

Comparison of Five Black-box Testing Methods for Object-Oriented Software

Kwang Ik Seo

Eun Man Choi

Dept. of Computer Engineering, Dongguk University, Seoul, Korea
{bradseo,emchoi}@dgu.ac.kr

ABSTRACT

As the size of software is getting huge, it is difficult for testers to check out all parts of source code in white-box style during integration testing or system testing period. Therefore functional test methods based on requirements information are frequently used in system level test. There have been a lot of test methods based on requirement specification. Each method has a different approach to specify software requirements. Test engineer should consider those various aspects of approaches and select proper black-box testing method to be applied. This paper presents the empirical comparison of major black-box testing methods and shows the different results by applying them to test a certain software system. The result shows that black-box testing methods check different levels of code construct. Test planer should consider the combination for the efficient test methods which combine extended use case test method and OCL test method.

Keywords: *Specification-based test, Comparison of testing methods, Black-box test, Performance of testing methods.*

1. INTRODUCTION

Software testing is used to evaluate the correctness, completeness and quality of developed computer software. There are two main approaches of software testing. One is white-box testing and the other is black-box testing. White-box testing method is used for logical and analytic test in unit test level. Meanwhile black-box testing is used in integration level or system level because black-box test does not need to look into source code but just need to execute a system by using input data and output result.

Even if we select a testing method among black-box testing methods, the test result will be different because

the technique specifying requirements and the method extracting test data are different in spite of the same black-box style. If we don't apply a proper test method for black-box testing, we can't trust the test result and it produces enormous loss due to a waste of time, manpower and money. Accordingly we need the various comparative studies of test techniques to apply properly in the testing field.

A couple of papers defined the elements to compare the performance or the efficiency and the method to build test case. These papers propose the method of comparison based on cost, efficiency, usefulness, and the number of faults[1][2][3]. Most researches showing the results of test methods comparison are related to statement coverage test, branch coverage test and data flow coverage test based on white-box test. The subject system in these papers experiment is also developed by procedural paradigm[4][5][6]. It is difficult to find out the comparison papers of black-box testing based on OO specification even though programming paradigm was shifted to Object-Oriented.

This motivation makes this paper execute five black-box testing methods based on requirements of two target systems developed by OOP and analyzes testing results. Because test case specification and checking parts of five testing techniques are all different, it is difficult to acquire a uniform of the comparative criterion. But the number of accessed variables or methods in an object can be applied to the criterion equally. Five black-box testing methods used for empirical comparison are use-case driven testing, black-box testing using collaboration diagram, testing using extended use-cases, testing using formal specifications(OCL or Object-Z). Two target systems are adopted to evaluate these five testing methods. One is ATM system and the other is the session scheduling system.

This paper is organized as follows. Section 2 of this paper introduces testing methods based on requirements. Section 3 describes how to test ATM system and session scheduling system by executing five different testing methods. Section 4 shows the comparison of testing method's approach. Section 5 presents analysis of coverage results in target systems. In section 6, we propose efficient compound mixed from test techniques to make coverage the largest. Our conclusions are presented in Section 7.

2. TEST METHODS BASED ON REQUIREMENTS

Along with information technology development, the complexity of software is increasing rapidly. Accordingly various requirements and modeling techniques are developed in order to help comprehend system and communicate opinion with others. Requirements specification is a baseline of software project. Therefore functional testing frequently uses various forms of requirements such as use-case diagram[4], OCL[5], collaboration diagram[6], Object-Z[7], extended use-case diagram[8], state transition diagram[9] and Petri-net[10]. How these different forms of requirement specification make effect on selecting test cases and testing results? That is big research question in this experiment.

Software paradigm was turned toward object-oriented development method from traditional software development. In these days object-oriented languages are widely used in implementing software systems. Accordingly in this paper, we choose to make experiments of testing methods closely related to object-oriented development methods and also pick systems implemented by JAVA programming language as a target system. In object-oriented point of view we shift out simple use case, extended use case, Object-Z mixed with state diagram, OCL devised to constraint UML.

UML is lovely used by OO developers for analyzing and modeling requirements. All of the testing methods compared in this paper are using UML specification to select test cases. Evaluating five black-box testing methods based on UML specification make clear in selecting UML based black-box testing. In our research two target systems are neither parallel system nor real-time system required to synchronous verification. Both are sequential systems operated a designed sequential task. In other words, these are sequential systems whose main functions are to control sequential transaction by a

state transition simulation and a user interface. Therefore, we would not consider the test methods designed to a parallel processing and a synchronous specification like Petri-net technique.

2.1. Use case Driven Testing

Test case extraction from simple use-case diagram[4] is shown in Figure 1. First, by using use-case diagram test engineers describe functions of a system and define the flow of events from those functions. Second, they redefine events flow with the graph which is predefined in natural language for composing possible scenarios. Last, test cases are extracted from scenarios with adding exceptions. In this method, algorithm in the system and interaction between program modules are not considered. Test engineers only forecast, define and verify possible errors.

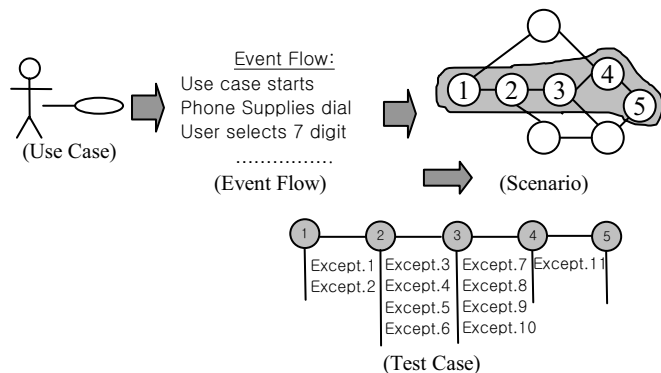


Figure 1. Process for use case driven testing

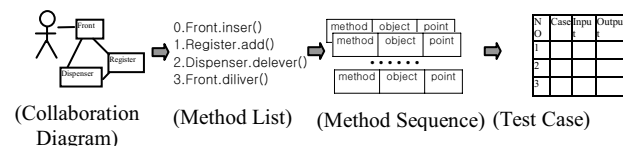


Figure 2. Process for testing using collaboration diagram

2.2. Black-box Testing Using Collaboration Diagram

Collaboration diagram in UML represents correlation and messages to display interaction between objects. Black-box testing method using collaboration diagram[7] begins to make function-centered collaboration diagram and then extracts test cases with correlation and message flow between objects. In other words, test engineer composes sequence of operation's

calling and defines input/output values and makes test cases for invoking series of message passing.

2.3. Testing Using Formal Specification

Object-Z was converted from Z, a formal specification language for describing object-oriented systems[10]. Test cases are extracted from Object-Z specification as displayed in Figure 3. First, the major classes are described with Object-Z and then Object-Z is transformed to state transition diagram along with transition paths of objects states. Finally, we make test scenarios from state transition diagram and select an input data and expected results for each scenario.

UML has a formal specification method to compensate modeling only using diagrams. Object Constraints Language is useful to present both responsibility and authority of objects clearly by using pre-condition and post-condition constraint. In order to build test cases from OCL we need to partition domain of functions to be tested and then express constraints in OCL. Next we analyze the relations between objects in the partitioned domain and divide each object's components such as attributes, initial values and variables. Finally, we arrange the components and the constraint of the objects to make test cases. Testing makes sure that the components satisfy their constraints.

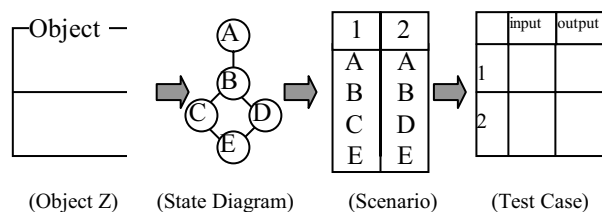


Figure 3. Test process using Object-Z specification

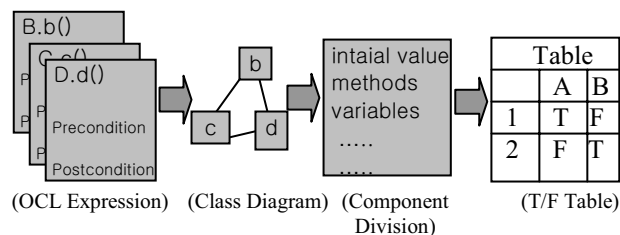


Figure 4. Process for testing using OCL

2.4. Testing Using Extended Use Cases

The test case extraction of extended use case test method is shown in Fig. 5. First, extended use case test

method makes use case. Second, we compose a scenario and extract classes or parameter used in classes. Third, we combine the scenario and parameter of classes and extract MM-path(Method/Message path). Finally we make test cases by setting input events and output events expected of MM-path from a scenario.

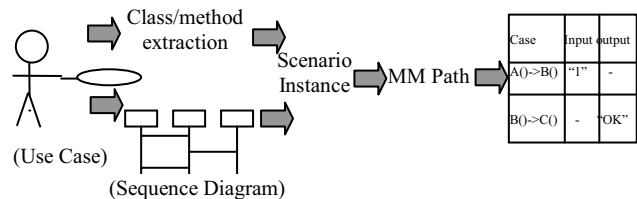


Figure 5. Process for testing extended use cases

3. EXPERIMENT OF TESTING ATM SOFTWARE

ATM(Automated Teller Machine) software is selected for this study and the system is implemented using Java language. Data transmission and function selection occur frequently between ATM system and a user. ATM system opens two accounts for a customer. A customer needs PIN and customer number for transaction like cash withdrawal.

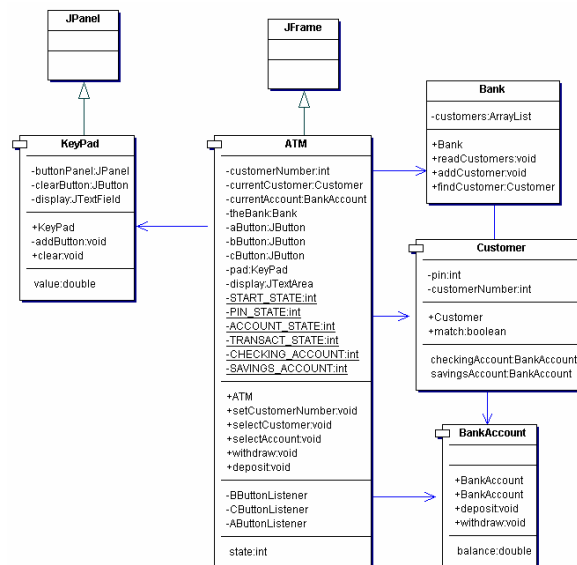


Figure 6. ATM class diagram

Figure 6 shows ATM software architecture. KeyPad class inherited from JPanel class makes user interface to input data. ATM class inherited from JFrame class which is a super class of ATM class builds a frame and

accesses KeyPad class, Bank class, Customer class and BankAccount class. Bank class uses Customer class.

3.1. Experiment #1: Use Case Driven Testing

In exterior user's point of view, use case diagram defines system functions to specify system requirements. Figure 7 shows a use case for 'Withdrawal function'. In case of withdrawal balance can be enough or shortage. Also we need to consider both normal case and exception case to make test cases. A test case for cash withdrawal is shown in Table 1.

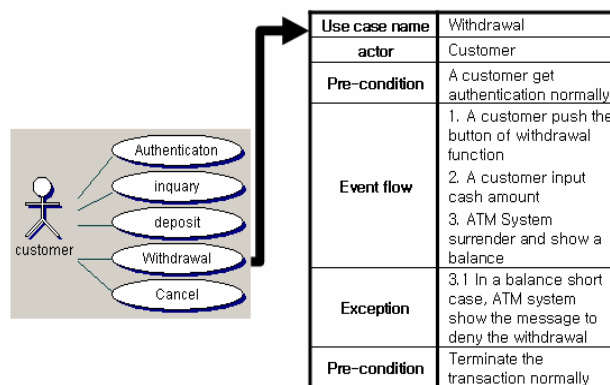


Figure 7. Use case for withdrawal

Table 1. Test cases for checking withdrawal

State	Input	Output	Balance
Normal	30\$	1. 30\$ cash	Before: 50\$
		1. remainder 20\$	/ After: 20\$
Abnormal	80\$	1. Balance short message	Before: 50\$
			/ After: 50\$

3.2. Experiment #2: Black-box Testing Using Collaboration Diagram

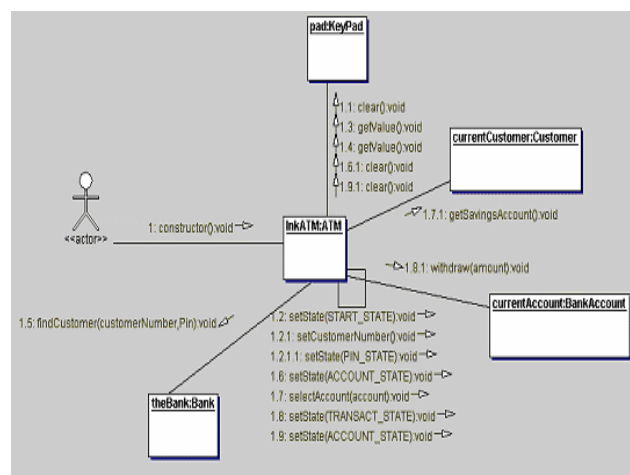


Figure 8. Collaboration Diagram

Collaboration diagram in Figure 8 displays the sequence that after a customer creates ATM object, ATM system prepares ATM functions and communicates messages between objects. Table 2 shows test cases for collaboration diagram which describes systems creating ATM object and preparing transactions. Also it describes the sequence of access between objects and messages to be collaborated. From collaboration diagram, we understand the methods of functions and expect input/output values. Test cases were built from compounding of those method and input/output data.

Table 2. Test cases extracting from collaboration diagram

No	Test cases	Input	Oracle
1	ATM.ATM()	Instrumented code	actionName:constructor linkEndObjectName:ATM
2	Pad.clear()	Instrumented code	actionName:clear() linkEndObjectName:KeyPad
3	ATM.setState(STAR T_STATE)	Instrumented code	actionName:setState(STAR T_STATE) linkEndObjectName:ATM
...	next :
16	currentAccount.withdraw()	Instrumented code	actionName: withdraw() linkEndObjectName: BankAccount
17	setState(ACCOUNT_STATE)	Instrumented code	actionName: setState(ACCOUNT_STATE) linkEndObjectName: ATM
18	Pad.clear()	Instrumented code	actionName: clear() linkEndObjectName: KeyPad
			next : NULL

3.3. Experiment #3: Testing using Object-Z specification

After generating Object-Z specification of target system, we draw state transition diagram about an important object. Then we make a scenario and expect input/output values from the sequence of state transition diagram. Figure 9 shows state transition diagram and scenario about a withdrawal and deposit function. Table

3 shows the test cases driven by state transition diagram of withdrawal object.

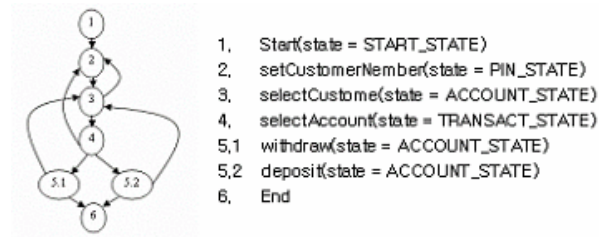


Figure 9. ATM State Transition Diagram

Table 3. Test cases based on Object-Z

	Input		Oracle	Remarks
	ID	PIN	Balance	
S C E N A R I O	1. normal withdrawal			Balance 0
	1	1234	Withdrawal: 300	
	2. normal deposit			Balance 0
	1	1234	Deposit : 3000	
	3. PIN Error			
	1	1234	Error Message	
O	4. Cancel a selection of Account			
	1	1234	Initial screen	
	5. deposit after withdrawal			Balance 0
	1	1234	Withdrawal:300 / Deposit: 200	200 decrease
	6. withdrawal after deposit			Balance 0
	1	1234	Deposit: 300 / Withdrawal: 100	200 increase

Table 4. Partition of object category

Class	LB (Low Boundary)	LB+1, HB-1	HB (High boundary)
ATM	1	N/A	N/A
KeyPad	1	N/A	N/A
Bank	1	N/A	N/A
Customer	1	N/A	N/A

Table 5. Test cases based on OCL

Test Script	Test Data	Oracle
pad.setValue()	Pstate1	F
	Pstate2	T
	Pstate3	F
	Pstate4	F
theBank.findCustomer(customerNumber, pin)-customer	Pstate1	F
	Pstate2	F
	Pstate3	T
	Pstate4	F
theBank.findCustomer(customerNumber, pin)<>NULL	Pstate1	T
	Pstate2	F
	Pstate3	F
	Pstate4	F
currentCustomer.getCheckingAccount()	Pstate1	F
	Pstate2	F
	Pstate3	F
	Pstate4	T

3.4. Experiment #4: Testing using OCL specification

Table 4 shows the partition of object category. The range of multiplicity depends on the system. But in this case only one customer can create a transaction at one time. Therefore LB(Low Boundary) is just 1.

Table 5 shows the final test cases derived from OCL specification. In ATM system, a state attribute has information of transaction about what ATM serves for customer. List of test script includes methods changing the value of state attributes. List of test data is the value of the state attribute. Value of Pstate1 is 'START_STATE', value of Pstate2 is 'PIN_STATE', value of Pstate3 'ACCOUNT_STATE' and value of Pstate4 'TARNSACTION_STATE'. So we can check that the value of states can changed correctly according to calling of a method.

3.5. Experiment #5: Extended Use Cases

For using extended use cases we should prepare scenarios with specific instance of use case. Then we extract the message paths of methods from scenario and map input/output value into use case instance. Table 6 shows scenario about the function of selecting "Check Account". Table 7 is test cases from extended use case test method about initial state of ATM. Initial state should be ready to receive customer ID and PIN.

Table 6. ATM Cash Withdrawal Use Case

Agent	Action
ATM	Customer number request
User	Customer number entry
ATM	PIN request
User	PIN entry
ATM	Account selection
User	Checking selection
ATM	Amount and transaction type request
User	Amount entry and transaction selection
ATM	Cash delivery
User	Cash withdrawal
ATM	Account selection
User	Exit
ATM	Customer number request

Table 7. Test cases based on extended use case

Test Case including Test Script	InputOutput	Note
ATM.ATM()->pad.clear()		Before data
AButtonListener.actionPerformed()->	"1"	input
ATM.setCustomerNumber()->		Data
pad.setValue()		input(click button No .1)
ATM.setState()->pad.clear()->	" "	After data
display.setText("Enter PIN A=OK")		input

4. RESULTS OF EXPRIMENTS

Most of the test case derivation methods affect a format of test case, amount of test data, coverage because of their unique test case extraction technique and approach. In this section we compare features of test derivation methods.

Figure 10 shows coverage of each test case method and domain of system during composing test cases. Simple use case driven testing method doesn't concern inside of the system at all. However, in case of testing method using extended use cases test engineers make use scenarios for selecting test cases. When test engineers find a fault, they make a specific test case by inspecting inside of the system such as source code in detail. For instance, if test engineers make test cases on a basis of certain scenario and find faults at node D in System Exterior Domain in Figure 11, test engineers would concern source code as faults related with node D and then they will build new test case again more specifically.

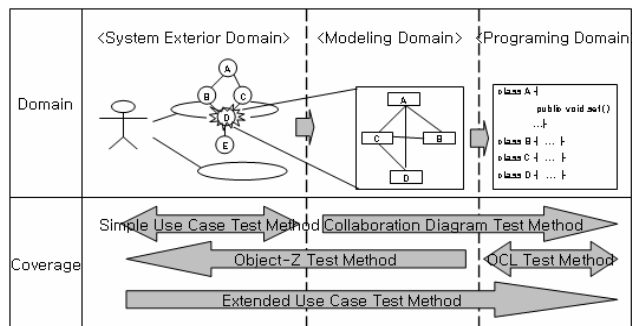


Figure 10. Approach features of test methods

In case of Object-Z test method we first understand the relationship of methods between objects in system and make scenarios to be tested finally. In Figure 11, available scenarios are two paths as shown in system exterior domain. So, we can test only two scenarios, (A, B, D, E) and (A, C, D, E). In case of OCL test method

the technique to derive test cases cannot be related to scenario at all. So to speak, test cases can be derived from only certain states, methods, member variables or relationship of classes. It means that the volume of test cases depends on how to partition a target and relationship.

Test case derivation methods have different points of system domain and system access direction during testing. For example, black-box testing using collaboration diagram and Object-Z start to test with the model domain. But levels of coverage are different so that after testing with collaboration diagram test method goes down into source code level as a program domain. But testing method with Object-Z goes up to functions of system level. Therefore if test document depend mainly on modeling specification language like class diagram and test level is a system functional level, Object-Z test method is better than others. Meanwhile, if test document depend mainly on modeling specification languages like class diagram and purpose of the testing is to test inner flow of program source code focusing certain value of data, testing method using collaboration diagram will be the best choice.

4.1. Coverage Analysis

For identifying characteristics of five black-box testing methods, we measure coverage of each testing scenario as standard for comparison. The scenario is that customer accesses ATM system, selects account, selects transaction and finishes the transaction. We measure the percentage of coverage invoked by test cases generated by each testing method.

Table 6 shows that the coverage of extended use case test method is 84%, collaboration diagram 44% and Object-Z test method 44%. The coverage of the extended use case test method is almost twice larger than Object-Z or collaboration diagram test method. The reason of big difference is that extended use case accessed logical flows as well as functions because the process of extended use case test method proceeds from black-box type to white-box type. That is to say, test direction is top-down process down from system level to source code level. On the contrary, the Object-Z or collaboration diagram test method's coverage is low because the method is only on the basis of black-box test of a test transition of state and a relation between objects. Also Table 6 shows that the percentage of coverage is 74% in OCL test method which partitions dependency and multiplicity of a relationship between classes, data members, methods and their combination.

In the case of simple use case diagram test method, the percentage of coverage is the lowest as 24% because this method is completely black-box test method which covers only the fact related to UI(User Interface).

Table 8. Coverage with the same scenario

	Simple Use Case	Collaboration Diagram	Object-Z	OCLExtended Use Case	
Target Variable(24)	9	10	11	21	23
Object					
Method(27)	3	12	11	16	19
The number among 51	12	22	22	37	42
Coverage(%)	24%	44%	44%	74%	84%

4.2. More Experiment

To acquire reliability of coverage analysis, we test the Session Scheduling system which is larger than ATM with all five black-box testing methods introduced in Section 2. Figure 11 is the class diagram of Session Schedule System. Session Schedule manager opens classes to session and students take courses. While students try to take courses, system checks a capacity of classes and completion of required subjects. Students can add, delete and refer to classes.

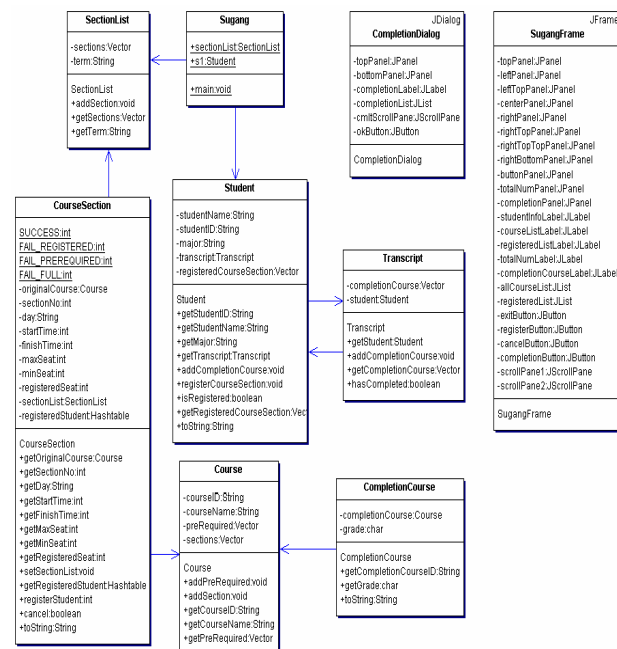


Figure 11. Class diagram of session scheduling system

The scenario to test session scheduling system is the following. Students access system, select the subject among the subject, and then browser shows selected

subjects list in right side. This scenario is used to each test derivation method equally. For this test, we found member variables and methods as test target and test data, and the result of coverage is shown in Table 9.

As shown in Table 9, data shows that simple use case's coverage is 41%, collaboration diagram's coverage 46%, Object-Z's coverage 48%, OCL's coverage 66% and extended use case's coverage 81%. Even though there is little difference from ATM system test's results, you can figure out that the coverage of simple use case test method and of extended use case test method in session schedule system, gets larger than ATM system. As we already mentioned the reason for these gaps in section 4, the reason is the different direction to access domain of the system and the different viewpoint of each method as well.

Table 9. Coverage Of Session Schedule System

	Simple Use Case	Collaboration Diagram	Object-Z	OCLExtended Use Case	
Target Variable(59)	33	25	23	47	53
Method(51)	12	26	30	26	36
Total Number(110)	45	51	53	73	89
Coverage (%)	41%	46%	48%	66%	81%

5. Combination for Coverage Maximization

In the table 8 and table 9, although the percentage figures are some different, the comparison of the coverage says the same result of the coverage percentage like "simple use case test method < collaboration diagram test method < Object-Z test method < OCL test method < extended use case test method". Especially OCL test method has approximately 70 % and extended use case test method has about 80%. Therefore we can refer to extended use case or OCL if you need to consider broader coverage as test purpose in the test planning stage. In addition, we can test software systems efficiently with combination of extended use case test method and OCL test method.

As you look into the Section 3 and Section 4, those test methods are some different aspects in the process of extracting the test cases between OCL and extended use case test method. For extended use case test method, it is black-box test based on the functions of the system by using a scenario which contains the test of a logical flow of the inner program in the system. Otherwise,

OCL test method isn't for the verification about the logical flow but it tests the relationship with member variables or methods or objects, which are components of the logical flow. Hence, first if we test a scenario instance which represents the logical flow by extended use case test method, and then traces the fault from the error spot by OCL Test Method when the error rises, it would be efficient test work.

6. CONCLUSION

It is necessary for developing quality software that developers make specification of user requirements before implementing a system. So, the method using specification is the easiest and plain way to test whether the system is built correctly or not. We tested ATM and Session Schedule System with five test methods, and compared features and coverage of each method. This empirical experiment shows how and why the coverage is different according to the point of view in test approach.

Moreover, we considered the combination for the efficient test methods which combine extended use case test method and OCL test method. If we use the data from this study about various black-box test methods as the required material and the results of this experiment is organized well, it will be helpful in planning and improving the performance of software testing

7. References

- [1] L.A. Clarke, A. Podgurski, D.J. Richardson, S.J. Zeil, "A Comparison of Data Flow Path Selection Criteria," *8th IEEE International Conference on Software Engineering*, pp.244-251, August 1895.
- [2] L. Lauterbach & W. Randall, "Experimental evaluation of six test techniques," *Proceedings of COMPASS'89*, Gaitersburg, MD, pp.36-41, June 1989.
- [3] S. Ntafos, "A Comparison of Some Structural Testing Strategies," *IEEE Transaction on Software Engineering*, Vol. 14, No.6, pp.868-874, June 1988.
- [4] Dave Wood & Jim Reis, "Use Case Derived Test Cases," *STAREAST on Software Quality Engineering Conference*, 1999.
- [5] Eun Man Choi, "Generating Test Cases for Object-Oriented Design Specification Described by OCL," *Journal of Korea Information Processing Society*, Vol.8-D, No.6, pp.843-852, December 2001.
- [6] Aynur Abdurazik & Jeff Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation," *The Third International Conference on the Unified Modeling Language (UML'00)*, October, York, UK, pp.383-395, 2000.
- [7] Chun-Yu Chen, Constructing Usage-Based Testing On Object-Z Formal Specification Based Specification, *Ph.D. Dissertation*, Auburn University, 1999.
- [8] Eun Man Choi, "Use-Case Driven Test for Object-Oriented System," *Proceedings of the IASTED International Conference*, ACTA Press, August, pp.164-192, 2001.
- [9] Y. G. Kim, H. S. Hong, D. H. Bea & S. D. Cha, "Test cases generation from UML state diagrams," *IEEE Transaction on Software Engineering*, Vol. 164, No. 4, pp.187-192, August 1999.
- [10] S. Ramaswamy, "A Petri net based approach for establishing necessary software design and testing requirements," *System, Man and Cybernetics*, 2000 IEEE International Conference, Vol.4, pp.3087-3092, October 2000.
- [11] P. Hsia, "Formal Approach to Scenario Analysis," *IEEE Software*, Vol.11, No. pp.33-41, 1993.
- [12] K. Koskimies & H. Mossenbok, "Scene: Using Scenario Diagram and Active Text for Illustrating Object-Oriented Programs," *Proceedings of 18th International Conference of Software Engineering*, pp.366-375, March 1996.
- [13] K. Yukse, S. Dupont, D. Harnoir and C. Froidure, "FTTx automated test solution: Requirements and experimental implementation," *IEE Magazine Electronics Letter*, Vol.41, No.9, pp.546-547, 2005.
- [14] R. Boddu, G. Lan, G. Mukhopadhyay, B. Cukic, "RETNA: from requirements to testing in a natural way," *12th IEEE International Requirements Engineering Conference*, pp.262-271, 2004.
- [15] P. Jorgensen, "Object-Oriented Integration Testing," *Communications of the ACM*, Vol.37, No. 9, pp. 30-38, 1994.
- [16] J. McGregor and T. Korson, "Integrated Object-Oriented Testing and Development Process," *Communications of the ACM*, Vol.37, No.9, pp. 57-77, 1994.
- [17] C. Lott, A. Jain, S. Dalal, "Advances in Model-Based Testing(A-MOST2005): Modeling requirements for combinatorial Software testing," *ACM SIGSOFT Software Engineering Note*, Preceedings of the first international workshop on Advances in model-based testing A-MOST '05, Vol.30, pp.1-7, 2005.
- [18] E. Weyuker and B. Jeng, "Analyzing Partition Testing Strategies," *IEEE Transactions on Software Engineering*, Vol.17, No.7, pp.703-711, 1991.
- [19] R. Lutz, I.C Mikulski, "Requirements discovery during the testing of safety-critical software," *Software Engineering 25th IEEE International Conference*, pp.578-583, 2003.
- [20] S. Berner, R. Weber, R.K. Keller, "Requirements & testing: Observations and lessons learned from automated testing," *Proceedings of 27th international conference on software Engineering*, pp.571-579, 2005.
- [21] D. Eshelman & M. Klafter, "Activity diagram – a requirements distribution process for test equipment development," *IEEE AUTOTESTCON 2004 Proceedings*, pp.37-43, 2004.