An automatic approach to detecting and eliminating lazy classes based on abstract syntax trees

Wei Liu[1], Zhigang Hu[1, 2*], Hongtao Liu[2]

[1]*School of Information Science and Engineering, Central South University, Changsha 410083, China*

[2]*School of Software, Central South University, Changsha 410075, China*

Received 27 September 2013, www.cmnt.lv

Abstract

To detect and eliminate lazy classes in source code, an automatic approach based on abstract syntax trees (ASTs) is proposed. Source code files transform to ASTs at first, then the relationships between classes are extracted from the ASTs. Three common relationships are considered, which are generalization, association and dependency. Some definitions are proposed to represent the classes set of different kinds of relationships. After carrying out several set operations on these sets, the candidate lazy classes set is obtained. By further manual examination, the true lazy classes are acquired. Finally, a specific lazy class will be removed automatically from the project. Four projects are tested to detect and eliminate the lazy classes. The experimental results show that the proposed detection algorithm has high precision rate. In addition, this approach has good efficiency, and its execution time has a linear relationship to the size of a system.

Keywords: code smells, lazy classes, abstract syntax trees, refactoring, classes set

1 Introduction

Code smells are symptoms or indicators in the source code that indicate potential problems. The well-known 22 code smells are described in [1] by Martin Fowler and Kent Beck. The identifications or detections of code smells are useful in the sense that they might constitute prescriptive guidance for performing certain types of refactoring. Some common code smells emerge frequently in the existing code, such as Duplicated Code, Long Method, Large Class, Lazy Class, Switch Statements and so on. Code smells affect the maintainability of software systems, and they are important indicators for code refactoring [2, 3].

Recently, code smells detection and automatic refactoring become hotspots in software engineering research. Lots of code smell detection approaches have been proposed. Radu Marinescu [4] presented a metric-based approach to detecting code smells with detection strategies and developed a PRODETECTION toolkit that supported code inspections based on detection strategies. Naouel Moha et al. [5, 6] proposed a DECOR (DEtection & CORrection) method that described all the steps necessary for the specification and detection of code and design smells. Moreover, they introduced an approach to automating the generation of detection algorithms from specifications written using a domain-specific language, and they specified 10 smells and generated automatically relevant detection algorithms using templates. Foutse Khomh et al. [7, 8] presented a Bayesian approach for the detection of code and design smells, their approach could handle the inherent uncertainty of the detection process. In addition, they presented BDTEX (Bayesian Detection Expert) and GQM (Goal Question Metric) based approach to building Bayesian Belief Networks (BBNs) from the definitions of code smells. Hui Liu et al. [9] proposed a detection and resolution sequence for different kinds of bad smells to simplify their detection and resolution, they highlighted the necessity of managing bad smell resolution sequences with a motivating example and recommended a suitable sequence for commonly occurring bad smells. A. Ananda Rao et al. [10] proposed a quantitative method, which made use of the concept design change propagation probability matrix (DCPP matrix) to detect two important bad smells, which were shotgun surgery and divergent change. Although there are many code smells detection approaches, few of them can detect Lazy Class. So, a special approach to detecting and eliminating lazy classes is needed. Min Zhang et al. [11] performed a systematic literature review of 319 papers about code bad smells and analysed in detail 39 of the most relevant papers, they found that our knowledge of some code bad smells remains insufficient and some code bad smells receive little most research attention, such as the Lazy Class.

In addition, some tools have been developed for detecting code bad smells automatically and several research works have been done on them. Francesca Arcelli Fontana et al. [12, 13] gave reviews about the current panorama of the tools for automatic code smell detection, and they assessed many frequently-used tools, such as CheckStyle, in Fusion, PMD, and so on. Moreover, they outlined the main differences among these tools and the different results they obtained. Their research results show

[*]***Corresponding author*** e-mail: zghu@csu.edu.cn

that few tools could detect Lazy Class, let alone eliminate Lazy Class automatically.

2 Lazy class and abstract syntax tree

2.1 LAZY CLASS

Lazy Class is one of the bad smells in code, which indicates a useless class or a class with few responsibilities. Each class, which we have created should cost time and money to maintain and understand. Too many lazy classes will increase the complexity and scale of a software system. So, a class that is not doing enough to pay for itself should be eliminated. But if we face a huge project with millions of source code lines and thousands of classes, it is a so hard thing to find all lazy classes by manual handling. How to detect and eliminate lazy classes automatically is a meaningful topic in software engineering, especially in the field of code smells identifying and refactoring.

In this paper, we propose a novel and systematic approach to detecting and eliminating lazy classes automatically.

2.2 ABSTRACT SYNTAX TREE

In order to detect and eliminate lazy classes, we can analysis UML diagrams such as class diagrams. But a class diagram only describes high level relationships between classes, it loses some detail information, such as some dependency relationships. To get more relationship information between classes, we have to handle source code directly. However, source code analysis will raise the complexity and execution time. It has higher time and space complexity. In addition, during the stage of source code analysis, there is a lot of useless information affecting the execution efficiency.

To balance the complexity and efficiency for detecting the relationships between classes, we need a trade-off method. Abstract Syntax Tree (AST) is a proposed way to represent source code, which contains more information than class diagram. AST is used as an intermediate expression. In our approach, we use Java language as a sample, and the proposed approach can be used for other object-oriented languages. The Abstract Syntax Tree maps plain Java source code in a tree form, which is more convenient and reliable to analyse and modify programmatically than text-based source [14]. Every Java source file is entirely represented as tree of AST nodes that are all subclasses of the ASTNode.

In our approach, Eclipse is used as an IDE (Integrated Development Environment) to analysis Java source code and implement refactoring. It provides JDT (Java Development Tools) and Eclipse AST to handle Java source code. Eclipse JDT contains a group of APIs to access and operate source code, it contains two different ways to access Java source code: Java Model and AST. Eclipse AST is an important part of Eclipse JDT, which is defined in the package named org.eclipse.jdt.core.dom. In Eclipse AST, there are some classes to modify, create, read, and delete source code. In order to have good expandability and flexibility, Eclipse AST is designed based on the Factory Method pattern and the Visitor pattern [15].

3 Automatic detection and elimination algorithm

After transforming source code to abstract syntax trees by Eclipse AST, we can detect and extract all relationships between classes by handling the ASTs. If a class is a lazy class (redundancy class), it has no relationship to other classes. In general, there are three kinds of relationships between classes, including generalization, association and dependency. If we find that all of the other classes have no any relationship to a specific class, the class is maybe an islet. It means that the class is very likely a lazy class. So, the problem of detection of lazy classes is transformed to a problem of finding isolated classes.

In order to describe the process for searching isolated classes and detecting candidate lazy classes, a series of definitions are proposed as follows.

Definition 1: Project Classes Set (PCS). PCS is a set that stores all classes' names in a project.

Definition 2: Super Classes Set (SCS). SCS is a set that stores all super classes' names of a specific class. The super interfaces are also in SCS.

Definition 3: Associate Classes Set (ACS). ACS is a set that stores all associate classes' names of a specific class. Association classes' instances are attributes of a specific class.

Definition 4: Dependent Classes Set (DCS). DCS is a set that stores all dependent classes' names of a specific class. Generally, dependent relationships are represented by three main ways: a class's instance is one of the parameters of another class's method, a class's instance is the local variable in a method of another class, and a class invokes another class's static methods. If a class has one of the three aforementioned relationships to a specific class, it will be added to the specific class's DCS.

Definition 5: Relevant Classes Set (RCS). RCS of a class is a union set of the class's SCS, ACS and DCS. The formula to calculate RCS of class *i* as follows:

$$RCS(i) = SCS(i) \cup ACS(i) \cup DCS(i) . \quad (1)$$

In addition, RCS of a project is a union set of all classes' RCS in a software system. The equation to calculate RCS of a project as follows:

$$RCS(\text{Project}) = \bigcup_{i=1}^{n} RCS(i) . \quad (2)$$

Definition 6: Lazy Classes Candidate Set (LCCS). LCCS of a project is a set that stores all candidate lazy classes in a software system. Candidate lazy classes are in the *PCS* but not in the *RCS*. We can obtain the *LCCS*(Project) using the following equations:

$$LCCS(\text{Project}) = PCS(\text{Project}) - RCS(\text{Project}). \quad (3)$$

For example, if the *RCS*(Project) = {A, B, C, E, F} and the *PCS*(Project) = {A, B, C, D}, the *LCCS*(Project) = *PCS*(Project) – *RCS*(Project) = {A, B, C, D} –g {A, B, C, E, F} = {D}. Here, E and F are in *RCS* but not in *PCS*, which are called library classes, such as the classes in JDK or other open source libraries. Library classes list in *RCS*, but they are not parts of the current system and do not list in the *PCS*. D is a candidate lazy class, it is a part of the system but maybe none of the others needs it Pseudo-code of automatic detection algorithm for candidate lazy classes is listed in Table 1.

TABLE 1 Pseudo-code of the automatic detection algorithm

Line	Pseudo-code
	Input: The name of the root directory, which contains source code files for detecting.
	Output: A set stored all candidate lazy classes' names.
1	declare a null Set named projectClassSet
2	declare a null Set named relevantClassSetofProject
3	
4	for each source code file in the directory
5	create an AST for the file
6	add the class name to projectClassSet
7	
8	declare a null Set named superClassSet
9	store all directly super classes of the current class to superClassSet
10	
11	declare a null Set named associateClassSet
12	declare a null Set named dependentClassSet
13	for each FieldDeclaration in the AST
14	store the field's type name (not primitive type) to associateClassSet
15	if there is a ClassInstanceCreation node
16	store the type name of the instance in ClassInstanceCreation node to dependentClassSet
17	end if
18	if there are TypeLiteral nodes
19	store the type names of classes in TypeLiteral nodes to dependentClassSet
20	end if
21	end for
22	
23	for each MethodDeclaration in the AST
24	store the parameters' type names (not primitive type) to dependentClassSet
25	store the exceptions' type names to dependentClassSet
26	store the type names of instances in all ClassInstanceCreation nodes to dependentClassSet
27	store the type names of classes in all static MethodInvocation nodes to dependentClassSet
28	store the type names of classes in all static fields access nodes (QualifiedName) to dependentClassSet
29	store the type names of exception in all CatchClause nodes to dependentClassSet
30	store the type names in all InstanceofExpression nodes to dependentClassSet
31	store the type names of classes in all TypeLiteral nodes to dependentClassSet
32	end for
33	
34	declare a null Set named relevantClassSetofClass
35	relevantClassSetofClass = superClassSet ∪ associateClassSet ∪ dependentClassSet
36	relevantClassSetofProject = relevantClassSetofProject ∪ relevantClassSetofClass
37	end for
38	
39	declare a null Set named lazyClassSet
40	lazyClassSet = projectClassSet - relevantClassSetofProject
41	return lazyClassSet

In Table 1, projectClassSet is used to store PCS (Line 1) and relevantClassSetofProject is used to store RCS of a project (Line 2). For each source code file in the project, an AST is created firstly, then the relevant class name is added to projectClassSet (Line 6). In Line 8-9, superClassSet is used to store SCS, and associateClassSet is declared to store ACS in Line 11 and dependentClassSet is declared to store DCS in Line 12. In Line 13-21, for each field of the class, if the type of field is not a primitive type, the type name is stored to associateClassSet which is used to store ACS. ClassInstanceCreation node and TypeLiteral node are also considered in the FieldDeclaration. If there is a ClassInstanceCreation node or a TypeLiteral node, relevant class's name will be added to DCS. In Line 23-32, all dependent classes are extracted from each method, eight situations are considered to detect different kinds of dependent classes. At last, relevantClassSetofClass is used to store RCS of current class (Line 34), and relevantClassSetofClass is the union set of SCS, ACS and DCS (Line 35). The relevantClassSetofClass is added to relevantClassSetofProject (Line 36). In Line 39, a set named lazyClassSet is used to store LCCS, lazyClassSet is the difference between projectClassSet and

relevantClassSetofProject, and the lazyClassSet is returned finally (Line 40-41).

Obviously, time complexity of the algorithm is $T(n) = n\times(m + k)$, here, n is the number of source code files in the project, m is the average number of fields in each class and k is the average number of methods in each class. Generally, m and k are not too large. If we define a suitable constant C, we can consider as: $1 \le (m + k) \le C$, and $T(n) = n\times(m + k) \le C\times n = O(n)$. It means that the execution time has a linear relationship to the number of source code files, and the automatic detection algorithm has good efficiency.

After detecting all candidate lazy classes stored in LCCS, we have to examine the candidates meticulously by manual. Some candidate lazy classes are not true lazy classes, for example, the entry class of a system, or a class, which is located in a configuration file, or a class which is used in user interface files (e.g. Java Server Pages). If a real lazy class is confirmed, we should eliminate it automatically. Pseudo-code of the automatic elimination algorithm for a lazy class is listed in Table 2.

In Table 2, all source code files in the project are checked. Several files maybe have more than one class, and each class transforms to a TypeDeclaration node respectively. In this approach, we do not save the package name of a class, so these classes which have a same class name should be considered. If another class's name equals to a specific class's name, corresponding TypeDeclaration node is stored into lazyClassNodeList. Finally, if we find that the size of lazyClassNodeList is greater than 1, it means that there are at least two classes with the same class name, we need to select the target class by manual. Otherwise, the TypeDeclaration node is deleted automatically. Time complexity of this automatic elimination algorithm is: $T(n) = n\times m$, here, n is the number of source code files in the project and m is the average number of class in each file. Most of the files have only one class, and a few files have more than one class. For a large project, we can assume that m trends to a constant. So, the time complexity is: $T(n) = O(n)$. Algorithm's execution time is proportional to the number of source code files, which can be used to represent the scale of system.

TABLE 2 Pseudo-code of the automatic elimination algorithm

Line	Pseudo-code
	Input: The name of the root directory, which contains source code files before refactoring and the class name of a true lazy class (lazyClassName). **Output:** The source code after modifying.
1	declare a null List named lazyClassNodeList
2	for each source code file in the directory
3	for each TypeDeclaration node in this file
4	if the class name equals to lazyClassName
5	store the TypeDeclaration node into lazyClassNodeList
6	end if
7	end for
8	end for
9	
10	if the size of lazyClassNodeList is 1
11	delete the TypeDeclaration node in lazyClassNodeList
12	else
13	prompt that some classes have the same name and need to be selected by manual
14	end if

4 Experiments and Results Analysis

To evaluate accuracy and performance of the detection and elimination algorithms, four projects are used to detect the lazy classes and their brief information is listed in Table 3. Among them, SunnySport is a desktop purchase-sell-stock management system developed by Java, JHotDraw is a Java GUI framework for technical and structured graphics, the "Ice Hockey Manager" is a hockey team management game running under Linux, MacOS and Windows, and TinyUML is a free software tool for easy and quick creation of UML 2 diagrams based on Java.

TABLE 3 Brief information of the four examined projects

Measures	Sunny Sport	JHotDraw	TinyUML	IceHockey Manager
Version	1.0	5.1	0.13_02	0.3
Line of code	10265	8419	13739	18085
Number of source code files	51	144	194	218
Number of Classes/Interfaces	105	155	207	222
Number of attributes	658	331	715	1432
Number of methods	377	1314	1644	1664

Precision is used to analyse and evaluate the accuracy of the detection results. We identify the true lazy classes by manual. And the formula for calculating precision as follows:

$$\text{Precision} = \frac{TP}{TP + FP}, \quad (4)$$

where, *TP* (True Positive) represents the number of true lazy classes in the *LCCS*. *FP* (False Positive) represents the number of false lazy classes in the *LCCS*. After examining the candidate lazy classes in *LCCS* one by one, *TP* and *FP* are obtained. Precision values of the four examined projects are listed in Table 4.

TABLE 4 Precision of the automatic detection algorithm

Project	TP	FP	TP + FP	Precision
SunnySport	2	0	2	100%
JHotDraw	6	0	6	100%
TinyUML	65	1	66	98.48%
IceHockeyManager	2	0	2	100%

In Table 4, three of the four project's precision values are 100%. All candidate lazy classes in LCCS of them are true positive instances. For example, the lazy classes in JHotDraw are DiamondFigure, NothingApplet, JavaDrawApplet, PertApplet, PatternPainter and JavaDrawViewer, to which none of other class has relationship. In TinyUML, there are 66 candidate lazy classes, including 64 test classes named XXXTest, a useless interface and a Main class, which is the entry of the project. The Main class is not a real lazy class, so it's a false positive instance. In general, the proposed approach has high accuracy for detecting lazy classes.

Moreover, we evaluate and analyse the performance of the proposed algorithm. The experiment is performed in a workstation equipped with a 2.67 GHz dual core processor and 2GB of RAM. For each project and each lazy class, we perform the detection and elimination program five times respectively. The average execution time of automatic detection is listed in Table 5.

TABLE 5 Automatic detection time of four projects

Project	Number of source code files	Line of code	Average execution time of detection (ms)
SunnySport	51	10265	1700.2
JHotDraw	144	8419	1903.4
TinyUML	194	13739	2274.4
IceHockeyManager	218	18085	2324.6

In Table 5, the average execution time is increased with the expansion of system's scale. We use the number of source code files as the *X*-axis and the average execution time of lazy classes' detection as the *Y*-axis. The linear relationship is shown in Figure 1.

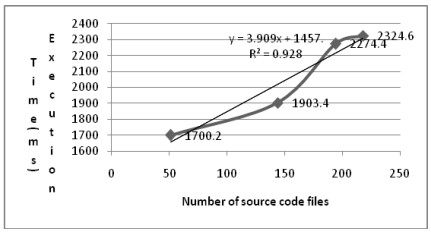

FIGURE 1 The linear relationship between number of source code files and average detection time.

Figure 1 presents the relationship between number of source code files and the execution time. In Figure 1, the thin line is a linear regression trend line. According to the algorithm analysis in Section 3, time complexity of the detection algorithm is T(n) = O(n). Execution time is proportional to the number of source code files which can be used to represent the scale of system. Experimental results are in accord with the analysis results, and show that the automatic detection algorithm has good efficiency.

To evaluate performance of the automatic elimination algorithm in Table 2. We select some true lazy classes from the *LCCS*, and the average execution time is list in Table 6.

TABLE 6 Automatic elimination time of four projects

Project	Number of source code files	Class Name	Average elimination time of class (ms)	Average elimination time of project (ms)
SunnySport	51	com.SunnySport.util.StockinTableModel	218.5	218.5
		com.SunnySport.util.DButiltow	218.5	
JHotDraw	144	CH.ifa.draw.contrib.DiamondFigure	358.5	358.7
		CH.ifa.draw.samples.nothing.NothingApplet	358.5	
		CH.ifa.draw.samples.javadraw.JavaDrawViewer	359	
TinyUML	194	test.tinyuml.ui.IconLoaderTest	608.5	608.5
		org.tinyuml.model.UmlModelListener	608.5	
		test.tinyuml.draw.NullElementTest	608.5	
IceHockeyManager	218	org.icehockeymanager.ihm.clients.devgui.ihm.scenario.TMScenarioList	686.5	686.5
		org.icehockeymanager.ihm.clients.devgui.gui.icons.icons	686.5	

In Table 6, the average execution time is also increased with the expansion of system's scale. We use the same method as Figure 1 to draw the relationship diagram between the number of source code files and the average execution time. The result is shown in Figure 2.

In Figure 2, the average elimination time also has a linear relationship to the number of source code files. The experiment results are in accord with the algorithm's complexity analysis. Execution time is in direct proportion to the system's scale.

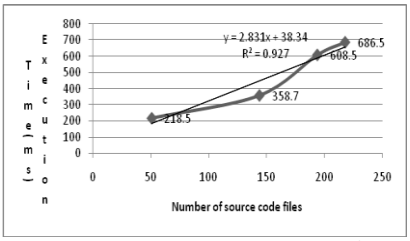

FIGURE 2 The linear relationship between number of source code files and average elimination time.

5 Conclusions

In this paper, a novel approach based on abstract syntax trees to detecting and eliminating lazy classes automatically is proposed. At the beginning, all source code files in a project transform to ASTs, then the relationships between classes are extracted from these ASTs. We analyse three kinds of classes' relationships, which are generalization, association and dependency. And we present some definitions to represent the classes set of different kinds of relationships. The candidate lazy classes set is obtained after a series of operations on these sets. We examine candidate lazy classes by manual, and remove true lazy classes finally. In order to verify our approach's correctness and evaluate its performance, four projects are used to perform the experiments for detecting and eliminating lazy classes. The experimental results show that the precision of the detection algorithms is very high. Moreover, our approach has good efficiency and it can be used for projects of different scales. Its execution time has a linear relationship to the size of system.

In the future work, we will improve and perfect our approach. On one hand, we will detect more kinds of files in a project except for source code files. Some information on classes' relationships lies in the configure files or user interface files, such as the JSP files, XML files. So, we need to detect these files in the next research work. On the other hand, we will find the isolated class groups in a system. Isolated class group is a series of classes, which have relationships to some other classes in the same group, but none of the class beyond the group needs them. All classes in an isolated class group form an islet of a set of lazy classes, and they need to be detected and eliminated together.

References

[1] Fowler M 1999 Refactoring: Improving the Design of Existing Code *Addison-Wesley Boston*

[2] Sjoberg D I, Yamashita A, Anda B C, Mockus A, and Dyba T 2013 *IEEE Transactions on Software Engineering* **39**(8) 1144-56

[3] Yamashita A, Counsell S 2013 Code smells as system-level indicators of maintainability: An Empirical Study *Journal of Systems and Software* **86**(10) 2639-53

[4] Marinescu R 2004 Detection strategies: Metrics-based rules for detecting design flaws *Proceedings 20th IEEE International Conference on Software Maintenance* 350-9

[5] Moha N, Gueheneuc Y G, Duchien L, Le Meur A 2010 DECOR: A method for the specification and detection of code and design smells *IEEE Transactions on Software Engineering* **36**(1) 20-36

[6] Moha N, Guéhéneuc Y G, Le Meur A F, Duchien L, Tiberghien A 2010 From a domain analysis to the specification and detection of code and design smells *Formal Aspects of Computing* **22**(3-4) 345-61

[7] Khomh F, Vaucher S, Guéhéneuc Y G, Sahraoui H 2009 A bayesian approach for the detection of code and design smells *QSIC'09 9th International Conference on* pp *Quality Software* 305-14

[8] Khomh F, Vaucher S, Guéhéneuc Y G, Sahraoui H 2011 BDTEX: A GQM-based Bayesian approach for the detection of antipatterns *Journal of Systems and Software* **84**(4) 559-72

[9] Liu H, Ma Z, Shao W, Niu Z 2012 *IEEE Transactions on Software Engineering* **38**(1) 220-35

[10] Rao A A, Reddy K N 2008 Detecting Bad Smells in Object Oriented Design Using Design Change Propagation Probability Matrix *Proceedings of the International MultiConference of Engineers and Computer Scientists (IMECS 2008) Hong Kong China, March 2008 International Association of Engineers* 1001-7

[11] Zhang M, Hall T, Baddoo N 2011 Code bad smells: a review of current knowledge *Journal of Software Maintenance and Evolution: research and practice* **23**(3) 179-202

[12] Fontana F A, Braione P, Zanoni M 2012 Automatic detection of bad smells in code: An experimental assessment *Journal of Object Technology* **11**(2) 1-38

[13] Fontana F A, Mariani E, Morniroli A, Sormani R, Tonello A 2011 An experience report on using code smells detection tools *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* 450-7

[14] Kuhn T, Thomann O 2013 Eclipse Corner Article Abstract Syntax Tree: http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html/ 1 Sep 2013

[15] Gamma E, Helm R, Johnson R, Vlissides J 1995 Design patterns: elements of reusable object-oriented software *Addison-Wesley: Boston*

Authors

Wei Liu, born in June, 1982, Hunan, China

Current position, grades: PhD Candidate of Computer Application Technology in Central South University, Senior Software Engineer.
University studies: PhD Student in Central South University, China, 2010.
Scientific interest: software engineering and data mining, including design patterns, refactoring, uml, reverse engineering, source code analysis and optimization, software metrics and parallel computing.
Publications: 10 publications.
Experience: more than 30 software projects.

Zhigang Hu, born in September, 1963, Shanxi, China

Current position, grades: Professor and Doctoral Supervisor at Central South University. Vice dean of the School of Software.
University studies: PhD, Mechanical Design and Theory, Central South University, China, 2003.
Scientific interest: software engineering, operating system, parallel computing, grid computing, cloud computing, embedded systems and high-performance platform.
Publications: more than 100 publications.
Experience: several research projects of National 863 Plan and the National Natural Science Foundation (NNSF).

Hongtao Liu, born in February, 1991, Hunan, China

Current position, grades: Master Degree student of Software Engineering in Central South University.
University studies: Bachelor of Engineering, Central South University, China, 2013.
Scientific interest: Software engineering, including design patterns, source code analysis, optimization and refactoring.
Publications: 5 publications.