International Journal of Applied Engineering Research, ISSN 0973-4562 Vol. 10 No.90 (2015)

A Case Study on Performance Improvement through extracting Software Performance Degradation Factors

Geon-Hee Kang[1], R.Young Chul Kim[2*], Jae-Hyup Lee[3]
[1,2*]Dept. of Computer Information & Communication, Hong-ik University, Sejong, Korea
{[1]kang, [2*]bob}@selab.hongik.ac.kr
[3]Dept. of Computer Engineering, Koreatech University, Korea
[3]jae@koreatech.ac.kr

Abstract - Modern software industry is getting bigger on code size. Software is being used everywhere in our community, which needs the rapid development of the software. In this situation, nobody may guarantee software either high or low quality in software engineering. So there is a big issue how to make software high quality. According to these, software needs to be fixed for rapidly changed requirements. We suggest how to improve performance through extracting software performance degradation elements during refactoring of code complexity in software visualization. Due to adapting this visualized process with performance degradation extraction, we can do better performance of the legacy system, and fix the bad coding habits to programmers.

Keywords - component; Software Performance, Software Analysis, Code Visualization, Factors against performance degradation

I. INTRODUCTION

As modern software industry is getting bigger, there is an issue about high quality of software. However, the software industry is growing, which has focused on a code-centric development for the rapid release of the software. So the software is possible to produce the lower quality. Most software can be the invisibility which makes it difficult to manage them. Through resulting the code visualization, it is needed to manage software quality [1]. There are various quality attributes of the software such as, accuracy, reliability, efficiency, integrity, maintainability, portability, interoperability, time behavior, etc.

This paper introduces how to extract software performance degradation factors with the rule-checker, and also shows to compare performance on either changing or removing of performance degradation factors with a case study.

The paper consists as follows; Chapter2 explains what software performance and code visualization are. Chapter3 mentions the extracting method for performance improvement. Chapter 4 shows the improvement of the software performance through change and remove the Software Degradation on Case Study. Chapter 5 gives Conclusion and future works.

.

II. RELATED WORKS

2.1 Software Performance

Performance in computer science can be interpreted in various aspects, especially, availability, size and weight, response time. Software availability refers the probability that the Software will operate according to the requirements at the time. Availability calculation is calculated from the mean time between failure (MTBF) and the mean time to repair (MTTR). MTBF is the average time the application is run until an error occurs. MTTR means the average time required to repair & restore the service after a failure. Size and Weight can have a significant impact on performance because resources are limited in the Embedded System. Response time means the time it took to respond back from the input to the system or the execution unit. System speed will decrease as a response time is longer [2]. We use a case study extracted some elements which are related to Software Performance in this study, and compare them before/after removing and changing it.

2.2 Code Visualization

In order to be successful in software development and management, there are essential ways like software development processes, test automation, and quality certification. But, it is too lacking in resources and professional engineers to carry out such work. Therefore, we consider that the absolutely essential way is how to do Code visualization. It is also a technique to improve the efficiency of maintenance and quality control. Code Visualization consist of the visualization and the documentation. First, the visualization is a way to overcome software invisibility and easily understand the entire software development process to perform a quality control. Second, the documentation is a way to know how to develop the enterprise management, to plan for communicating with the outside or in certain circumstances, and also to enhance business's understanding between internal staff [1]. We use our tool chain code to show the package achitecture through a code visualization. Figure 1 shows the package achitecture with code visualization.

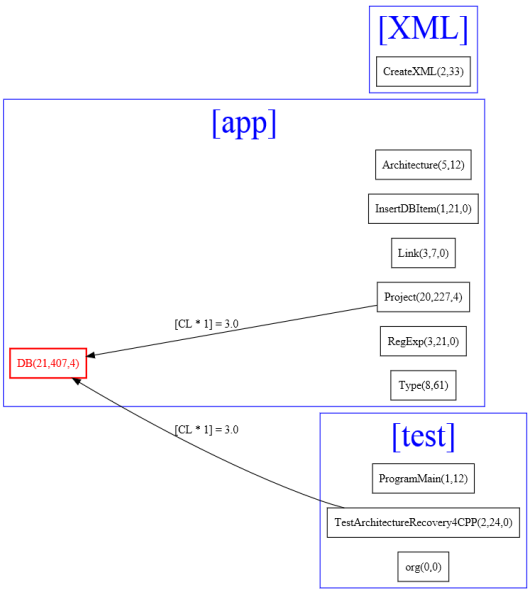

Figure 1. Package achitecture with code visualization [1]

III. THE EXTRACTION METHOD FOR PERFORMANCE IMPROVEMENT

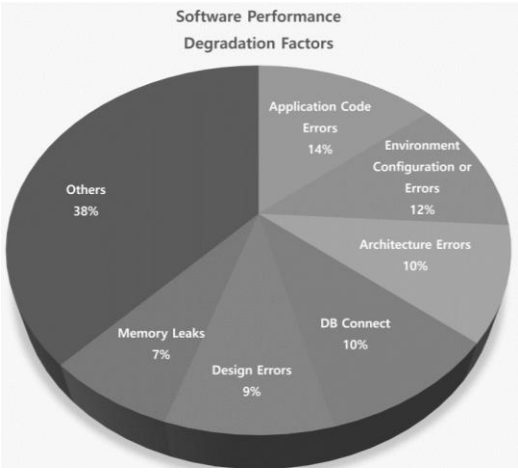

Figure 2. Software Performance Degradation Factors [3]

Figure 2 shows the causes of the Software performance degradation. According to 2005 information system management guidelines [3], Software performance degradation factors include environment problems such as insufficient memory problem and the Java virtual machine issues. However, 33% of the Software performance degradation is occupied with the Software Architecture and Source Code error. In the actual development field, it seems to match the key elements of the development for the company without considering problems due on the due date. In this current situation, it is necessary to extract the performance degradation factors, and also improve the performance of the software quality.

3.1 Degradation factor extraction method

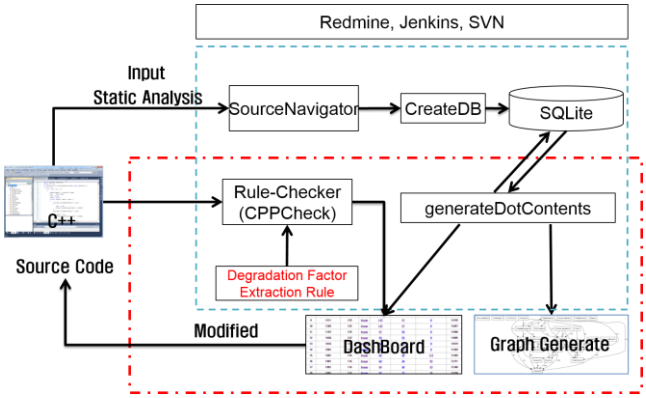

Figure 3. Degradation Factor Extraction Mechanism for Software Performance Improvement

Figure 3 show degradation factor extraction mechanism for software performance improvement. This mechanism is as follows; Enters the existing source code in the Source Navigator [4], extracts the source code for complete information, calculates quantitative index with the extracted information, and prints the Architecture View and Dashboard [5]. Finally, input the rule to define a performance degradation pattern in the regular expression, and then extract performance degradation with the Rule-Checker (CPPCheck [6]).

Software performance degradation factor can be divided into two different thing. The first is a degradation factor for the loop. The loop is repeated without reducing the unnecessary number of iterations. The second is a hinder for the control statement. Impediments to the control statement is generated when the multi- control statement with an unnecessary control structures and variables of the inner loop [7].

Table 1. Degradation factor extraction rule & Expected code

	Regular Expressions	**The expected code**
Loop	[a-z,A-Z,_]([a-z,A-Z,_,0-9])* \&lt ; ([0-9])+ ; [a-z,A-Z,_]([a-z,A-Z,_,0-9])* \+\+	int i = 0 ; i < 20 ; i++
Control Statement	[a-z,A-Z,_]([a-z,A-Z,_,0-9])* \< ([0-9])+ ; [a-z,A-Z,_]([a-z,A-Z,_,0-9])* \+\+ \) \{ if \(	int i = 0 ; i < 20 ; i++){ if(
	if \(([a-z,A-Z,_,0-9,\b,\s,\+,-,\/,*,\%%,-\>,\>,\<,\[,\]])* == ([0-9])+ \)	if(a == 1)

An example of a code extraction is expected over the rule pattern. Table 1 shows the regular expression rules for software degradation extraction. In the case of *the loop*, the internal condition of *for-loop* expression is a single variable to hold the pattern in 1 increments repeat. One of the variables are to spot the pattern when compared with the value from the *if* statement and if *a loop* in front of a case of a control statement with an if statement immediately after it exists.

3.2 Factors against performance degradation

Table 2 is an example of the modified code for performance improvement, which includes the performance impediments.

Table 2. Factors against performance degradation [6]

	Degradation code	Transformation code

Loop	int sum =0;
	for(i = 0 ; i<1000; i++){
	sum += array[i];
	}
		int sum = 0;
		for(i = 0; i<1000; i+=4){
		sum+= array[i];
		sum+= array[i+1];
		sum+= array[i+2];
		sum+= array[i+3];
		}

Control Statement	for (i=0;i<1000 ;i++){
	if(i & 0x01){
	do_odd(i);
	}else{
	do_even(i);
	}
	}
		for(i = 0;i<1000;i+=2){
		do_odd(i);
		do_even(i+1);
		}

	if(a==1){
	}else if(a==2){
	}else if(a==3){
	}else if(a==4){
	}.....
		switch(i){
		case 1:
		break;
		case 2:
		break;
		case 3:
		break;
		case 4:
		break;
		default :
		break;
		}

An example of a loop is the code that adds the value of the array during the *i-th* iteration. The changed code indicates the number of iteration of the loop reduced down to one-fourth of original by increments i by 4 which means that it makes computer to skip from i to i+3 process. Speed is getting better as long as the number of loops is reduced. If the *if-else* statement in the *for-loop* statement carried out in an odd or even number, this can be done the same as the original code if you remove the *if-else* statement to the variable I, and assign the value of i and i + 1 value. When multiple *if-else* statement, the condition value is a form of "variable == value" which can be changed by the *switch case* statement. Such changes may improve the speed by reducing the unnecessary repetition of the control statement.

IV. CASE STUDY

Figure 4 shows the execution screen of the RoboCAR simulator developed by SELab researcher of Hong-ik University. The simulator can easily control the RoboCAR to have four wheels using the script language, and perform the robot in virtual environment. It is also possible to build customization of the virtual environment, and to communicate the HiMEM modeling tool using TCP/IP [8].

Table 3 shows the result to remove the unnecessary control statements in *getAABB* method of *CODERoboCarbody* class. After the change works, we measure the average speed to perform 1,000 times repeated. In the result, we can get a speed increase 53.13% due to reduce the speed of the code from 0.000032ms to 0.000015ms.

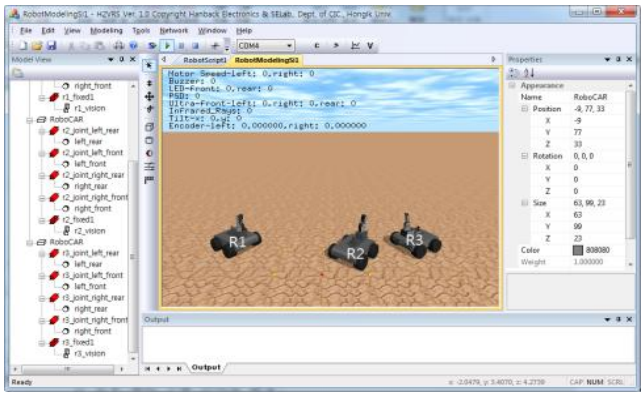

Figure 4. Our Simulation tool of Robocar[8]

TABLE 3. Removing the inner loop of the control statements

Removing the inner loop of the control statements	Before	for(int i = 0; i< m_meshCount[idx];i++){
		for (int j = 0;j = m_pTrimesh[idx][i].vertexCount*3 ;j++){
		if(0==i%3){
		maxX = max(maxX,m_pTrimesh[idx][i],vertices[j]);
		minX = min(minX,m_pTrimesh[idx][i],vertices[j]);
		}else if(1==i%3){
		maxY = max(maxY,m_pTrimesh[idx][i],vertices[j]);
		minY = min(minY,m_pTrimesh[idx][i],vertices[j]);
		}else if(2==i%3){
		maxZ = max(maxZ,m_pTrimesh[idx][i],vertices[j]);
		minZ = min(minZ,m_pTrimesh[idx][i],vertices[j]);
		}
		}
		}

	After	for(int i = 0; i< m_meshCount[idx];i++){
		for (int j = 0;j = m_pTrimesh[idx][i].vertexCount*3 ;j+=3){
		maxX = max(maxX,m_pTrimesh[idx][i],vertices[j]);
		minX = min(minX,m_pTrimesh[idx][i],vertices[j]);
		maxY = max(maxY,m_pTrimesh[idx][i],vertices[j+1]);
		minY = min(minY,m_pTrimesh[idx][i],vertices[j+1]);
		maxZ = max(maxZ,m_pTrimesh[idx][i],vertices[j+2]);
		minZ = min(minZ,m_pTrimesh[idx][i],vertices[j+2]);
		}
		}
		}

Table 4 shows the comparison between the control statement and the switch case statement. If we discover the multiple *if-then-else* codes, the code is translated to the *switch-case* statement. For example, the method named with "create" in *CODESignalCode* class has several *if-then-else* statements, which are manually changed like table4. To obtain a measurement result of the modified parts, we repeatedly performed 1,000 times to change the code in both the before and the after. In the result, we can get a speed increase 26.69% on average such that the before change is 0.00251ms, and the after change is a 0.000184ms.

Figure 5 is a part of the architecture view including speed information of the module. Like this, the developer or

Table 4. Comparison between the Control Statement and the Switch-Case Statement

	Before	After

Multiple If then else -> Switch Case	if (idx == 0){
	dBodySetPosition(m_visionID[idx],pos[0],
	pos[1],pos[2]);
	dMassSetZero(&mass);
	dMassSetBox(&mass, m, side[0], side[0],
	side[1],side[2]);
	dBodySetMass(m_visionID[idx], &mass);
	SetMass(&mass);
	}else if (idx == 1){
	dBodySetPosition(m_visionID[idx],pos[0]
	,pos[1]-0.2 * SCALE_FACTOR, pos[2]-0.07f *
	SCALE_FACTOR);
	}else if (idx == 2){
	dBodySetPosition(m_visionID[idx],pos[0],
	pos[1]* 0.35f * SCALE_FACTOR, pos[2] *
	0.31f * SCALE_FACTOR);
	dMatrix matrix;
	dRFromEulerAngles(matxrix,90*
	(M_PI/180.f), 0);
	dBodySetRotation(m_visionID[idx],
	matrix);
	dMass sub_mass;
	dMassSetZero(&sub_mass);
	dMassSetBoxTotal(&sub_mass, 0.001f,
	side[0],side[1],side[2]);
	dBodySetMass(m_visionID[idx],
	&sub_mass);
	}else if (idx == 3){
	dBodySetPosition(m_visionID[idx],pos[0],
	pos[1]* 0.17f * SCALE_FACTOR, pos[2] *
	0.44f * SCALE_FACTOR);
	
	dMass sub_mass;
	dMassSetZero(&sub_mass);
	dMassSetBoxTotal(&sub_mass, 0.001f,
	side[0],side[1],side[2]);
	dBodySetMass(m_visionID[idx], &sub_mass);
	}
		switch(idx) {
		case 0:
		dBodySetPosition(m_visionID[idx],pos[0],
		pos[1],pos[2]);
		dMassSetZero(&mass);
		dMassSetBox(&mass, m, side[0], side[0]
		, side[1],side[2]);
		dBodySetMass(m_visionID[idx], &mass);
		SetMass(&mass);
		break;
		
		case 1:
		dBodySetPosition(m_visionID[idx],pos[0],pos
		[1]-0.2 * SCALE_FACTOR, pos[2]-0.07f *
		SCALE_FACTOR);
		break;
		
		case 2:
		dBodySetPosition(m_visionID[idx],pos[0],pos
		[1]* 0.35f * SCALE_FACTOR, pos[2] *
		0.31f *
		SCALE_FACTOR);
		dMatrix matrix;
		dRFromEulerAngles(matxrix, 90 *
		(M_PI/180.f), 0);
		dBodySetRotation(m_visionID[idx], matrix);
		
		dMass sub_mass;
		dMassSetZero(&sub_mass);
		dMassSetBoxTotal(&sub_mass, 0.001f,
		side[0],side[1],side[2]);
		dBodySetMass(m_visionID[idx], &sub_mass);
		break;
		
		case 3:
		dBodySetPosition(m_visionID[idx],pos[0],pos
		[1]* 0.17f * SCALE_FACTOR, pos[2] *
		0.44f *
		SCALE_FACTOR);
		
		dMass sub_mass;
		dMassSetZero(&sub_mass);
		dMassSetBoxTotal(&sub_mass, 0.001f,
		side[0],side[1],side[2]);
		dBodySetMass(m_visionID[idx], &sub_mass);
		break;

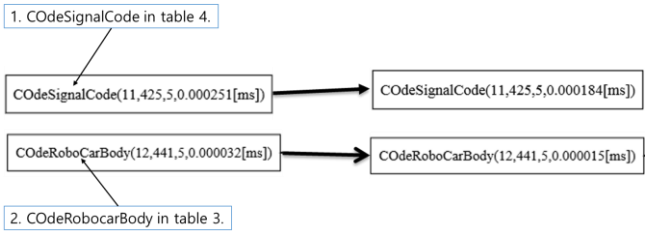

Figure 5. Architecture View for Class_name (#method, LOC of the class, #method over 25 LOC, the speed data of the module)

administrator obtains the related information in architecture view [5] to display the speed of each module.

V. CONCLUSIONS

In this paper, we show the degradation pattern of software performance such as unnecessary repetition, the loop control statements, and multiple control structures on a variable. We also extract threw degradation elements with applying the performance patterns in Rule-Checker using regular expression. We have to change the code to eliminate the unnecessary control code in the *loop* statements, and to change a multi-control with a *switch case* statement. These changes can get a speed increase for each 53.13% and 26.69%. Also, we visualize the speed information of module on architecture view. It can be achieved to improve the performance of the previous systems, and shows a bad coding habit to programmer. The future study is finding more performance degradation factors. Also, we will analyze and verify the performance of static analysis as well the dynamic analysis.

ACKNOWLEDGMENT

This work was supported by the Human Resource Training Program for Regional Innovation and Creativity through the Ministry of Education and National Research Foundation of Korea (NRF-2015H1C1A1035548) and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2013R1A1A2011601).

REFERENCES

[1] Nipa , 2013, "SW Engineering Center SOFTWARE ENGINEERING WHITE BOOK : KOREA 2013,"Nipa
[2] Henry H. Liu, 2009, "Software Performance and Scalability: A Quantitative Approach 1st Edition," Wiley.
[3] NIA, 2005, "Guideline for Performance Management of information System," NIA
[4] http://sourcenav.sourceforge.net/
[5] Geon-Hee Kang, R. Young Chul kim, Geun Sang Yi, Young Soo Kim, Yong B. Park, Hyun Seung Son, 2015, "A Practical Study on Code Static Anlysis through Open Source based Tool Chains," KIISE Transaction on Computing Practices, 21(2), pp. 148-153.
[6] http://cppcheck.sourceforge.net/
[7] Geon-Hee Kang, R. Young Chul Kim, Sang Eun Lee, Su Nam Jeon, 2015, "Extracting performance factors against performance degradation through code Visualization," Proc. 5th International Conference on Convergence Technology 2015, 5(1), pp. 276-277.
[8] Woo-sung Jang, Chan-Min Park, Cheul-Hee Lee, R. Young-Chul Kim, 2012, "A Study on test Case Extraction And Application For Intelligent transport RoboCAR Drive Control Verification," Proc. 38th Korea info. Process Society Fall Conf. 2012, 19(2), pp. 1452-1455.