applied sciences

MDPI

Article

Code Visualization for Performance Improvement of Java Code for Controlling Smart Traffic System in the Smart City

Bo Kyung Park [1], Geon-Hee Kang [2], Hyun Seung Son [3], Byungkook Jeon [4,*] and R. Young Chul Kim [1,*]

1 SE Lab., Department of Software and Communication Engineering, Hongik University, Seoul 04066, Korea; park@selab.hongik.ac.kr
2 Telecommunications Technology Association, Seongnam 13591, Korea; lustiness_hee@tta.or.kr
3 Reliability Technology Institute, Moasoft, Seoul 04066, Korea; hson@moasoftware.co.kr
4 Department of Computer Science, Gangneung-Wonju National University, Wonju 26403, Korea
* Correspondence: jeonbk@gwnu.ac.kr (B.J.); bob@hongik.ac.kr (R.Y.C.K.); Tel.: +82-33-760-8003 (B.J.); +82-44-860-2477 (R.Y.C.K.)

Received: 11 March 2020; Accepted: 13 April 2020; Published: 22 April 2020

Abstract: In an intelligent smart city like Sejong city in Korea, automatic and smart software is absolutely necessary for autonomous traffic and vehicles control systems. Therefore, these systems need to have an accurate and timely performance; otherwise, safety issues may arise. To resolve this, we propose our code visualization approach to adapt an object-oriented smart traffic control simulator, which analyzes Java code's complexity and identifies bad codes against performance. To achieve an accurate performance, we must repeatedly perform refactoring until attaining a range of system-defined performances through effective code visualization. As a result, we enhance the code complexity of the cyber-physical system-based software to achieve the exact performance. With this approach, we expect to obtain an accurate performance and complexity improvement of the object-oriented traffic control simulator without significant power consumption.

Keywords: cyber-physical system (CPS); SW performance; code visualization; refactoring; SW complexity

1. Introduction

In the fourth Industrial Revolution, the smart Sejong city in Korea is focused on solving energy and transportation problems including environmental monitoring, traffic information analysis, utility monitoring, public transportation, and incident reporting. Development of smart cities requires smart technologies and smart infrastructure. Software plays a key role in integrating these technologies. The cyber-physical system (CPS) technology is used to monitor and integrate smart city technologies [1,2]. CPS connects the general objects of the physical world with various computer functions of a complex system. Furthermore, CPS may include an extended concept of the existing embedded system, which has limited resources such as available power and memory and performance limitations leading to several operational problems [3]. Therefore, it is necessary that the software should operate reliably to deliver a high performance even in a limited resource environment.

Today's embedded systems have evolved from a simple to a complex structure with numerous computing units. These systems require accurate performance and diverse functions, which may necessitate huge and complex source codes, but CPS-based software performance or quality cannot be guaranteed [4,5]. Therefore, to overcome this problem, we propose code visualization for performance measurement and improvement of CPS-based software, especially Java code, through static analysis.

The procedure is as follows. (1) First, extract information by parsing the source code for identifying the internal structure of the CPS Software code and store the extracted information into our defined DB tables. (2) Thereafter, define queries to represent the internal structure based on our defined quality factors such as the coupling and cohesion indicators, and the performance degradation factor patterns. (3) Extract these patterns with a RuleChecker (programming mistake detector, PMD) [6]. (4) Measure the quantitative SW performance by using profilers. (5) Visualize the measured coupling indicators, performance degradation factors, and profiling information from the extracted inner structure of the code. (6) Finally, recommend improving the low-quality and critical code up to its defined quality factors, which aids in performance enhancement of the code. This method is expected to improve the performance and quality of CPS-based software.

This paper is structured as follows. Section 2 mentions the related materials and methods. Section 3 describes the performance measurement mechanism of Java code for a CPS-based smart traffic control simulator, the performance improvement of the system, and the case studies. The last section presents the conclusions and future scope.

2. Materials and Methods

2.1. Software Complexity

Software complexity includes the various characteristics of every module that affect the interactions within a software [7]. Over the last two decades, various complexity calculation techniques have been proposed in object-oriented programming systems. Traditional software complexity measurement metrics include McCabe's cyclomatic [8], Halstead's complexity measure [9], and Knot metrics [10]. We consider software coupling based on McCabe's complexity for measuring complexity metrics.

In general, if the level of cyclomatic complexity is more than 10–15, software management is difficult. For example, Microsoft Developer Network (MSDN) states that the complexity should not exceed 25 [11]. The Software Engineering Institute (SEI) of Carnegie Mellon summarizes the requirements for complexity and error rate, as shown in Table 1.

Table 1. Relationship between software complexity and error rate.

Complexity	Error Rate
Less than 10	5%
11~20	20%
21~50	41%
More than 51	60%

Cyclomatic complexity has the advantage of quantitatively presenting the complexity of a program. We specifically consider the increasing complexity within/between modules in the order of data, stamp, control, external, shared, and content coupling [8,12]. Therefore, content coupling is more complex than data coupling.

Cyclomatic complexity and Halstead's complexity measure are numerically represented and easy to understand for developers. However, these metrics cannot provide the detailed complexity between the modules of a software. Although the calculation uses part of the program, it is not perfect. Coupling can, however, accurately express the complexity between the modules in a software. This method also makes it easy to understand the basic control structures of a software. In this study, we apply the coupling method based on McCabe's complexity to reduce the code complexity of a software.

2.2. Tools for Software Performance Analysis

Profiling is a dynamic program analysis method that measures the time complexity together with the space of a program, the use of specific commands, and the frequency of function calls [13,14]. Common performance analysis tools use a variety of techniques to collect hardware and software

information. In this study, we use a profiler to verify the performance improvement through detecting and refactoring performance degradation factors.

Typically, Java applications run on top of a Java virtual machine. Therefore, traditional profiling methods cannot collect much detailed information. In Table 2, we list the Java profilers used for measuring the performance of Java applications that apply the profiler in the development of automation tools.

Table 2. Comparison of Java profilers. JVM, Java virtual machine.

Profiler	Live Object Analysis	CPU Sampling	Heap Trace	Thread Analysis	Method Analysis	Standalone (Console)	CallTree	Open Source	Automation Possibilities	Specificity
JMemProf	○	×	○	×	×	×	×	○	×	Ant Dependency
Java memory profiler (JMP)	○	×	○	○	○	×	×	○	×	Writing Code Directly
NetBeans Profiler	○	○	○	○	○	×	○	○	×	Net Bean IDE Dependency
JAMon API	×	×	×	×	○	×	○	○	×	Writing Code Directly
JBoss Profiler	×	×	×	×	○	○	×	○	×	Java Server Application Only (Using API)
Java Interactive Profiler (JIP)	○	○	○	×	○	○	○	○	○	Network Communication required between the JVM
Profiler4j	×	○	○	×	○	○	○	○	×	Network Communication required between the JVM
Hprof	○	○	○	×	○	○	×	○	○	Profiler provided by default in Oracle

JMemProf analyzes the information on the running objects and the information stored in heap memory when the application is run dynamically [15]. JMemProf has some limitations in that it does not extract reaction-speed data and depends on a Java build tool called Ant. Therefore, this tool cannot be used for automation purposes.

Java memory profiler (JMP) is a Java Application Programming Interface (API) based on the Java virtual machine tools interface (JVMTI) [16]. This tool does not profile CPU applications while profiling Java applications, but extracts information about running objects, heap memory, thread information, and method information during actual execution. This tool is also difficult to apply to automation processes.

NetBeans Profiler is a plug-in profiler provided by NetBeans Integrated Development Environment (IDE). This profiler can obtain data of heap, Central Processing Unit (CPU), and event through the Java virtual machine tools interface (JVMTI) [17]. However, this profiler is just dependent on the integrated development system of NetBeans. So automation is almost impossible.

JAMon provides an API for measuring the performance of web applications written in Java [18]. However, because this profiler only provides API, it is quite difficult to automate performance measurement.

JBoss Profiler extracts all information (method execution count, CPU usage, execution time, thread share, and so on) necessary for measuring the speed of the program [19]. However, JBoss Profiler is difficult to apply to actual automation because of only running on JBoss WebApplication Server.

Java interactive profiler (JIP) has less overhead costs and is used by applications to interoperate profiles of multiple applications [20]. However, it is very inconvenient because it needs to communicate with the JVM and has the ProcessID (PID) each time data is extracted.

Profiler4j is a Java profiler based on bytecode instrumentation [21]. This tool simply enters the source code file of the project and connects to the JVM to automatically profile it. Profiler4j is unstable for overall performance measurement. Therefore, this tool is not suitable for extracting and visualizing information.

Finally, Hprof can obtain information such as CPU utilization, heap allocation statistics, and monitor contention profiles [22]. The resulting data from Hprof are automatically printed to a text file after profiling. Hprof displays information in the order of rank, cumulative, count, and method. This information is run from the command line and automatically creates a data file. Therefore, we choose this profiler to be the best for visualizing the performance of Java code.

2.3. Related Studies of Smart City's Software

A smart city is a technology that combines smart technology and smart infrastructure. Smart software plays an important role in integrating these technologies. One part of a smart city is focusing on solving transportation problems such as traffic information analysis, transportation monitoring, public transportation systems, and accident reporting. In particular, a smart city, Sejong, in Korea needs smart software for autonomous traffic systems, pedestrian control systems, and vehicle control systems, which are directly linked to safety issues. These systems must be accurate and perform at the right time. For this, many researchers are conducting various studies on safety, reliability, and performance.

In [23], Anilloy Frank et al. proposed an Internet of Things (IoT)-based traffic control system to overcome the effects of traffic congestion. The proposed system measures the actual traffic density on the road and uses real-time video image processing technology. The system also determines traffic density on both sides of the road and activates the user's software application to control traffic.

In [24], Omar Abdel Wahab et al. deals with the problem of detecting and identifying malicious vehicles CEAP (collection, exchange, analysis, propagation), which is a cluster-based intelligent detection model for malicious vehicles. This model combines support vector machine (SVM) classification and monitoring concepts to optimize the decision-making process. This method uses machine learning techniques of SVM in an incremental and online manner to classify whether or not multi-agent Vehicular Ad-Hoc Network (VANET)'s smart vehicles are cooperative. This method collaboratively collects representative evidence to make an integrated decision, and analyzes the collected data using online SVM. This method can improve the accuracy of detection and the routing process, and then reduce false information.

In [25], quality of service optimized link state routing (QoS-OLSR) protocol deals with the VANET clustering problem. Many researchers have proposed several clustering algorithms for VANET and MANETs (mobile ad hoc networks). However, mobility-based algorithms ignore the quality of service requirements that are important to VANET's safety, emergency, and multimedia services. Moreover, because the QoS algorithm is dedicated to MANET, it ignores high-speed mobility constraints. To solve these problems, QoS-OLSR [25] proposed a method of forming a stable cluster and maintaining stability in the event of communication or link failure and satisfying service quality requirements.

In [26], Ke Hong claims that performance, security, and safety are necessary because our daily lives are rapidly relying on smart-end systems such as smartphones, wearable devices, and emerging autonomous vehicles. These types of assurance are essential to the design and implementation of software for smart-end systems. The current system lacks the ability to test and verify performance, security, and safety requirements for software against new attempts. Therefore, this study proposes the test method of smart-end systems' performance, security, and safety requirements; detection of important security principles through program analysis; and verification of compliance with safety requirements.

These studies are important as a way to solve traffic problems in smart cities. However, each study is slightly different as follows. The work of [23] has developed a system for traffic control. In contrast, [24] and [25] are about the stability and service on the network, which is different from the

purpose of our study. In [26], the research is related to the performance, security, and verification of the software in the smart end system, which is like the purpose of our study. However, this research attempts to develop a secure software by testing and verifying the requirements of performance, security, and safety aspects of the smart-end-system. In contrast, our paper analyzes the complexity of inner code structure through our visualization method, and guides bad code structures for refactoring to enhance software performance. As a result, the improved code is improved in quality and performance compared with the existing code. These factors are different from other researchers [23–26].

3. Performance Measurement Mechanism for Java Codes

Figure 1 shows the performance measurement and quality improvement process for smart traffic control simulator's code written in Java [27,28]. This process consists of four steps: code visualization, extracting performance degradation factors, performance measurement, and SW quality improvement.

The first step is to visualize the internal structure of the Java code by code visualization. We improved the quality of the software by measuring its coupling based on the visualized results. Step 2 uses the RuleChecker to define the rule for that pattern. We thereafter extracted the elements that violate the rule that is newly defined. The extracted information is stored in the database as an Extensible Markup Language (XML) file. Step 3 involves performance measurement and dynamically analyzes the Java code using the Hpof profiler. The software performance is measured using dynamically analyzed results. HprofDataExtractor refines the data extracted by the profiler into XML data. Step 4 improves software quality by refactoring to reduce software complexity and performance degradation factors.

Finally, steps 1, 3, and 4 are repeated until satisfactory and accurate performance is achieved. The steps are described in the following subsections.

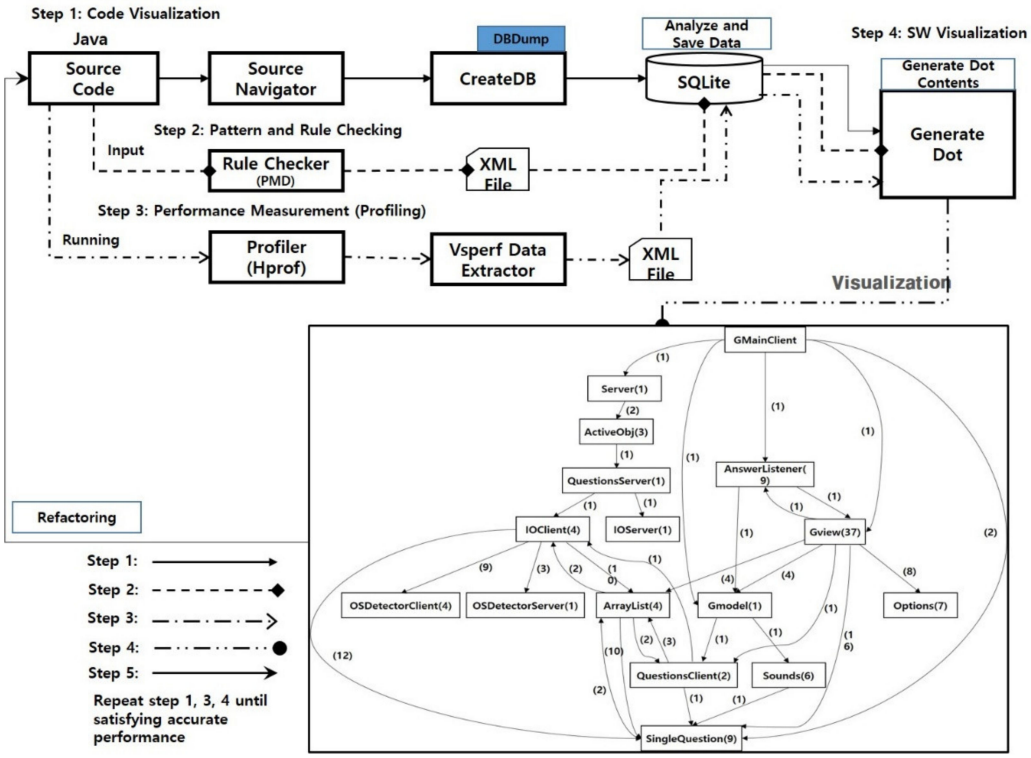

Figure 1. Performance measurement mechanism for Java code of cyber-physical system (CPS)-based traffic control simulator. PMD, programming mistake detector.

Step 1: Code Visualization

Code visualization is the process of visualizing the internal complex structure of a source code and improving the software quality based on coupling and cohesion [29,30]. Code visualization can

consist of either commercial or open sources. Even though commercial tools are expensive and limited to linking and expanding with other open sources, we can plug and play with the static analysis, that is, critical important functions of them. In addition, open source tools are flexible, but have important functions than commercial tools. In this paper, we just use open source tools (Source Navigator, SQLite, Graphviz) for the flexible construction of code visualization. For example, source navigator (SN) parses source code [31], SQLite can store the analyzed data in database tables [32], and Graphviz creates images of data extracted from database tables [33]. In our other case, we did plug and play these tool chains with the static analysis of the commercial Liverpool Data Research Associates (LDRA) tool.

Figure 2 shows a code visualization process such as analyzing a Source Navigator Database (SNDB) file, extracting code information, saving data to DB table, defining quality indicators, and visualizing code.

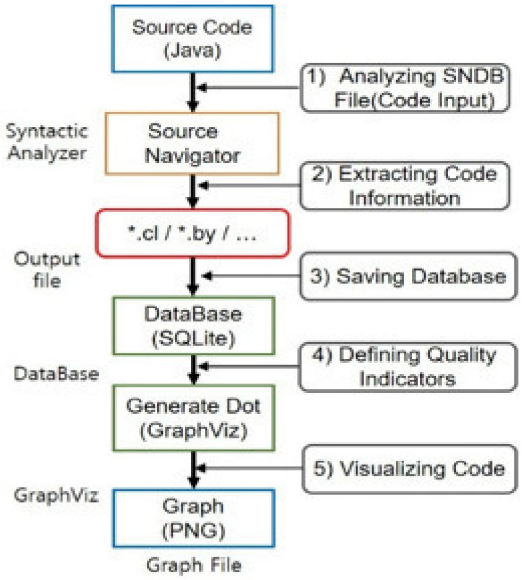

Figure 2. Code visualization process.

(1) Analyzing SNDB File: Store the data analyzed by the SN into SNDB files. Thereafter, input the program source code into the SN, and extract the information from the source code analyzed by static analysis. The SNDB file contains the complete information of the code program such as function information, local variables, global variables, and parameters. Figure 3 shows the SNDB files generated after parsing the SN.

Figure 3. SNDB files generated after parsing the source navigator (SN).

SNDB files are in the form of binary files (.cl, .con, .e, .fu, .gv, .iv, and so on) at the source analysis stage. These file formats can be converted to text data that can be directly read using DBdubmp.exe.

(2) Extracting Code Information: Extract all code information using DBdump.

(3) Saving Database: Save data in each table of DB. The tables of the database are SNDB_BY, SNDB_CL, SNDB_IN, SNDB_IV, SNDB_LV, and SNDB_MD, respectively. SNDB_BY stores the referenced class name, SYMBOL name, and SYMBOL type; SNDB_IN contains information such as subclasses and inherited classes; SNDB_IV contains information on member variables; SNDB_LV contains information on local variables; while SNDB_MD contains information about the method.

(4) Defining Quality Indicators: Define the quality indicators to determine the quantitative measurement score and make the query for extracting the degree of coupling strength for high quality of the software.

Figure 4 shows the types of coupling strength [34]. As the content coupling in a module increases, the density of coupling increases. In other words, as the degree of coupling strength increases, the complexity increases. Therefore, in order to improve software complexity, the degree of coupling strength must be lowered. In this study, we investigated how to extract data coupling and stamp coupling. For this purpose, we entered the information extracted from the SN into DB tables of SQLite, and thereafter defined queries to identify and measure coupling in source codes.

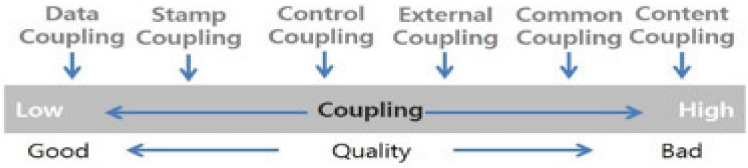

Figure 4. Types of coupling strength.

Table 3 shows the example of source codes and queries used to extract data and stamp coupling. An example of a data coupling method is 'dataTest' of data class. This method directly enters the value of the basic scalar data type. An example of stamp coupling is to create a SampleDataClass, input the data, and pass the class as a parameter of stampTest of the stamp class.

Table 3. Example of code and queries for extracting data and stamp coupling.

Coupling	Sample Source Code	Query
Data	Data data = new Data(); data dataTest(1, 1.01f, 2, 2);	Select REFER_CLASS_NAME, REFER_SYMBOL_NAME, REFERREED_CLASS_NAME, Referred_symbol_name, from SNDB_BY where referred_type = 'mi'
Stamp	Stamp stmp = new Stamp(); SampleDataClass sdc = new SampleDataClass(1, 1.01f, 2, 2); Stmp.stampTest(sdc);	Select argument_types from SNDB_MD where class_name = '" + class + "' and method_name = '" + method + "'

In addition, the query for extracting data coupling is called the class (REFER_CLASS_NAME) called from the SNDB_BY table, the name of the symbol (REFER_SYMBOL_NAME), and the name of the class (REFERRED_CLASS_NAME) and symbol (REFERRED_SBOL_NAME). The extracting query sets the condition of 'Referred_type' to 'mi'. The query only detects the method call part.

The query of stamp coupling receives the parameter (argument_types) data from the SNDB_MD (method definition) table. This query is entered in the 'isData_Stamp_True' method to determine the type of coupling. We defined queries to identify and measure coupling in source codes for other couplings such as control, external, common, and content.

(5) Visualizing Code: The quality indicators are quantified using the extracted information. Thereafter, it extracts code data by using a query called 'GenerateDotContents'. The Graph Description Language (DOT) script quantifies the quality indicator value. Subsequently, the Graphviz tool displays quantified values from the DOT script. Finally, it extracts the data into an image file.

Step 2: Patterns and Rules Checking for Performance Degradation Factors

To visualize CPS-based code performance, we create patterns and rules for the performance degradation factors [28,29]. Some factors of performance degradation are loop unrolling and loop down count, unnecessary control statements of the inner loops, and multiple if-then-else, among others. We used PMD (XPath) for rule extraction.

Programming mistake detector (PMD) is JAVA's static analysis tool [6], which can identify non-standard and potential problems in source code that the compiler cannot find. The main function of PMD is to extract high-value classes by analyzing dead code, empty if/while statements, excessive and complex expressions, unoptimized code, redundant code, as well as cyclomatic complexity. For C/C++ code, CPPCheck can automatically detect patterns by defining them in the source code as regular expressions and manually entering extraction paths [35]. For Java code, PMD is executed instead of CPPCheck from the command line, and users can add new rules, that is, PMD provides its own PMD rule designer, so we only need to create XPath rules.

Table 4 shows code patterns for performance degradation and rules for regular expression and XPath. For C/C++ code, we mentioned the regular expression pattern for loop unrolling and loop down count in detail [36,37]. The example code of Table 4 is increased by 1 from 0 to 1000 in the *'for statement'*. The sum variable adds the '*i*-th' value of the array. In this code, to detect *' i < 1000; i ++'* in the target code, '*i*' is a variable. The first character of a variable cannot be a number. Thus, the first character of the variable is defined as *'[a–z, A-Z, _]'*. All other characters of the variable can be any character, number, or '_'. Therefore, the remaining characters of the variable are defined as *'([a-z, A-Z, _, 0-9]) *'*. Thereafter, the iterator is recognized regardless of the length. To compare values inside a conditional expression 'i <1000', '*\<*' is used for '<' and '([0-9]) +' is used for 1000 as numeric values to compare conditions. For the incremental expression i++, the variable expression '[a-z,A-Z,_]([a-z,A-Z,_,0-9])*' is used, as described earlier. For the postfix increment '++', '\+ \+' is used.

Table 4. Code patterns and rules of performance degradation. PMD, programming mistake detector.

Pattern Name	Code Pattern	Regular Expression Rule (for C)	XPath Rule (PMD) (for Java)

(1) Loop unrolling and loop down count	int sum = 0;
	for (int I = 0; i<1000; i++)
	{
	sum + = array[i];
	}
		[a-z,A-Z,_]([a-z,A-Z,_,0-9])* \<
		([0-9])+;
		[a-z,A-Z,_]([a-z,A-Z,_,0-9])* \+\+
			//ForUpdate//
			PostfixExpression[@Image='++']

(2) Unnecessary control statements of inner loops	for(I = 0; i<1000; i++){
	if(i&&0 × 01) {
	do_odd(i);
	}else{
	do_even(i);
	}
	}
		[a-z,A-Z,_]([a-z,A-Z,_,0-9])* \<
		([0-9])+;
		[a-z,A-Z,_]([a-z,A-Z,_,0-9])* \+\+\)
		\{
		if\(
			//ForStatement//Statement//Block//
			BlockStatement//Statement//IfStatement

(3) Multiple if-then-else	if (a==1){
	}else if (a==2){
	}else if (a==3){
	}else if (a==4){
	}.....
		if\(
		([a-z,A-Z,_,0-9, \b, \s, \+„ V, *,
		\%%, -\>, \>, \<, \[, \]])*
		==
		([0-9])+\)
			//Statement//IfStatement[@Else='true']

Figure 5 shows the XPath extraction method using the PMD rule designer [6]. For a Java code, the user enters the Java code in the PMD rule designer in step ① of Figure 5, and clicks the Go button in step ②. At this time, the PMD rule designer generates the Abstract Syntax Tree (AST) tree in step ③. Using the generated AST tree, the user inputs the XPath query to extract in Step ④. Finally, the PMD rule designer automatically generates XML of an XPath query in Step ⑤. The XPath of code (1) in Table 5 is as follows:

//ForUpdate//PostfixExpression[@Image='++']

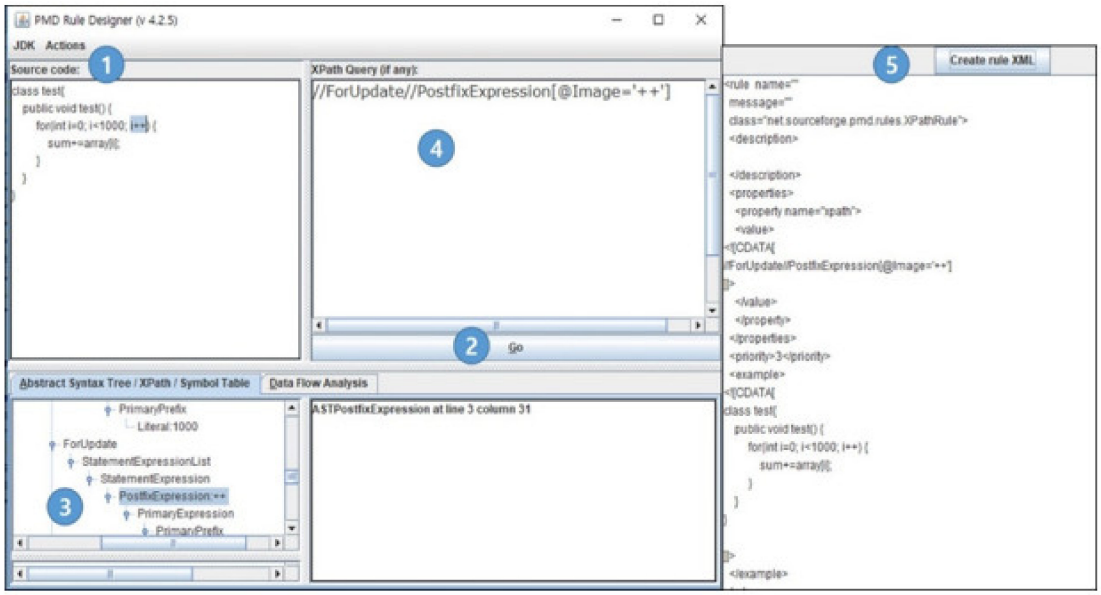

Figure 5. XPath extraction method using the PMD rule designer.

In this Java code, 'ForUpdate' refers to a 'for statement'. The PostfixExpression [@Image = '++'] means that the postfix expression is a '++' value in the 'for statement'. This pattern can detect all patterns such as the loop unrolling and loop down count example code. The pattern of unnecessary control statements of inner loops detects a condition that contains unnecessary conditional statements inside the loop. In the code pattern of pattern (2) in Table 4, 'i' of the 'for statement' is increased by 1 from 0 to 1000. The 'i' increases by 1 from 0 to 1000 in the 'for statement'. In the 'for statement', if the result of 'i && 0x01' is true, the *'do_odd ()'* method will be executed.

For C/C++ code [12], the regular expression of this pattern is as follows. This pattern represents the termination and incremental conditions of the 'for statement', like the loop unrolling and loop down count pattern. Thereafter, the opening of the body is indicated by "{". Finally, the beginning of the 'if-statement' is expressed as ("if \ (").

For Java, the XPath of the code pattern of unnecessary control statements of inner loops is composed of 'for', 'block', and 'if' statements, which are expressed as follows:

//ForStatement//Statement//Block//BlockStatement//Statement//IfStatement

Multiple if-then-else statements determine various conditions with one variable. This code can create various conditions through the value of the 'a' variable. In Table 4, for C/C++ code [12], the regular expression of the code pattern of pattern (3) is as follows. The beginning of an 'if-statement' is represented by 'if \ ('. The variable inside the if statement (a) is represented by '([a-z,A-Z,_,0-9, \b, \s, \+,, V, *, \%%, -\>, \>, \<, \[, \]])*'. To compare the 'a' value, use '==' sign. Finally, the number is represented as '([0-9]) + \)'. For Java code, the XPath of the code pattern of pattern (3) extracts the pattern in which the else statement is true.

Step 3: Performance Measurement (Profiling)

We assume that it is not important to measure the performance of all software. For software performance measurement, we detect degradation factors, and simultaneously measure the performance of those modules based on the primary language paradigm such as the basic statements, loops statements, branch statements, and calling mechanisms. We apply Hprof to measure the speed and resource usage of Java code for CPS [27].

Figure 6 shows the process of HProf execution. HProf runs during compilation of Java source codes and stores the bytecodes required for the profile in a class file. Thereafter, Hprof executes the class file and communicates with the JVM. Finally, we obtain four usage modes analyzed from the

source code such as heap allocation profiles, heap dump, CPU usage sampling profiles, and CPU usage time profile. However, we focus on using the CPU usage time profile in our study. This mode visualizes the number of executions and time information during the execution of the Java methods.

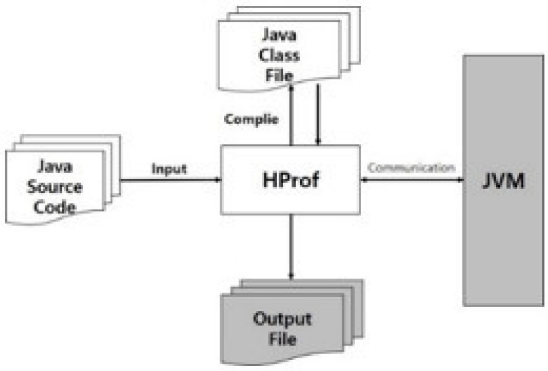

Figure 6. HProf execution process. JVM, Java virtual machine.

Figure 7 shows the profile mode of CPU usage time. Part ① of Figure 6 is the 'Total' data as the last execution time of the Java application. Part ② is the 'Rank' as the rank of occupancy of real time. Part ③ is the 'Self' as the execution time of the method itself. Part ④ is the 'Accum' as the total cumulative time. The performance time of the method is calculated by comparing the CPU usage profile data with the overall time.

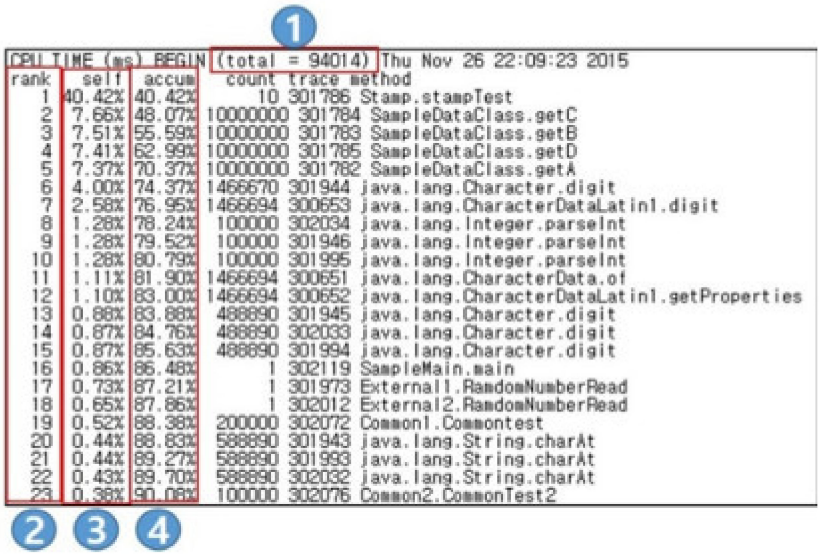

Figure 7. Result of CPU usage time profile mode.

Step 4: Software Visualization

Data collected in steps 1–3 are stored in a database (using SQLite). On the basis of this data, we create DotScript programs to visualize the internal structure of the software. Thereafter, we can generate the visual graph to illustrate SW complexity results, performance determinants, and performance measurement results of the original code. With this visualization, we show the inner problematic parts of the original source code. After refactoring with the problematic parts, we can rebuild the visualization with the improved results.

Step 5: Repeat steps 1, 3, and 4 until a satisfactory and accurate performance is achieved.

4. Results

In the hardware-based RoboCAR control system of Hanback Electronics Company in Korea, they need to have the software simulator of RoboCAR system to control remotely their hardware car, and then monitor their sensing data in real time as a training tool for education. Therefore, we developed a robot development system as a concept of simulation based on a virtual environment that deals an abstracted model with real hardware. In particular, the smart traffic control system in a smart city requires autonomous traffic control for people and vehicles. To achieve this, sophisticated and standardized software development is required. Therefore, many Korean software companies are doing their own research and development in their own way.

We show a smart traffic control simulator in Figure 8. In a virtual environment, this tool can simulate a variety of environments and situations (camera simulation, traffic light control, multiple vehicle creation, and so on) before operating a real smart traffic control system. In addition, the tool can create virtual robot cars, and simulate them with simultaneously operating the real car in a real environment. The smart traffic control simulator controls simulated models through the center screen. They can make a new robot car model with the part modules on the left of the screen, and represent the location data of the property window on the right of the screen. Table 5 describes detailed components of the smart traffic control simulator. The smart traffic control simulator consists of a menu bar, shortcut icon bar, model view tool bar, simulation view tab, property window, and output window.

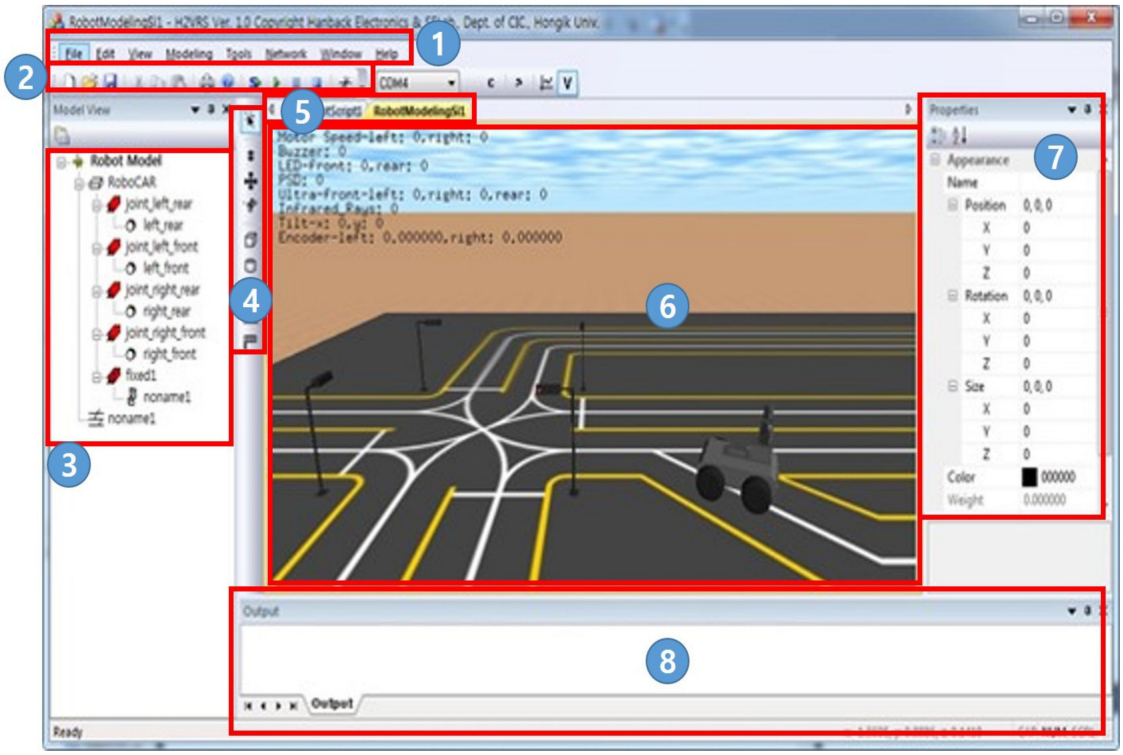

Figure 8. Smart traffic control simulator.

Table 5. Components of the smart traffic control simulator.

Number	Name	Description
1	MenuBar	Provide functions related to simulation such as saving, loading, starting, and stopping.
2	Icon Bar	Frequent used icons among the functions in the menu bar.
3	Model View	Assemble with part modules of the simulation view.
4	ToolBar	Provides tools for drawing robot models.
5	Simulation View Tab	Manage multiple simulation views.
6	Simulation View	Screen for running a robot model.
7	Property	Manage properties of robot model.
8	Output	Output the robot modeling and debugging results.

Figure 9 shows a test platform of a smart traffic system to test on a smart traffic control simulator.

Figure 9. Test platform of a smart traffic light system.

We published 'Evaluation of a Smart Traffic Light System with an IOT-based Connective Mechanism' to apply with a smart traffic control simulator to connect with the Internet of Things (IoT) technology [38]. With this, it is possible to extract the data needed to build an efficient and smart traffic signal mechanism. On the test platform with the simulator, the efficiency of smart traffic signals can be verified in a simulation environment.

Figure 10 shows the code visualization of the smart traffic control simulator. This is the result of code visualization. In Figure 10, part ① is the module that is inside the simulator package. Part ② is the module that is defined out of the simulator. Part ③ indicates the highest score of strong coupling between modules. Part ④ indicates each score of coupling with external modules. Both *DirectionCtrl* and *RobotModelingSimulationView* are called from the *CarModule* of part ⑤ with the red bolded arrows, which have strong couplings of modules among the visualization results.

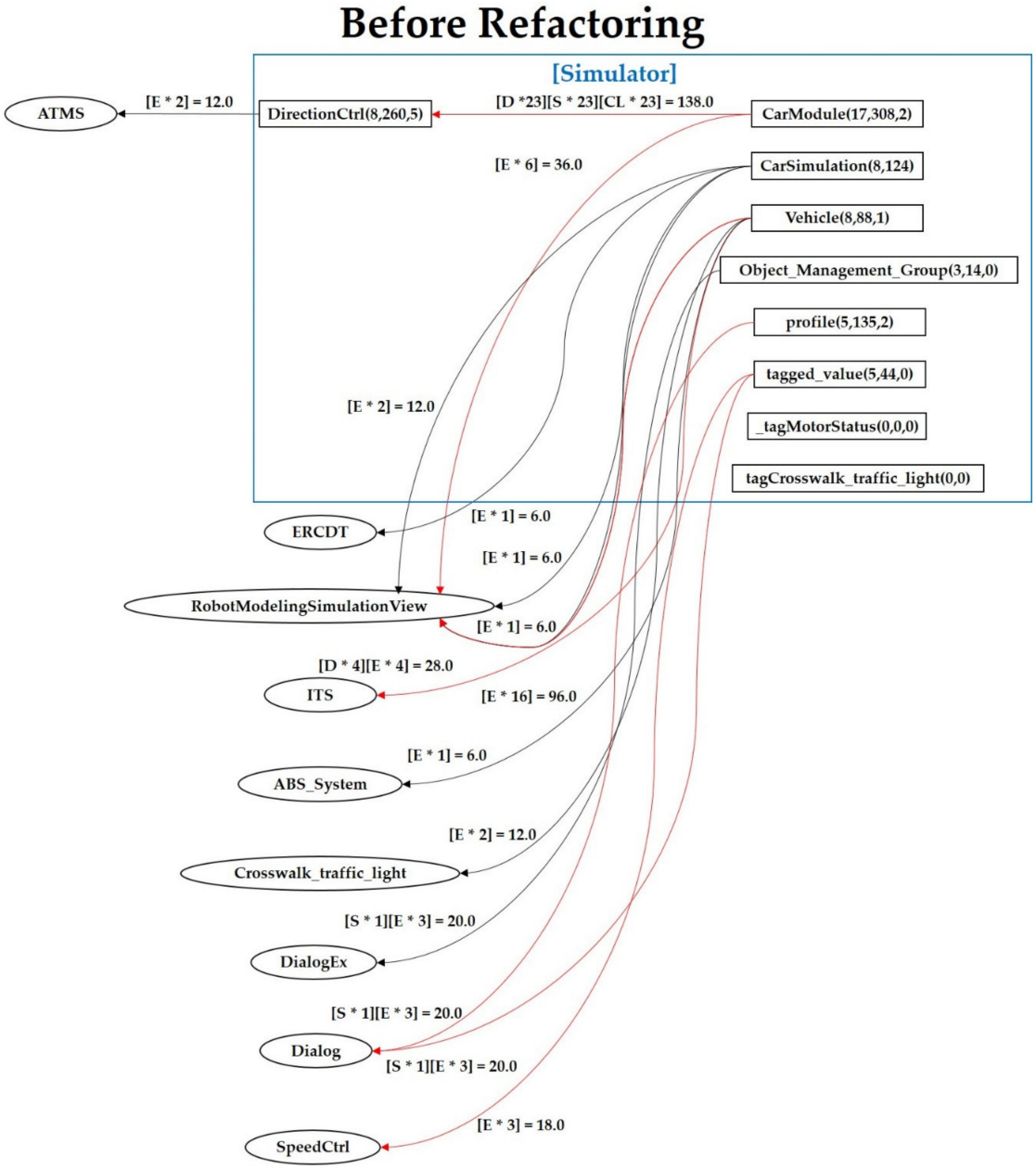

Figure 10. Inner code visualization of smart traffic control simulator before refactoring.

For example, *DirectionCtrl* of the red box in part ① has (8, 260, and 5), where 8 means the number of calls, 260 means execution time [ms], and 5 means the number of performance degradation factors. The module also has a red-color incoming arrow with [D*23][S*23][CL*23]=138.0, where D means 23 times of data coupling, S means 23 times of stamp coupling, CL means 23 times of control coupling, and their sum is 138.0.

To optimize the modules, we set the extraction criteria based on improving the coupling for refactoring. The extraction criteria are that the number of methods is 15. Here, the extracted module was displayed in red color. With this visualization of bad code patterns based on our defined quality indicators, we can easily be guided to rewrite the original code for better quality, that is, refactoring.

This section may be divided by subheadings. It should provide a concise and precise description of the experimental results, their interpretation, as well as the experimental conclusions that can be drawn.

4.1. Improving Coupling

The method of improving the couplings of the smart traffic control simulator is as follows. The degree of coupling is improved in the order of (1) stamp coupling-> data coupling and (2) external coupling-> data/stamp coupling. For reusability, we refactored the coupling of the smart traffic control system.

In the 'Before' area of Figure 11, we show the source code with the stamp coupling before refactoring. In other words, a source code 'StampTest' receives a SampleDataClass object as a parameter and executes a '*result* +=' statement with the getA(), getB(), getC(), and getD() methods of object c in a loop of the *for statement*.

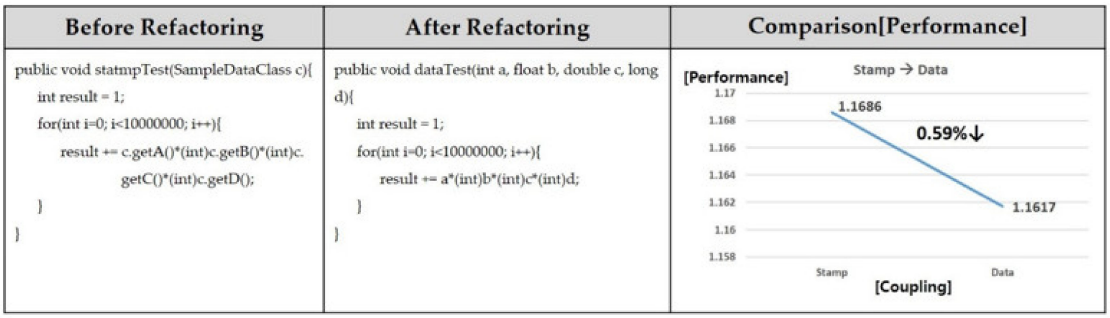

Figure 11. Enhancement of performance after refactoring stamp into data coupling.

Meanwhile, in the 'After' area of Figure 11, we refactor with the data coupling as directly taking a parameter without calculating from the internal data of the object. In the graph on the right of Figure 11, we show the performance enhancement when refactoring from stamp coupling to data coupling. The processing speed is 1.1686 [ms] for stamp coupling and 1.1617 [ms] for data coupling. This coupling change yields a 0.59% performance improvement. When the data are received as an object through a parameter, it is copied in memory, but a copy is not made when data are directly received.

In the 'Before' area of Figure 12, we show the external coupling before refactoring. The random.txt file is read through the FileReader object. The corresponding FileReader object is entered into the BufferedReader class to read the data in random.txt through the BufferedReader object. As each line goes by, data are added to an Arraylist called arr, and the operation is performed inside the loop through the data. However, the external coupling cannot be completely eliminated in 'Before'. Therefore, in the 'After' area of Figure 12, we improved the external coupling in three ways. The first way is to create a method that only loaded the data directly as in part ① and set it as private to prevent direct access. In the second method, as in part ②, the code sets up a getter that uses the data in another module. Finally, as in part ③, we refactor to accept only the ArrayList through the parameter and perform the operation. The improvement results are shown in the lower left graph. In the case of external coupling, the processing speed is 0.04236 [ms], and after refactoring with data/stamp coupling, the processing speed is 0.0198 [ms], which is improved by 55.85%.

As a result, we refactor the high coupling into the low coupling for software complexity. However, performance improvements are achieved in the case of stamp and refactoring of external coupling, which may be recommended.

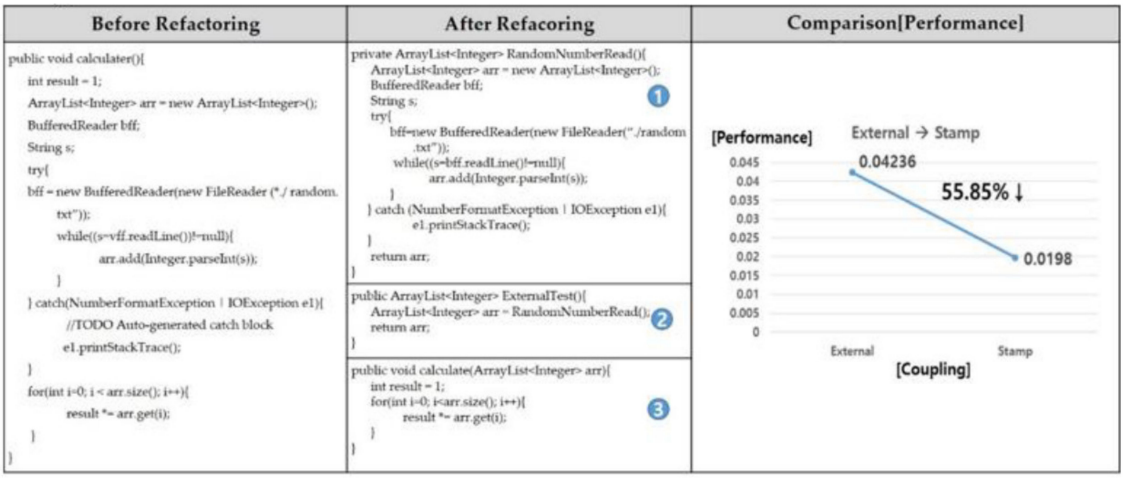

Figure 12. Enhancement of performance after refactoring external into data/stamp coupling.

4.2. Enhancement of Performance Degradation Factors

We define and detect degradation factor patterns to improve the performance of the smart traffic control simulator.

Figure 13 shows the source code for improving loop unrolling before and after refactoring. The case that is before improvement repeated the operation with i increasing by 1 from 0 to 1,000,000. Thereafter, we divide the sum by i*1.93. The code after the improvement can improve the performance by executing the functions in one iteration from i to i + 3 at once and the total number of iterations is reduced by 25%. The result is a 0.21% performance improvement that is increased by running all the loops, which takes 3.3849 [ms], and running less iterations, which takes 3.3841 [ms], that is, the performance is improved as the number of iterations decreases.

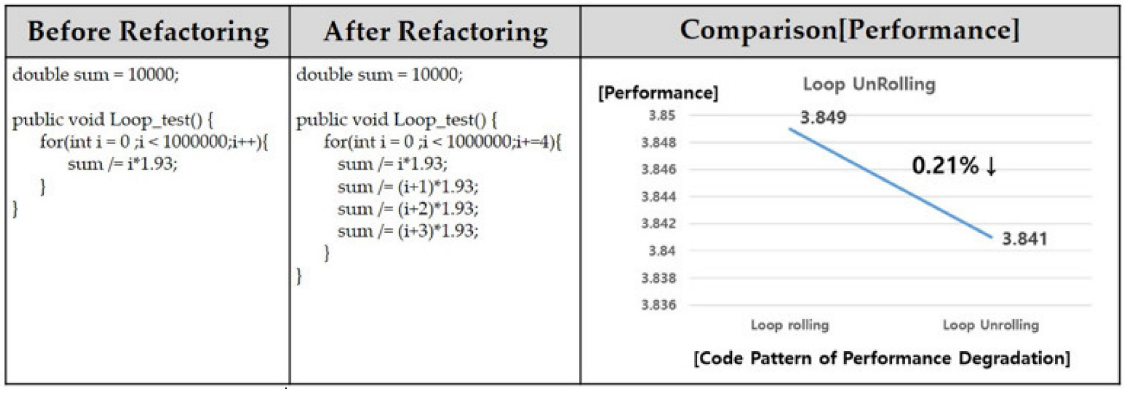

Figure 13. Before and after improving loop unrolling.

Figure 14 shows the source code before and after improving the loop down counter. In the source code before improvement, 'i' increased by 1 from 0 to 1,000,000. In the code after the improvement, we set the initial value of 'i' to 1,000,000, decreased 'i' by 1, and repeated until 'i' is zero. As a result, each execution time before and after the improvement is 3.86 [ms] and 3.85 [ms], respectively. Therefore, the performance is improved by about 0.26%. Compared with the up counter before improvement, the down counter does not allocate a register to store the final value. The down counter does not compare the value with zero each time. It can also reduce one command that compares 'i' with a number.

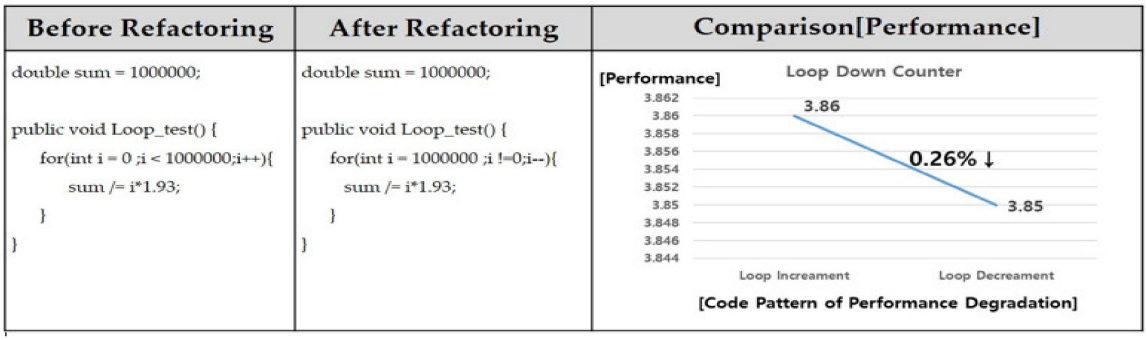

Figure 14. Before and after improving loop down counter.

Figure 15 shows the source code before and after the improvement for unnecessary control statements inside the loop. In this code, 'i' is increased by 1 from 0 to 1,000,000. Here, we discuss the code ('if (i% 4 == 0)'). If the remainder is different, this code performs different functions through an 'if statement'. The result of the performance measurement is 90.72 [ms]. Meanwhile, like the previous loop unrolling, it performs i ~ i + 3 at once and reduces the number of repetitions to 25% of the original code. The performance measurement result of this method is 53.82 [ms]. In other words, the performance measurement time is reduced from 90.72 [ms] to 53.82 [ms], and 40.67% improvement is achieved. The reason for the improvement is that the branch does not execute because the 'if then else statement' inside the loop is missing. In addition, the code after the improvement has reduced the number of iterations like loop unrolling.

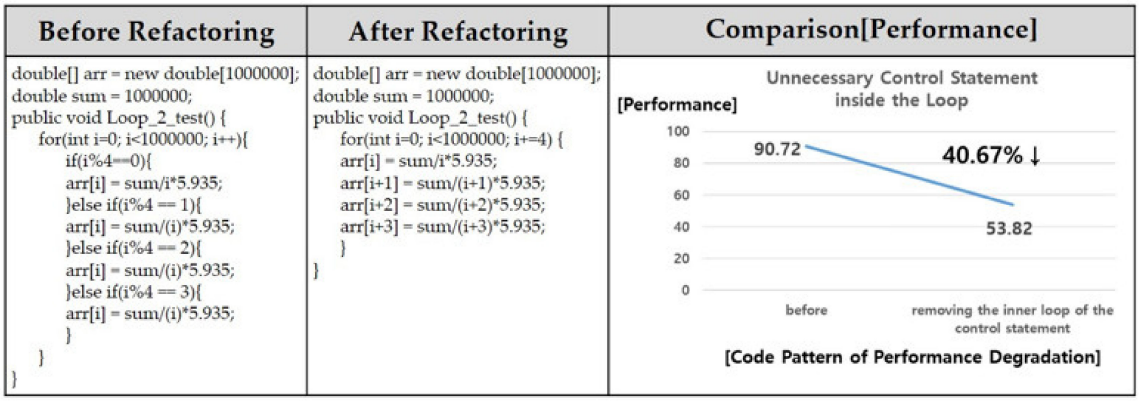

Figure 15. Before and after improving unnecessary control statement inside the loop.

Figure 16 shows the source code before and after the improvement for multiple if-then-else statements. The code before improvement differs from the remainder when the codes are divided by 5 through multiple if-then-else statements. Depending on the remainder, the sum value of the code may have a different value. Refactoring this code into a switch-case statement puts 'i%5' in the conditional statement of the switch statement. Depending on the value obtained from the result, the code for different case statements is executed. The improved code takes 664.993 [ms]. In other words, the performance measurement time is reduced from 686.377 [ms] to 664.993 [ms]. This means that there is a 3.12% performance improvement. The reason for the improvement is that, when multiple if then else statements are executed, all the conditions are compared. The function is performed on the branch statement that also satisfies the condition. However, the code that is improved with switch-case statements can perform functions at once without requiring multiple comparisons.

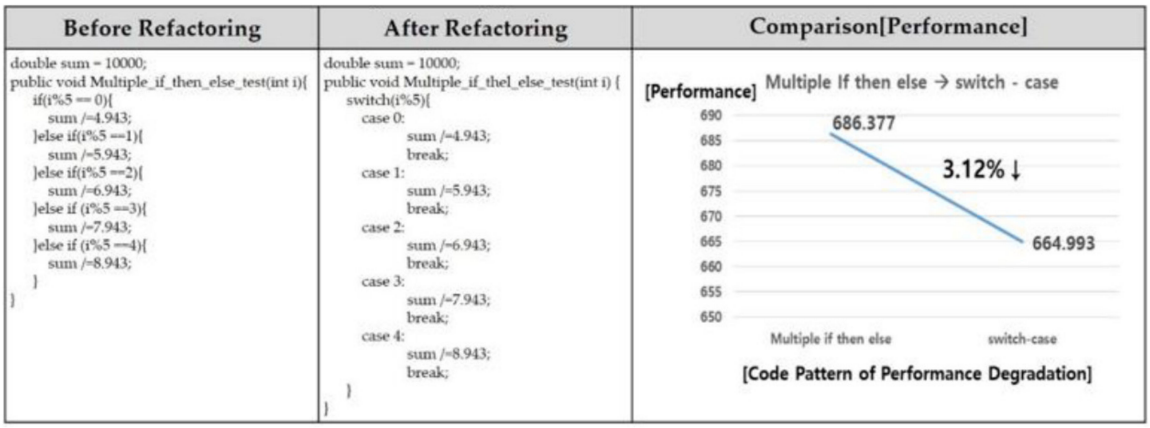

Figure 16. Before and after improving multiple if-then-else statements.

4.3. Result of Refactoring

Figure 17 presents the improved result. At the three checkpoint times, we can enhance the inner complexity of the simulators, which have low couplings between modules.

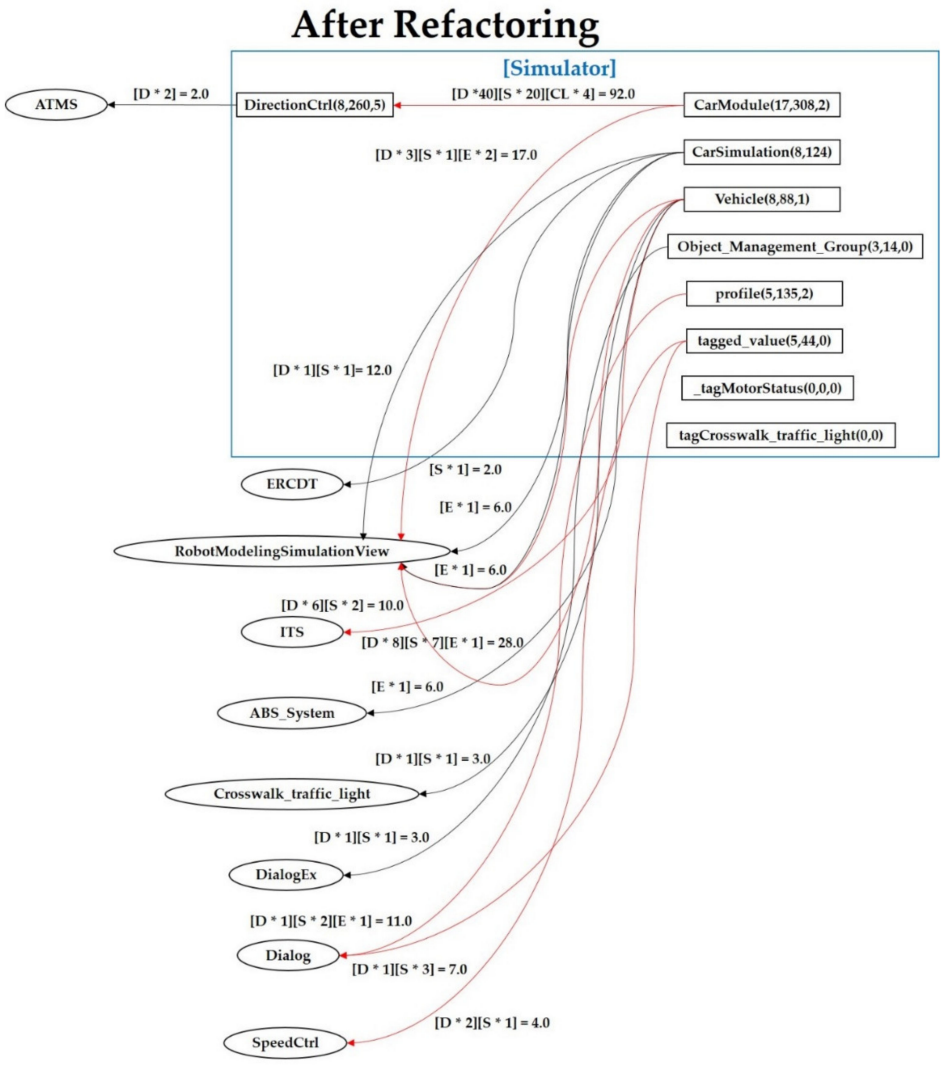

Figure 17. Improved result.

5. Discussion

We showed the performance measurement and quality improvement process in Section 3. In Section 4, we applied the proposed method to the smart traffic control simulator. In this section, we compare the results of both before and after refactoring about the smart traffic control simulator to evaluate the proposed approach.

5.1. Refactoring Results in Terms of the Level of the Coupling Mechanism

Table 6 lists the results of the improvement in the complexity of each module including CarModule, DirectionCtrl, CarSimulation, Vehicle, Object_Management_Group, Profile, tagged_value, _tagMotorStatus, and tagCrosswalk_traffic_light at each checkpoint time. However, it is not necessary to refactor the tagged_value, _tagMotorStatus, and tagCrosswalk_traffic_light modules because of their 0 complexity values. The second row of Table 5 shows the complexity values of the code before refactoring at the first baseline time. There are decreasing complexity values at each checkpoint time after refactoring, that is, we get low complexity values for all modules at the three checkpoint times.

Table 6. Refactoring results.

Weights of Coupling / Checkpoint	CarModule	DirectionCtrl	CarSimulation	Vehicle	Object_ Management_ Group	Profile	Tagged_ value	_tagmotorStatus	tagCrosswalk_ traffic_light
Baseline	138 + 36 = 174	12	6 + 12 + 6 = 24	96 + 12 + 6 + 28 = 142	8	20 + 6 = 26	20 + 18 = 38	0	0
Checkpoint1	113 + 36 = 149	12	6 + 12 + 6 = 24	72 + 12 + 6 + 28 = 118	8	15 + 6 = 21	16 + 14 = 30	0	0
Checkpoint2	97 + 32 = 129	7	2 + 8 + 2 = 12	29 + 3 + 6 + 18 = 56	3	16 + 6 = 22	12 + 10 = 22	0	0
Checkpoint3	92 + 17 = 109	2	2 + 3 + 2 = 7	28 + 3 + 6 + 10 = 47	3	11 + 6 = 17	7 + 6 = 13	0	0

The difference between couplings scores of *CarModule* is reduced as 174 – 109 = 65. Additionally, the difference between coupling scores of *vehicle* is also reduced as 142 – 47 = 95. The coupling scores of the remaining modules are also reduced. Here, the coupling score of *_tagmotorStatus* and *tagCrosswalk_traffic_light* is 0. This means that those codes are the dead codes. Therefore, deleting these two modules does not affect the entire code.

5.2. Identification Results in Terms of Performance Degradation Factors of the Basic Control Structures of 'CarModule'

Figure 18 shows the performance between before and after refactoring of the smart traffic control simulator. After identifying *CarModule* with the highest coupling value of baseline time in terms of the level of the coupling mechanism in Table 5, we focus on improving the performance degradation factors of *CarModule*. In Figure 18, we show the original information of 'before-refactoring' with the number of calling, execution time [ms], number of performance degradation factors such as *Multiple_if_then_else(1000000, 686.3778[ms], 4)*, and the changed information of 'after-refactoring' such as *Multiple_if_then_else(1000000, 664.9939[ms], 0).*

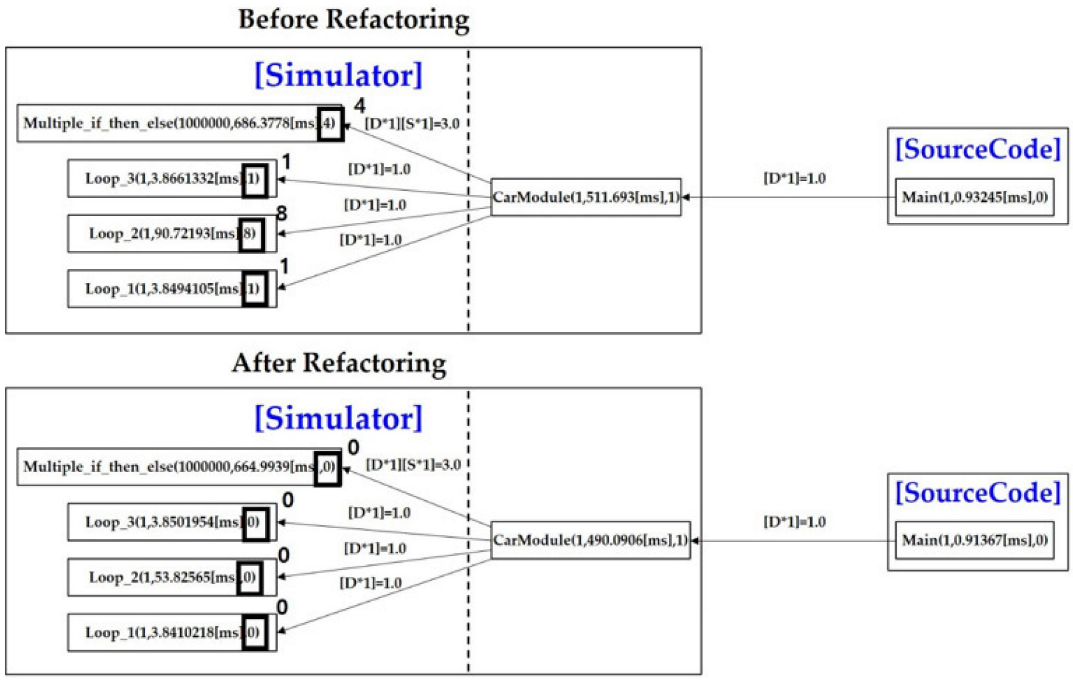

Figure 18. Code performance view between 'before' and 'after' refactoring.

In other words, before refactoring, we can see the number of degradation factors. As we show above, "multiple_if_then_else" modules have the digit 4. "Loop_3" module has 1, "Loop_2" module has 8, and "Loop_1" module has 1. After refactoring, we see an improvement in the values of these degradation factors to zero. The execution time between the modules is reduced, which implies that the CPS-based software speed has improved after refactoring of the code. Our approach extracts the bad code pattern of the source code based on quality factors. Thereafter, we visualize these code parts for refactoring.

5.3. Performance Result in Terms of Performance Measurement

Figure 19 depicts the methods of a '*CarModule*' with performance indicators such as execution time and the number of executions. We show performance measurements before and after refactoring methods of the same *CarModule*, which represented the method name, execution frequency, and execution time. The arrows connect between before and after refactoring the methods. The arrow value denotes the changed execution time. For example, a comparison was drawn for the *vehicleCtrl* method before and after refactoring. The difference value between these methods was −2347.4297 [ms], which indicated that the performance speed has improved after refactoring of the code.

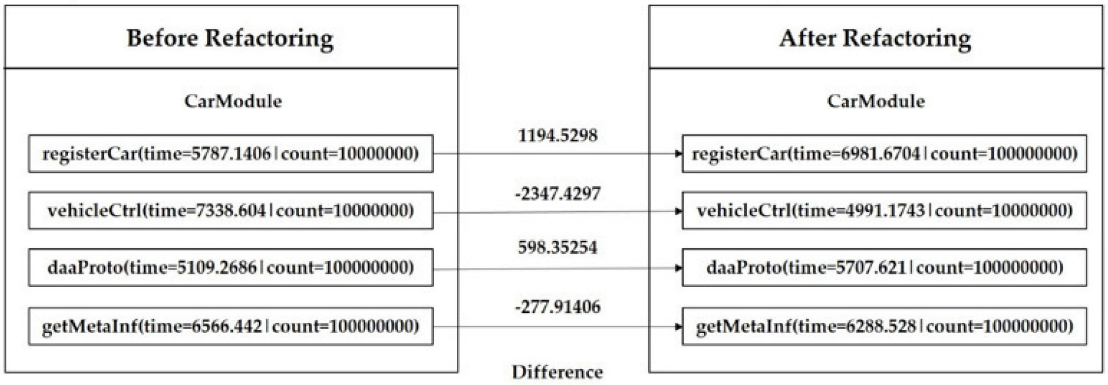

Figure 19. Detailed class view with performance indicators on *CarModule.*

Table 7 shows the performance measurement results. The total execution time before improvement is 24,801.455 [ms]. The total run time after improvement is 23,966.994 [ms]. The difference of the execution time of the *CarModule* from after to before is −834.4615 [ms]. Thus, we reduce the execution time of the most complex *CarModule* through refactoring.

Table 7. Performance measurement results.

Performance / Refactoring	Before	After	Difference Value
registerCar	5787.1406	6981.6704	1194.5298
vehicleCtrl	7338.604	4991.1743	−2347.4297
daaProto	5109.2686	5705.621	596.3524
getMetaInf	6566.442	6288.528	−277.914
Total	24,801.455	23,966.994	−834.4615

6. Conclusions

In this study, we propose a performance measurement mechanism of Java code for identifying high quality CPS-based software. For this purpose, we incorporate our code visualization approach into an object-oriented smart traffic control simulator, which analyzes Java code's complexity and identifies bad codes against performance. To achieve an accurate performance, we can perform refactoring until arriving within a range of system-defined performances through effective code visualization. For this purpose, we perform dynamic analysis using the Java performance measurement tool (HProf), which extracts information about the execution speed and frequency of a particular method. As a result, we enhance the code complexity of the CPS-based software to obtain the exact performance.

In future studies, we will consider all performance degradation factors and their relationships for resolving the dependent performance problem of basic control structures.

Author Contributions: B.K.P., G.-H.K., H.S.S., B.J., and R.Y.C.K. designed the present study, reviewed the literature, and drafted the manuscript; B.K.P. and R.Y.C.K. performed the static analysis and code visualization; B.K.P. and R.Y.C.K. critically revised the manuscript; all authors gave the approval for the final version of the manuscript submitted for publication. All authors have read and agreed to the submitted version of the manuscript.

Funding: This research was supported by the Basic Research Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Education (NRF-2017R1D1A3B03034102 and NRF-2017R1D1A3B03035421), and also supported by the Ministry of Trade, Industry, and Energy (MOTIE), KOREA, through the Education Program for Creative and Industrial Convergence (Grant Number N0000717).

Conflicts of Interest: The authors have no conflict of interest.

References

1. Lee, J.; Bagheri, B.; Kao, H.A. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manuf. Lett.* **2015**, *3*, 18–23. [CrossRef]
2. Rajkumar, R.; De Niz, D.; Klein, M. *Cyber-Physical Systems (SEI Series in Software Engineering)*; Addison-Wesley Professional: New Jersey, NJ, USA, 2017.
3. Kim, W.; Jeon, I.; Lee, S.; Park, S. CPS Technology Trends. *Wkly. Technol. Trends* **2010**, *1*, 1–10.
4. Son, H.S.; KIM, R.Y.C. The Pre-Testing for Virtual Robot Development Environment. *IEICE Trans. Inf. Syst.* **2018**, *101*, 1541–1551. [CrossRef]
5. Son, H.S.; Kim, W.; Jeon, I.; Lee, H.; Jeon, J.; Kim, R.Y. A Method of Representing Topographic Environment Data through SEDRIS in Cyber Physical System. *Korea Soc. Simul. Mag.* **2011**, *1*, 11–18.
6. PMD. Available online: https://pmd.github.io (accessed on 27 January 2020).
7. Tina, B.; Vili, P.; Marjan, H. Towards a Reliable Identification of Deficient Code with a Combination of Software Metrics. *Appl. Sci.* **2018**, *8*, 1902.

8. McCabe, T.J. A complexity measure, Software Engineering. *IEEE Trans. Softw. Eng.* **1976**, *4*, 308–320. [CrossRef]
9. Hariprasad, T.; Vidhyagara, G.; Seenu, K.; Chandrasegar, T. Software Complexity Analysis using Halstead Metrics. In Proceedings of the International Conference on Trends in Electronics and Information (ICEI), Tirunelveli, India, 11–12 May 2017; pp. 1109–1113.
10. Dand, Z.G.; Hemlata, V. Analysis and evaluation of quality metrics in software engineering. *Int. J. Adv. Res. Comput. Commun. Eng.* **2015**, *4*, 235–240.
11. Microsoft. Available online: https://docs.microsoft.com/en-us/visualstudio/code-quality/ca1502?view=vs-2019 (accessed on 15 December 2019).
12. Kwon, H.E.; Son, H.S.; Seo, C.Y.; Kim, Y.S.; Park, B.H.; Kim, R.Y.C. A Study on Comparing Object Oriented Paradigm with the Cohesion and Coupling Mechanism between Traditional Module. In Proceedings of the Korea Computer Congress, Busan, Korea, 25–27 June 2014; pp. 556–558.
13. Koziolek, H. Performance evaluation of component-based software systems: A survey. *Perform. Eval.* **2010**, *67*, 634–658. [CrossRef]
14. Henry, H. *Java Performance and Scalability, A Quantitative Approach*; Createspace: South Carolina, SC, USA, 2013.
15. JMemProf. Available online: https://java-source.net/open-source/profilers/jmemprof (accessed on 12 January 2020).
16. Java Memory Profiler (JMP). Available online: http://www.khelekore.org/jmp/ (accessed on 12 January 2020).
17. NetBeans Profiler. Available online: https://profiler.netbeans.org/ (accessed on 13 January 2020).
18. JAMon API. Available online: http://www.jamonapi.com/ (accessed on 28 December 2019).
19. JBoss Profiler. Available online: https://developer.jboss.org/welcome (accessed on 11 December 2019).
20. Java Interactive Profiler. Available online: https://sourceforge.net/projects/jiprof/ (accessed on 15 January 2020).
21. Profiler4j. Available online: http://profiler4j.sourceforge.net/ (accessed on 15 January 2020).
22. HPROF. A Heap/CPU Profiling Tool. Available online: https://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html (accessed on 3 February 2020).
23. Frank, A.; Al Aamri, Y.S.K.; Zayegh, A. IoT based smart traffic density control using image processing. In Proceedings of the 2019 4th MEC International Conference on Big Data and Smart City (ICBDSC), Muscat, Oman, 15–16 January 2019.
24. Wahab, O.A.; Mourad, A.; Otrok, H.; Bentahar, J. CEAP: SVM-based intelligent detection model for clustered vehicular ad hoc networks. *Expert Syst. Appl.* **2016**, *50*, 40–54. [CrossRef]
25. Wahab, O.A.; Otrok, H.; Mourad, A. VANET QoS-OLSR: QoS-based clustering protocol for vehicular ad hoc networks. *Comput. Commun.* **2013**, *36*, 1422–1435. [CrossRef]
26. Hong, K. Performance, Security and Safety Requirements Testing for Smart Systems through Systematic Software Analysis. Ph.D. Thesis, The University of Michigan, Ann Arbor, MI, USA, 2019.
27. Kang, G.-H.; Kim, R.Y.C.; Lee, J.H. A Case Study on Performance Improvement through extracting Software Performance Degradation Factors. *Int. J. Appl. Eng. Res.* **2015**, *10*, 90.
28. Park, B.K.; Kang, G.; Kim, R.Y.C. Performance Measurement of Procedural Code for CPS Multiple-Joint Robotics Simulator. *Adv. Eng. ICT-Converg. Proc. (AEICP)* **2019**, *38*, 75–78.
29. Park, B.K.; Kang, G.; Kim, R.Y.C. Software Visualization Approach for Performance Measurement of Object-Oriented Code based on Cyber-Physical Systems (CPS) Software. *Adv. Eng. ICT-Converg. Proc. (AEICP)* **2019**, *38*, 28–30.
30. Park, J.; Son, H.S.; Kim, R.Y. Developing an Automatic Tool for Visualizing Source Code against Bad Smell Patterns. In Proceedings of the Global Conference on Engineering and Applied Science, Okinawa, Japan, 7–9 July 2017.
31. Source Navigator NG. Available online: http://sourcenav.sourceforge.net/ (accessed on 16 December 2019).
32. Appropriate Uses For SQLite. Available online: http://www.sqlite.org/whentouse.html (accessed on 8 January 2020).
33. Graphviz. Available online: http://www.graphviz.org/ (accessed on 21 January 2020).
34. National IT Industry Promotion Agency. *Software Engineering White Book*; NIPA: JinCheon, Korea, 2018.
35. Cppcheck. Available online: http://cppcheck.sourceforge.net/ (accessed on 25 January 2020).

36. Park, B.K.; Jeon, B.K.; Kim, R.Y.C. Improvement Practices in the Performance of a CPS Multiple-Joint Robotics Simulator. *Appl. Sci.* **2020**, *10*, 185. [CrossRef]
37. Kang, G.; Kim, R.Y.C.; Lee, S.E.; Jeon, S.N. Extracting performance factors against performance degradation through Code Visualization. In Proceedings of the International Conference on Convergence Technology, Hokkaido, Japan, 29 June–2 July 2015; pp. 276–277.
38. Lee, H.J.; Kim, R.Y.C.; Son, H.S. Evaluation of a Smart Traffic Light System with an IOT-based Connective Mechanism. *Int. Inf. Inst.* **2017**, *20*, 953–961.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).