Rule-Based Techniques Using Abstract Syntax Tree for Code Optimization and Secure Programming in Java

Nguyen Hung-Cuong(✉), Huynh Quyet-Thang, and Tru Ba-Vuong

Department of Software Engineering, School of Information and Communication Technology, Hanoi University of Science and Technology, Hanoi, Vietnam
{cuongnh86,trubavuong}@gmail.com, thanghq@soict.hust.edu.vn

Abstract. Although the quality of computer software consists of many different aspects, the security and the optimization are by far the most important metrics for estimating quality of software systems. The security ensures that application will work correctly and the optimization reduces the amount of resources needed: computation, memory, size of code, etc. Those techniques can be done by applying rules in abstract syntax tree, a tree representation of the abstract syntactic structure of source code. However, the process to optimize code often makes negative effect to the security of program. This work studies about applying rules in abstract syntax tree in Java and its effect on code optimization and secure programming problems.

Keywords: Abstract syntax tree · Code optimization · Secure programming

1 Introduction and Motivation

Nowadays, computer science appears in every aspects of our life, from house to office, from industry to entertainment. One of the most important requirements when building a computer software is the reliability of system: the high-reliability system ensures that work of users will be done accurately, in different contexts and different environments. Authors use dissimilar approaches to introduce new techniques to improve the security of software and those techniques are used widely: in C/C++ [1], in UNIX [2], in Java [3,4], etc. However, the increasing of complexity system makes more many challenges that have to be overcome.

The development of software technology makes more and more compound system, so arise demand that application should be optimized. Optimization is the progress of transforming software source code to make more efficient without changing its works. In more specific, code optimization is a machine-independent optimizations that can be done in source code of project or corresponding diagram to reduces the amount of resources needed: computation, memory, size of code, etc. on many fields of computer science: compiler [5], pipeline constraints [6], embedded processors [7], etc. Many optimization problems are NP-complete and thus most optimization algorithms depend on heuristics and approximations techniques.

Normally, not good optimization technique makes some negative effects to the quality of software: developer can not control the corresponding changing between modules before and after transforming. In addition, any technique have their own application in specific context: one technique is good for this context but can make uncontrolled problems in another one. This work is going to study about the secure programming and code optimization techniques that apply rules in abstract syntax tree in Java. So give results that in this technique, code optimization process does not reduce the reliability of system.

As a important attribute of software quality, software reliability is influenced strongly by software lifecycle [8]. In the software lifecycle of a commercial applications, 50 % of errors introduced and errors detected are in coding phase, so secure programming and code optimization techniques can affect significantly to the reliability of system.

In this study, we show result about the mutuality between secure programming and code optimization in Java when applying rules in abstract syntax tree. This paper is organised as follow: after this introduction section, Sect. 2 explains definition of abstract syntax tree and some regular problems of secure programming and code optimization. Note that those problems are in common programming language, not with any specific one. Next, Sect. 3 introduces rules and how to apply rules in abstract syntax tree. Section 4 shows some experimental results, including running environment, application design and running results. Then Sect. 5 discusses some related and future problem to extend current work.

2 Basic Techniques of Java-Application Development

As mentioned earlier, in this paper we study how to improve the quality of system. This section introduces two problems of software development: code optimization and secure programming.

2.1 Code Optimization

When size of system is raised rapidly, researchers focus on optimizing the source code to save the resource needed. Aho consider [5] code optimization as a sub problem of compiler technique. Some regular techniques in code optimization will be discussed following.

1. Use length property when check emptiness of string. For example, replace

```
public boolean isEmpty(String str){
   return str.equals("");}
```

   by

```
public boolean isEmpty(String str{
   return str.length()==0;}
```

2. Do not use constructor of class Integer:

   ```
   public void fubar(){
      Integer i = new Integer(3);}
   ```

   by

   ```
   public void fubar(){
      Integer i = Integer.valueOf(3);}
   ```

Today, several compiler can optimize source code in some specific context and it is called *compile level*. Notwithstanding, code optimization should be done by programmer because it is to complex to be executed automatically. And main trade off when optimize manually is the working-cost.

2.2 Secure Programming

Secure programming relates with fault problems: detection, tolerance, back up, etc. Target of developer is building a secure system, in this work of users will be done correctly. However, it is a difficult challenges to overcome in current state of science and technology. Some regular secure problems will be discussed following.

1. Variables are set *private*

   ```
   class Person{
      String name;
      int age;}
   ```

   by

   ```
   class Person{
      private String name;
      private int age;}
   ```

2. Let object uncloneable

   ```
   class Person{
      /*do not use clone()*/}
   ```

   by

   ```
   class Person{
   public final void clone()
     throws java.lang.CloneNotSupportedException{
        throw new java.lang.CloneNotSupportedException();}}
   ```

Many research show that most faults are made when some programming mistakes are mined, so developer need to be trained about those attacks.

2.3 Abstract Syntax Tree

Abstract syntax tree (AST), or just simply syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language: each node of the tree denotes a construct occurring in the source code. Aho et al [5] introduce some syntax trees of basic elements as follow: each construct is represented by a node that children of it emblem the semantic meaningful components of this construct.

1. Represent blocks by AST: consider block as a single statement, AST of blocks is simply the syntax tree for the set of order statements. Then replace symbolic node in this super-AST that represent a block by corresponding sub-AST, so that link of super-AST points to root of sub-AST.
2. An AST for a statement. Operator of statement need to be defined for construction of its AST: operator of a constructs that begin with a keyword is this keyword.
   - An operator `while` for while-statements.
   - An operator `do` for do-while-statements.
   - Conditionals can be handled by defining two operators `ifelse` and `if` for if-statements with and without an else part, respectively.
3. An AST for an expression: an interior node represents an operator and corresponding operands are symbolized by children of it.

An example of AST for Euclid function:

```
int Euclid{
  while b != 0
     if a > b
        a > b
        a := a - b;
     else
        b := b - a;
  return a;}
```

is given in Fig. 1.

3 Rules and Applying in Abstract Syntax Tree

Rules are principles that modify source code to improve quality of work. They have experimental meaning and are got from practician in computer science. They contain some important information: description; “how, why and where” to apply; etc. They have to be applied on given elements in specific context, so deploying rule in real development requires combining with AST-transformation of source code.

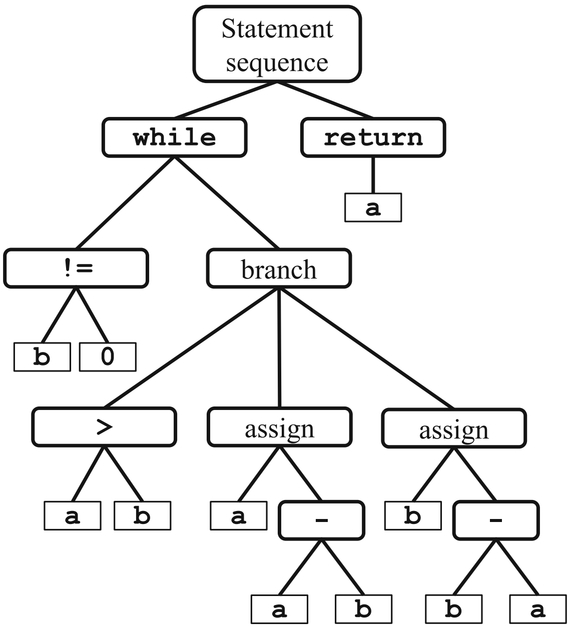

Fig. 1. Abstract syntax tree of Euclide function

3.1 Building Rules

From secure programming and code optimization problems that are discussed before, developer should synthesis a set of rules to using after. However, those techniques are used for general context and developer have to focus on techniques that adapt with Java. Furthermore, he need refer rules from secure source: from framework manufacturer, experiment developer, expert, etc... In this study, Java is a programming framework that is discussed, and then we focus on resources from Oracle, Google or AppPerfect. When apply rules in abstract syntax tree in Java to resolve secure programming and code optimization problems, developer should store them to reuse or gain experience to future work. In addition, programmer should save all information about rules that help user to lookup when needed.

3.2 Using Rules to Detect Potential Elements in Source Code

As discussed before, each rule impact to different elements of source code, and those elements are result of transform process to get AST from source code. So following 4-step strategy is used to detect potential elements:

Step 1. Transform every source code files in Java project into abstract syntax tree.
Step 2. Divide elements of each abstract syntax tree into groups, base on property of element: method calling, variable declaration, etc..

Step 3. Apply each rule to elements that are consistent: if a element is invalid with any rule, it is a potential element.
Step 4. List every potential elements and collect information that relates with each potential element. This list is supplied to developer.

Because size of set of rule is unpredicted and the application of rule on specific context is depend on semantic meaning of program, then potential elements should be listed and provided to programmer to decide apply or not.

3.3 Using Rules to Modify Source Code

After detecting potential elements, the next part is modifying source code so that rules are not violated. Developer has to decide what and how to transform by following scenario:

Step 1. Select one potential element that violates rules and all of its information.
Step 2. Check its violation again. If it is still invalid, there are two cases:
  - If this rule does not depend on program semantically, source code is changed automatically.
  - If this rule depends on program semantically, plug-in suggest developer all information to decide modify source code or not.

Step 3. Modify source code.

4 Experimental Results

4.1 Environment Descriptions

System computing performance depends on CPU Intel(R) Core(TM) i5 M520 2.40 GHz with memory RAM 4 GB. Operating system is Windows 7 Professional 64-bit and IDE is Java 6, 32-bit.

Our result is appreciated by using Java VisualVM, a tool that provides a visual interface for viewing detailed information about Java applications while they are running on a Java Virtual Machine.

Application is a Eclipse plug-in and developed by Plug-in Development Environment that supplied by Eclipse. It bases on Eclipse Juno (4.2) SR2, package "Eclipse for RCP and RAP Developers". Plug-in uses three tools of Eclipse Platform:

1. PDE (Plug-in Development Environment): providing tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites and RCP products.
2. JDT (Java Development Tools): providing the tool plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins. It allows to access, create and modify Java projects in Eclipse.
3. Eclipse Refactoring API (ERAPI): is a part of Language Toolkit (LTK) of Eclipse and is installed in two plug-ins: "org.eclipse.ltk.core.refactoring" and "org.eclipse.ltk.ui.refactoring".

4.2 Eclipse Plug-In Tool Descriptions

Product of our project is a Eclipse plug-in tool that:

+ Improve quality of application.
+ Make good programming behaviours.

Functions of tool:

1. Searching and analysing potential elements that can be impacted by code optimization and secure programming techniques.
2. Supporting method to modify those elements by showing supported information and recommending suggestions.

Data flow chart of plug-in is in Fig. 2. Our plug-in contain five modules:

1. Configuration Loader: this module saves and loads all information of rules in XML file. This configuration has to be loaded firstly and store through begin to end of session of Eclipse.
2. Preprocessor: transform every file in Java source code into abstract syntax tree and divide into groups.
3. Analyzer is one of the most important of system. It is used to detect potential elements in AST and store information of rules, including name, identity, type, priority, etc...
4. Display Problem Unit: show information about potential elements.
5. Refactoring Unit: tool that support to optimize source code.

4.3 Experimental Results

Result is got after comparing performance of system before and after using code optimization and secure programming techniques. Table 1 shows that running time of methods decrease clearly when applying code optimization and secure programming techniques. The largest slowdown is seen on method `getTopicDetailsTask()` with 33.53 % and this improvement shows that quality of source code is improved markedly.

Note. BKProfile is a intelligent web service that support to communication between lecturers, company, student and ex-student through personal profile. It is built in the form of question-answer system: users share their knowledge through making question or answering of another.

Second evaluation of this study is about system resource usage: memory and CPU. Those appreciations are got from running main functions of BKProfile system. Figures 3 and 4 show comparison of system resource using before and after optimization. Figures 3 indicates that after apply optimization techniques will decreasing CPU usage, both of maximum using and average. Figures 4 display statistical heap-using. Heap stores every objects that are created by Java in running time and is evaluated by two measures: heap-size and used-heap. In Fig. 4, we can see that there are two improvements of using memory:

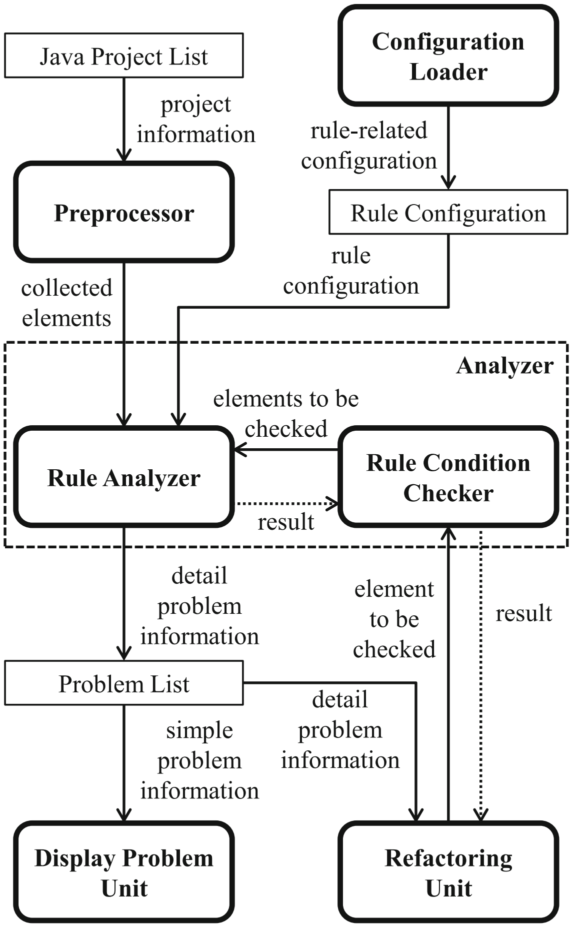

Fig. 2. Data flow chart of Eclipse plug-in tool

Table 1. Running time of methods

ID	**Program**	**Method**	**Times**	**Running time(ms)**		**Improve(%)**
				Before	**After**	
1-1	QuickSort	generateArray	1	2703	2671	1.18
1-2	QuickSort	sort	1	36558	25362	30.63
2-1	BKProfile	getTopStatsTask	10000	114	107	6.14
2-2	BKProfile	getTopicDetailsTask	10000	68	45.2	33.53
2-3	BKProfile	getAnswersTask	20000	1005	951	5.37
2-4	BKProfile	getQuestionDetailTask	20000	135	132	2.22
2-5	BKProfile	getSimilarQuestionsTask	20000	210	188	10.48
2-6	BKProfile	getFollowersTask (1)	20000	266	204	23.31
2-7	BKProfile	getFollowersTask (2)	20000	143	113	20.98

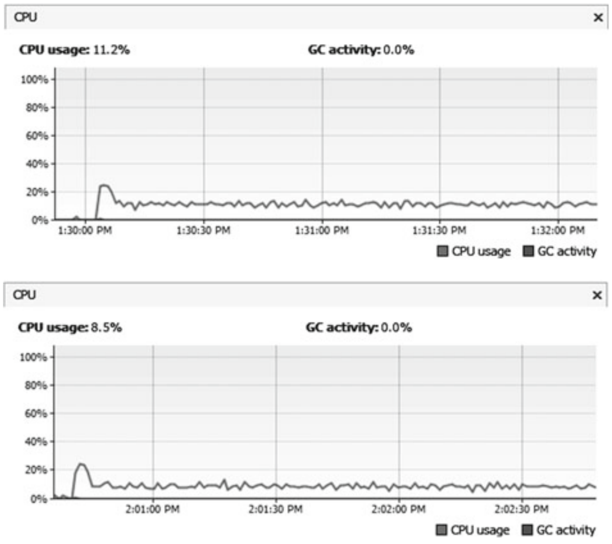

Fig. 3. CPU Usage

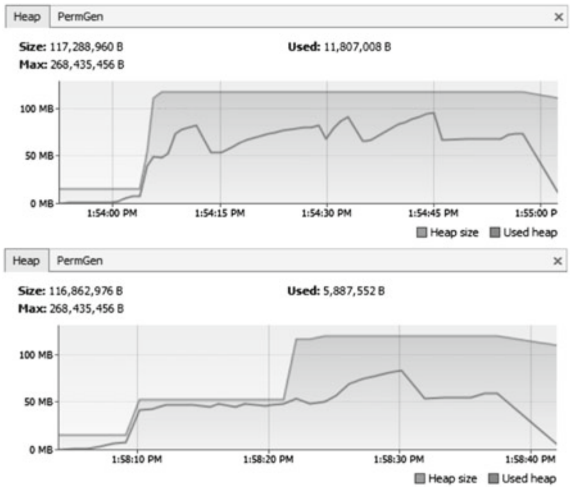

Fig. 4. Memory Usage

1. Heap-size is used less than before.
2. Used-heap is better than before with no sudden change.
3. Heap using-efficiency is better than before: average ratio between used-heap and heap-size is increased.

5 Conclusions

This study proposes architecture of a plug-in that support code optimization and secure programming techniques. After that, we build this plug-in for Java-environment Eclipse and evaluate results of performance of Java application before and after using plug-in to confirm it worth.

Code optimization and secure programming have a big number technique, and this study only works with four of them. So future works should extend set of rules to confirm application of AST on this field. Furthermore, refactoring process can be better, for example can be done automatically or semi-automatically.

References

1. Viega, J., Messier, M.: Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Input Validation & More. O'Reilly Media, Inc., Sebastopol (2009)
2. Wheeler, D.A.: Secure programming for linux and unix howto (2003)
3. Oaks, S.: Java Security. O'Reilly, Sebastopol (1998). (ISBN 1565924037)
4. Bloch, J.: Effective Java. Addison-Wesley Professional, Amsterdam (2008)
5. Aho, A.V., et al.: Compilers: principles, techniques, & tools. Pearson Education India (2007)
6. Gross, T.K.R.: Code optimization of pipeline constraints (1983)
7. Leupers, R.: Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools. Springer Publishing Company, Incorporated, New York (2010)
8. Pham, H.: System Software Reliability. Springer, London (2006)