed program analyzes [24], and statistical based measurement techniques [27]. These techniques enable developers to understand where energy is consumed within an application; they do not provide guidance to improve the application's energy consumption. Xueliang Li and John P. Gallagher [28] proposed an energy optimization framework guided by a source code energy model. This technique enables developers to be aware of energy usage induced by the code and to apply very targeted source-level refactoring strategies. In previous our work [5], we presented the techniques to optimize the source code for Android mobile applications that are optimizing the process, shifting bits, using intermediate variables, removing redundant mathematical operations, processing loop, reusing objects, and refactoring source code, in order to increase the performance. At present, there are few techniques have been automated and supported by a variety of tools [6], [13] that increase developers' performance and lighten their burden. However, in fact, these techniques are not enough to address the gap between understanding where energy is consumed and understanding how the code can be changed to reduce the energy consumed.

In this research, we mainly utilize PMD[1], and Android lint to create a new set of rules and Eclipse plugins, which are used to analyze, optimize, and refactor Java source code, and thus, increase the performance as well as decrease the energy consumption of Android applications. PMD analyzes Java source-code based on the evaluative rules that have been enabled during a given execution. The tool comes with a default set of rules, which can be used to unearth common development mistakes, such as having empty try-catch blocks, unused variables, objects that are unnecessary, etc. PMD also allows users to execute custom analyzes by allowing them to develop new evaluative rules in a convenient manner [4], [14], [18]. The Android lint is a static code analysis tool that checks Android project source files for potential bugs and optimization improvements on correctness, security, performance, usability, accessibility, and internationalization [4], [15], [16].

The rest of the paper, section 2, we discuss the optimizing and analyzing techniques for Java source code, the strategy for building and enforcing the inspection rules and the introduction of a new set of rules. In section 3, the techniques along with all functions of Eclipse IDE's plugins are presented. Section 4 is test results on real

[1] PMD does not officially stand for anything, it has several unofficial names, the most appropriate probably being Programming Mistake Detector

devices. The discussion of related works is presented in section 5. Finally, the conclusion remarks, limitation, and future works are presented in section 6.

2 Java Source-Code Analyzing and Optimizing Techniques for Android Applications

PMD and Android lint are used to carry out the rules for code optimization and static code analysis. These rules are quite voluminous and detailed. There are general rules built for multiple languages and specific rules for Java only. Therefore, we realized that the richness of the optimized source code may cause conflicts and misunderstandings between the rules, such as optimizing speed may increase the size of memory. So that, what we need to solve in this paper is to optimize code in terms of energy consumption and performance improvement of the system. The contributions of this paper includes a set of new rules is built to reduce energy consumption and improve performance of the system based on a set of rules of PMD and Android lint; a tool to optimize source code for Android applications, which can automatically change source code through Eclipse Refactoring API [12].

2.1 Source Code Analyzing Rules

According to [8] and [14], different rules have an impact on different source code's elements, which are the result of transforming the original code into an Abstract Syntax Tree (AST) for Java and Document Object Model (DOM) for XML. Therefore, we use the following strategy for discovering the potential elements:

- Step 1: Analyze Java (*.java) and XML (*.xml) source code of project into AST and DOM, respectively.
- Step 2: With each achieved AST and DOM, we categorize the elements into groups with a common type. (E.g., element method calls, variable declaration)
- Step 3: Each rule in the ruleset is applied to the elements that it associates with. If an element fails to meet the requirement of any rules, we consider it as a potential element.
- Step 4: We collect the necessary information of potential elements and put in a list, which will be delivered to users later.

The set of rules may be dramatically various and whether the rules should be applied in a particular case or not, sometimes, it depends on the semantics of the program. Therefore, the potential elements should be listed on a certain list that programmers decide the application of the rules.

2.2 Source Code Altering Rules

After analyzing the source code to find the potential elements, altering source code without violating the rules is the next issue. Proposing source-code altering and fixing methods are among the required functions in the application that we have built. However, there is one thing to be noticed that not all the rules can easily change and fix the source code because of the semantics of the program. Therefore, fixing source code is developers' duty. The steps to fix the source code are as follows:

- Step 1: Pick a potential element from the list and all necessary information about the rules that it has violated.
- Step 2: Check if the element still violates the rule (using previously mentioned analyzing techniques such as AST). If the violation remains, or yet to be removed, there are two possible situations:

  (a) If the rule does not depend on the program's semantics, the source code will be changed automatically.
  (b) If the rule depends on the program's semantics, the programmer will change the code manually on the basic of the information provided by the rule.

- Step 3: Apply the source code alteration

The rules themselves have their own violation testing and re-adjusting process. By using this tool, we propose a general strategy that either automatically simplifies the re-adjustment, or suggests a solution to the developers.

2.3 Strategy for Applying the Rules

According to the rules of code analysis and alteration are as follows:

- Classify AST's elements into different groups from beginning to increase the performance of the program. Sometimes different rules can affect the same elements and therefore, these rules can reuse already to classify groups.
- The benefit of having AST's elements classified from the beginning is that each element goes through only one inspection process. Otherwise, if AST's elements classification takes place every time the rules are applied to one particular group, each element will go through another inspection process. That means the initial classification helps to achieve better performance.
- Higher compatibility after extension: When the number of Java source-code files or rules increases, the proposed strategy will still take effect without any change. Especially, the larger amount of rules, the higher probability of AST's elements used. That strengthens the effectiveness of the initial classification.
- Always make sure the potential elements still violate the rules prior to source code alternation.

2.4 The Establishment of Source Code analyzing and altering Rules

The rules are built based on [8],[11], and [14]. The rules are also included useful information into these rules for future needs, such as suggesting examples and methods of using them without any violation. These details provide users a deeper understand-

ing of the rules so that they can avoid breaking them without using other help from the application. The set of rules built based on groups of (1) decrease battery consumption(e.g. lower CPU Usage, brightness, color display and network processes), (2) general rules (which can be applied to different programming languages), (3) specification rules built only for Java and XML files for layout display, (4) rules that clarify the source code and simplify the maintenance process, and (5) rules that help decreasing other costs.
In this paper, the set of 49 rules was built for optimizing source code and decreasing battery consumption. These rules are listed in Table 1:

Table 1. Rules for optimizing source code and decreasing battery consumption

No	**Name of rule**	**Description**
1	FinalVariable	Add “final” attribute to the variable you do not want to change its value.
2	FieldShouldBeStatic	Variable with “final” attribute should be “static”
3	AlwaysReuseImmutable-ConstantObjectsForBetter-MemoryUtilization	Reuse constant objects if possible
4	UnusedVariable	Unused variable should be removed
5	UncalledPrivateMethod	Unused private method should be removed
6	ShouldBeStaticInnerClass	Inner class should be static if possible
7	AvoidUsingMathClass-MethodsOnConstant	Avoid using Math class methods on constant
8	ConstantExpression	Expression with constant value should be immediately calculated.
9	UseShiftOperators	The use of bit shifting operation is more recommended than multiplication and division of the power of 2.
10	UseShortCircuit-BooleanOperators	Use “||”, “&&” instead of “|”, “&”
11	UseStringLength-CompareEmptyString	Use “String.length()” instead of “String.equal()” when checking empty string
12	UseStringEqualsIgnoreCase	Use “String.equalsIgnoreCase()” instead of “String.equals()” when comparing string if possible
13	AvoidUnnecessarySubstring	Avoid using “String.substring()” if unnecessary
14	AvoidUsingStringCharAt	Avoid using “String.charAt()”if unnecessary
15	AvoidConcatenatingString-UsingPlusOperatorInLoop	Avoid using operator “+” of class “String” to concatenate string in loop
16	UseSingleQuotesWhen-ConcatenatingCharacter-ToString	Using single quote instead of a double quote when concatenating a character to the string.
17	AvoidUsingStringTokenizer	Avoid using “StringTokenizer”
18	BoxedPrimitiveJust-CallToString	A boxed primitive is allocated just to call toString(). It is more effective to just use the static form of toString which takes the primitive value.
19	AvoidConstructing-PrimitiveType	Avoid initializing objects of the box of the primitive data types via the operator "new"
20	AvoidInstantiation-ForGetClass	Avoid initializing the object just to call the method “getClass()”

21	AvoidBooleanArray	Avoid using an array of elements of "bool" type
22	AvoidEmptyLoops	Avoid using empty loops
23	AvoidEmptyStaticInitializer	static initialization block
24	AvoidEmptyTryBlocks	Avoid using empty “try” blocks
25	AvoidEmptyFinallyBlock	Avoid using empty “finally” blocks
26	AvoidEmptyCatchBlocks	Avoid using empty “catch” block
27	AvoidEmptySynchronized-Block	Avoid using empty “synchronized” blocks
28	AvoidEmptyIf	Avoid using empty If
29	AvoidUnnecessaryIf	Eliminate condition statement always or never occur
30	AvoidWriteByteMethod-InLoop	Avoid using “DataOutputStream.writeByte()” in loop
31	AvoidReadByteMethod-InLoop	Avoid using “DataInputStream.readByte()” in loop
32	AvoidMethodCallsInLoop	Avoid calls methods in the loop if possible
33	AvoidObjectInstantiation-InLoops	Avoid initializing objects in the loop
34	PlaceTryCatchOutOfLoop	Place “try/catch” outside of loop.
35	AvoidVectorElementAt-InsideLoop	Avoid using “Vector.elementAt()” inner of loop.
36	EnsureEfficientRemoval-OfElementsInCollection	Delete Elements of “Collection” inner loop efficiently
37	EnsureEfficientRemoval-OfMapEntries	Delete Elements of “Map” inner loop efficiently
38	EnsureEfficientIteration-OverMapEntries	Iterate the Elements of “Map” efficiently
39	UseEntrySetInsteadOf-KeySet	Using “Map.entrySet()” instead of “Map.keySet()”
40	AvoidPollingLoops	Avoid using “Thread.sleep()” inner loop
41	AvoidUsingThreadYield	Avoid using “Thread.yield()”
42	AvoidCreatingThread-WithoutRunMethod	Avoid initializing “Thread” if have no “implement” object and “run” method
43	AvoidSynchronized-BlocksInLoop	Avoid using “synchronized” block inner loop
44	AvoidSynchronized-MethodsInLoop	Avoid using “synchronized” method inner loop.
45	AvoidSynchronized-ModifierInMethod	Avoid using “synchronized” in the method
46	AvoidLinkedList	Avoid using “LinkedList” in necessary.
47	DefineInitialCapacities	Should predefine the size of “AbstractList”
48	UseToArrayWithArray-AsParameter	Using “Collection.toArray(Object[])” instead of “Collection.toArray()”
49	UseDataSourceInstead-OfDriverManager	Using “DataSource” instead of “DriverManager” to get “Connection”

Each rule in Table 1 is implemented by two classes. One is analyzer-class and the other one is checker-class. For example, the rule FinalVariable, FinalVariableAnalyzer class is used to analyze the source code of a program P. It calls FinalVariableChecker class to check local field or variable without the final keyword. If the variable or field does not change the value in its scope, the variable violates against the FinalVariable rule. Analyzing and checking the rule are based on analyzing AST tree of the program P, e.g.

```
//a segment of code of program P
 @SuppressLint("WrongCall")// To disable lint checking for a
specific Java class or method
  @Override
    public void run()
    {⊗  long ticksPS = 100;//=> variable should be final
    long startTime = 0;
    long sleepTime;
    //to do something  }
```

This warning will be marked in the source code that helps programmer locate and fix easily. The FinalVariableChecker also return detailed information of violated variable or field against the rule. The information is recorded in Problem list and Refactoring Unit for further tasks.

In generally, from the source code of the program, PMD creates the AST of the source file checked and executes each rule against that tree. AST is Abstract Syntax Tree which is a finite, labeled tree where nodes represent the operators and the edges represent the operands of the operators. The violations are collected and presented in a report. PMD executes the following steps when invoked from Eclipse:

1. The Eclipse PMD plugin passes a file name, a directory or a project to the core PMD engine. This engine then uses the rules as defined in package rule (the rules are configured in XML file of config folder of a project) to check the file(s) for violations.
2. PMD uses JavaCC to obtain a Java language parser and passes an Input Stream of the source file to the parser
3. The parser returns a reference of an Abstract Syntax Tree back to the PMD
4. PMD hands the AST off to the symbol table layer which builds scopes, finds declarations, and find usages
5. Each rule in the RuleSet gets to traverse the AST and check for violations
6. A list of RuleViolations is recorded in Problem List and Refactoring Unit for a report and refactor source code.

3 Develop a plug-in tool for Eclipse IDE

Code inspection techniques for Android plug-in tool have (1) Android applications' project analysis feature, which enables to analyze a project containing Java source code to figure out the potential problems and violated rules, (2) the re-adjustment of problematic elements allow developers to fix and solve the potential problems to achieve optimal source code, (3) information on violated rules allows developers to select one potential element and review the violation details, (4) eliminate the potential element on the list, and (5) the complement of new rules allow developers to add new rules based on their experience. The tool is divided into many modules to simplify the design and deployment. The architectural model of the application is shown in Fig. 1 [9, 10].The application has five main components:

- *Configuration Loader* enables loading and saving the configuration of optimizing rules. Further information about the rules is also included. The loading must be done before process takes place because this configuration will be applied in most of the functions.

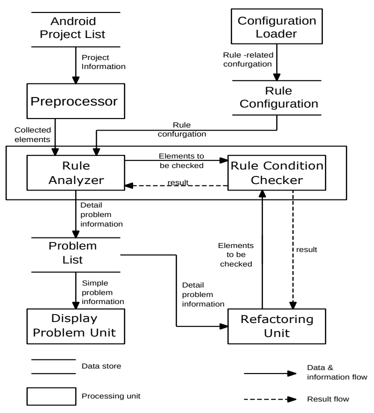


Fig. 1. The overview architectural model of the application

- *Preprocessor* collects useful information from Java source code, analyzes source file(s) from Eclipse project, transforms source file(s) into AST, and finally, classifies AST's elements into groups with the same type.
- *Analyzer* detects the potential elements in Java source code, retrieves the data from the preprocessor that have the same type of node group and manipulate them.
- *Display Problem Unit* displays all potential elements with problems found by the Analyzer. All information about these elements will be fully displayed so that developers will set the overview of the source code they are writing.
- *Refactoring Unit* provides users with a tool that supports source code alteration to solve and eliminate all potential problems; the result is a much better and optimal source code.

Analyzer component is one of the most important modules of the tool. It plays a vital role in identifying potential elements in Java source code. Each analysis will be based on a number of specific rules to analyze and detect errors. Each rule has an impact on a group or a node obtained from the certain preprocessor. The class diagram of Analyzer component is shown in Fig. 2a). The rule's information is described by RuleDescriptor and store in RuleDescriptionStore components.

The rule includes the following information: *Name of rule, Rule Identifying, Categorizing, Critical Levels of rule, Rule description, The reason for forming the rule, Examples and solutions, and Other information*.

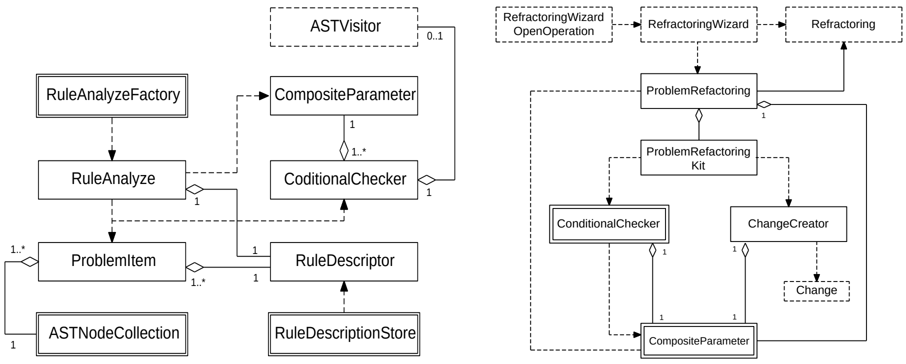
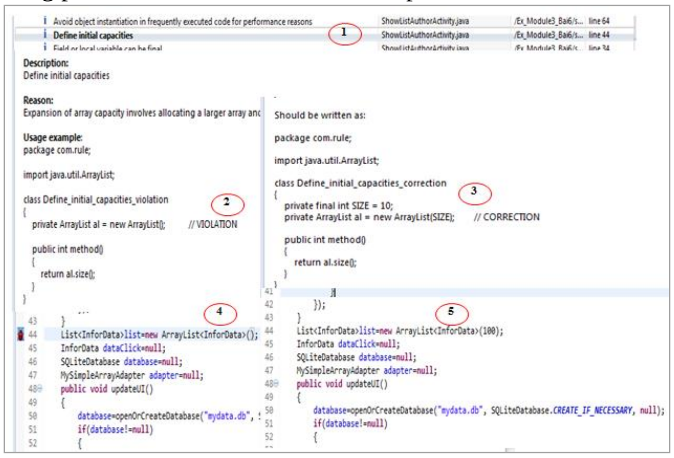


Fig. 2. Class diagram of a) Analyzer component, b) Refactoring Unit

ProblemItem is one of the potential elements detected by RuleAnalyzer. It contains a suite of RuleDescriptor, representing the violated rule and other information such as potential elements (ASTNode's derivative), locations (containing file, line…), and related information. Display Problem Unit is in charge of listing all problems detected in the source code. Refactoring Unit is responsible for refactoring the source code and supporting users in repairing and solving problems in order to achieve optimal source code. The class diagram of Refactoring Unit is shown in Fig. 2b). When a user chooses a problem from MarkingItem on Eclipse's user interface, all information of the problem will be obtained immediately. All information (Fig. 3 is an example) needed for the refactoring process will be included in CompositeParameter.



```
Avoid object instantiation in frequently executed code for performance reasons    ShowListAuthorActivity.java   /Ex_Module3_Bai6/s...   line 64
Define initial capacities                                                          ShowListAuthorActivity.java   /Ex_Module3_Bai6/s...   line 44
Field or local variable can be final                                               ShowListAuthorActivity.java   /Ex_Module3_Bai6/s...   line 34
```

Description:
Define initial capacities

Reason:
Expansion of array capacity involves allocating a larger array anc

Usage example:

```
package com.rule;

import java.util.ArrayList;

class Define_initial_capacities_violation
{
  private ArrayList al = new ArrayList();        // VIOLATION

  public int method()
  {
    return al.size();
  }
}
```

Should be written as:

```
package com.rule;

import java.util.ArrayList;

class Define_initial_capacities_correction
{
  private final int SIZE = 10;
  private ArrayList al = new ArrayList(SIZE);      // CORRECTION

  public int method()
  {
    return al.size();
  }
}
```

```
        ...
    }
    List<InforData>list=new ArrayList<InforData>();
    InforData dataClick=null;
    SQLiteDatabase database=null;
    MySimpleArrayAdapter adapter=null;
    public void updateUI()
    {
        database=openOrCreateDatabase("mydata.db", :
        if(database!=null)
        {
```

```
            }
        });
    }
    List<InforData>list=new ArrayList<InforData>(100);
    InforData dataClick=null;
    SQLiteDatabase database=null;
    MySimpleArrayAdapter adapter=null;
    public void updateUI()
    {
        database=openOrCreateDatabase("mydata.db", SQLiteDatabase.CREATE_IF_NECESSARY, null);
        if(database!=null)
        {
```

Fig. 3. Problem information

The dashed rectangle notations in Fig. 1, 2a), and 2b) are eclipse's components. The single border rectangles are classes or interfaces. The double border rectangles are significant classes that the tool was built. The dashed arrows indicate to reply the information.

4 Testing Result

In the scope of the paper, we set the goal of improving the quality of source code, decreasing the energy consumption of applications as well as enhancing the system performance. The set of rules are mentioned in Table 1 helps optimizing source code and decreasing battery consumption for Android phones. Testing result is collected from "Show CPU Usage" within Android's Developer Options. During the test, we collect the average percentage of CPU being used in the last 1-minute, 5 minutes and 10 minutes before and after optimization. Also, we conducted this test on 5 different applications and on the same devices with different settings. Fig. 4 and Fig. 5 display test results from two different devices respectively (*A and B in Fig.4 and Fig.5 mean After and Before optimization*). One is Motorola moto X: Android 4.4.4, OpenGL ES 2.0, Kernel Architecture i686, Kernel version 3.10.0-genymotion, screen size 4.59 inches, resolution 720x1280 pixels, screen density 320dpi, total RAM 2020MB, storage 4.84GB. And the other is HTC One XL: Android 4.2.2, OpenGL ES 2.0, Kernel Architecture i686, kernel 3.4.67-qenu+, screen size 4.59 inches, screen resolution 720x1280 pixels, screen density 320dpi, total RAM 1006 MB, storage 4.78GB.

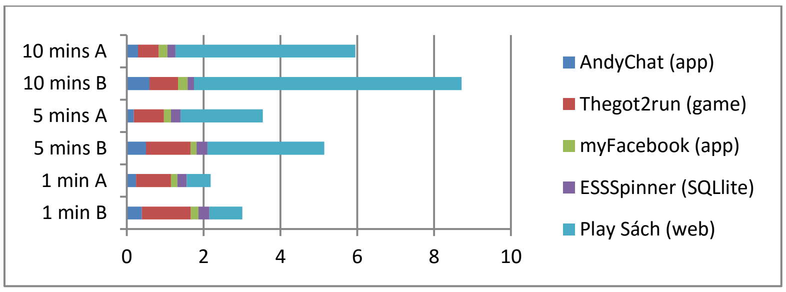


Fig. 4. Statistics of CPU Usage (%) testing on Motorola Moto X

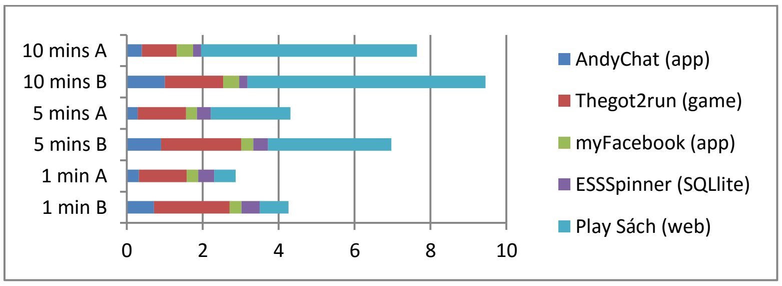


Fig. 5. Statistics of CPU Usage (%) testing on HTC One XL

According to Fig. 4 and Fig. 5, the use of source-code optimizing and refactoring techniques obviously enhances the performance of not only the application but also the device (the percentage of CPU used has dropped). The percentage of CPU used in 1 minute has dropped significantly after the optimal source code. This value will stabilize after 10 minutes, and in close proximity to the value before optimization.

Table 2. - Statistics of CPU Usage (%) testing on HTC One XL for each rule

Application	Rule	Before optimization	After optimization
Estate	AvoidEmptyIf	0.28/0.22/0.14	0.27/0.19/0.14
Estate	AvoidMethodCallsInLoop	0.34/0.13/0.12	0.08/0.07 /0.10
Who is millionaire	UseStringLength-CompareEmptyString	0.51/0.17/0.08	0.09/0.10 /0.08
Estate	Avoid object instantiation in frequently executed code	0.38/0.12/0.08	0.32/0.11 /0.07
Book management	A constant expression can be evaluated	0.43/0.12/0.08	0.11/0.09 /0.07
Book management	DefineInitialCapacities	0.14/0.09/0.06	0.0 /0.04 /0.05
Who is millionaire	AvoidSynchronized-MethodsInLoop	0.22/0.07/0.06	0.01/0.08 /0.07

(0.39 /0.5/0.59: the average % of CPU being used in the last 1-minute, 5 minutes and 10 minutes)

Table 2 indicates the CPU is used for each rule to be invoked. Depending on the specific rule that the reduction rate after 1 minute of CPU usage is very high, such as AvoidMethodCallsInLoop, UseStringLength CompareEmptyString, DefineInitialCapacities, and so on. Corresponding to each program, all the rules in Table 1 are implemented to find the violations of the program that can help programmers optimize source code to improve performance for applications. Furthermore, if more than one installed applications are optimized, the decrease in battery consumption is much more significant.

5 Discussion

In summary, the contributions of this paper are as following. We take the approach to check the quality of code and build the set of rules based on PMD and Android lint rules. The rules focus on the reduction of energy consumption and improvement of performance of the system. We also build the tool that combines the features of PMD, Android lint and Eclipse Refactoring API (introduced in Section 3). Although this tool is built for specific purposes, it can be developed for many other purposes by creating a custom set of rules. For the similar tools, CheckStyle [19] is a development tool help programmers writing Java code that adheres to a coding standard. It parses Java code and informs user all the errors found in accordance to the configuration provided in checkstyle.xml and the suppressions.xml file. The performed checks mainly limit themselves to the presentation and not analyze the content, as well as the correctness or completeness of the program. It may be useful to determine which level of check is needed for a certain type of program. FindBugs [20] uses static analysis to inspect Java bytecode for occurrences of bug patterns. It detects common error such as wrong Boolean operator. It also detects the error due to the misunderstanding of language features, such as reassignment of parameters in Java. The goal is globally the same CheckStyle, to find the patterns, which can lead to bugs using static analysis.

However, it can be a lot more controversial than FindBugs. The basic ruleset is always executed alongside with the custom ruleset. Android Lint provides great piece of advice to improve the quality of code. The user can use a separate lint file to define which rules to use or not and activate all the exacted rules. Android Lint always tests all the rules except the one whose the “severity” level is “ignore”. So if new rules are released with new version of ADT, they will be examined without ignorance. Although each tool has its own advantages and disadvantages, our tool is built by combining the features of PMD and Android lint for a specific purpose. Our future work is research on combining into a framework for improving the quality of code and increasing the performance of mobile applications for many different purposes.

6 Conclusion

In this paper, we studied several types of research and utilized plenty of Java source-code optimization techniques. We studied and built the set of rules and developed a plug-in, which automates the analysis, finds the potential problems and refactors the code. The quality of code is sharply improved and the battery consumption is dramatically decreased. The result is very useful for developers in agile development process, which supposedly required developers’ acquisition of software tester’s skills. The tool helps developers to analyze and to reveal problems quickly, effectively doing the refactoring process, and easily sharing the result in the group. Moreover, with our own method, engineers are able to extend and enrich the rules (based on their demand) to fulfill different needs in a much simple, effective, and reliable way.

The future work includes adding more rules to enrich the capabilities of the tool, as well as applying more rules for refactoring and supporting automatically. Currently, the tool can refactor potential problems on each component through the Eclipse interface. In the future, the tool can support automated refactoring potential problems across multiple components simultaneously. Further evaluation of usages and application for developing different types of mobile applications are also necessary. The long-term future work includes making the tool into a framework by combining the advanced features of Checkstyle, FindBugs, PMD, and Android lint to check the quality of code and performance improvement of the system.

References

1. Fagan, Michael: Design and code inspections to reduce errors in program development. Software pioneers. Springer Berlin Heidelberg, pp.575-607(2002)
2. Ackerman, A. Frank, Priscilla J. Fowler, and Robert G. Ebenau: Software inspections and the industrial production of software. Proc. of a symposium on Software validation: inspection-testing-verification-alternatives. Elsevier North-Holland, Inc., (1984).
3. Ackerman, A. Frank, Lynne S. Buchwald, and Frank H. Lewski: Software inspections: an effective verification process. IEEE software 3, pp. 31-36 (1989)
4. Gomes, Ivo, et al.: An overview on the static code analysis approach in software development. Faculdade de Engenharia da Universidade do Porto, Portugal (2009).

5. Huynh Quyet Thang, Nguyen Duc-Man, Do Le Nam.: Some Optimization Techniques And Performance Testing For Android Application Development. Proceedings of the 7th National Conference FAIR'7), ISBN:978-604-913-300-8, P.365-375 (2014)
6. Wedyan, Fadi, Dalal Alrmuny, and James M. Bieman.: The effectiveness of automated static analysis tools for fault detection and refactoring prediction. Software Testing Verification and Validation. ICST'09. International Conference on. IEEE (2009)
7. Soares, Gustavo, et al.: Generating unit tests for checking refactoring safety. Brazilian Symposium on Programming Languages (2009)
8. Bauer, Lujo, et al.: Mobile SCALe: Rules and Analysis for Secure Java and Android Coding. No. CMU/SEI-2013-TR-015. Carnegie-Mellon Univ Pittsburgh Pa SEI (2013)
9. Blewitt, Alex.: Eclipse 4 Plug-in Development by Example Beginner's Guide. Packt Publishing Ltd (2013)
10. Aeschlimann, Martin.: JDT fundamentals Become a JDT tool smith. Eclipse CON (2008)
11. Sharkey, Jeff.: Coding for life–battery life, that is. Google IO Developer Conference (2009)
12. Petito, Michael.: Eclipse refactoring, http://people. clarkson. edu/~ dhou/courses/EE564-s07/Eclipse-Refactoring.pdf
13. Frenzel, Leif.: The Language Toolkit: an API for automated refactorings in Eclipse-based IDEs. Eclipse Magazin 5 (2006)
14. How to write a PMD rule, http://pmd.sourceforge.net/snapshot/-howtowritearule.html
15. Android Lint, http://tools.android.com/tips/lint
16. Lint, http://developer.android.com/tools/help/lint.html
17. Improving Your Code with lint, http://developer.android.com/tools/-debugging/improving-w-lint.html
18. Christopher, Ciera Nicole.: Evaluating Static Analysis Frameworks. Analysis of Software Artifacts, pp.1-17 (2006)
19. Checkstyle tool, http://checkstyle.sourceforge.net/
20. FindBugs tool FindBugs™ - Find Bugs in Java Programs. http://findbugs.sourceforge.net/
21. Dawson, Maurice, et al.: Integrating Software Assurance into the Software Development Life Cycle. Journal of Information Systems Technology and Planning 3, no. 6 (2010).
22. Bourque, Pierre, and Richard E. Fairley.: Guide to the software engineering body of knowledge(SWEBOK(R)): Version3.0. Chap.10. IEEE Computer Society Press (2014).
23. Li, Xueliang, and John P. Gallagher.: A Source-level Energy Optimization Framework for Mobile Applications. arXiv preprint arXiv:1608.05248 (2016)
24. Hao, Shuai, Ding Li, William GJ Halfond, and Ramesh Govindan.: Estimating mobile application energy consumption using program analysis. In 2013 35th International Conference on Software Engineering (ICSE), pp. 92-101. IEEE (2013)
25. Manotas, Irene, Lori Pollock, and James Clause.: SEEDS: a software engineer's energy-optimization decision support framework. In Proceedings of the 36th International Conference on Software Engineering, pp. 503-514. ACM (2014)
26. Li, Ding, and William GJ Halfond.: An investigation into energy-saving programming practices for android smartphone app development. InProceedings of the 3rd International Workshop on Green and Sustainable Software, pp. 46-53. ACM(2014.)
27. Li, Ding, Shuai Hao, William GJ Halfond, and Ramesh Govindan.: Calculating source line level energy information for android applications.In Proceedings of the 2013 International Symposium on Software Testing and Analysis, pp. 78-89. ACM (2013.)
28. Li, Xueliang, and John P. Gallagher.: A top-to-bottom view: Energy analysis for mobile application source code. arXiv preprint arXiv:1510.04165 (2015).