REFERENCES

(1) Perry, D. E. and Kaiser, G. E. Adequacy Testing and Object-Oriented Programming. Journal of Object Oriented Programming, 2,5(1990):13-19.

(2) Smith, M. D. and Robson, D. J. Object-Oriented Programming – the Problems of Validation. Proceedings of Conference on Software Maintenance, San Diego, CA USA, November 26-29, 1990:272-281.

(3) Firesmith, D. G. Testing Object-Oriented Software. Proceedings of TOOLS, March 19, 1993.

(4) Barbey S. and Strohmeier, A. The Problematics of Testing Object-Oriented Software. Proceedings of SQM '94 Second Conference on Software Quality Management, Edinburgh, Scotland, UK, 1994:411-426.

(5) Berard, E. V. Issues in the Testing of Object-Oriented Software. Proceedings of Electro'94 International, 1994:211–219.

(6) Hayes, J. H. Testing of Object-Oriented Programming Systems (OOPS): A Fault-Based Approach. Proceedings of the International Symposium on Object-Oriented Methodologies and Systems (ISOOMS), Palermo, Italy, September 1994:205-220.

(7) The Object Management Group. OMG Unified Modeling Language Version 1.4. Available from http://www.omg.org [2007, 1 January].

(8) Binder, R. V. Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley, 1999.

(9) The Object Management Group. OMG XML Metadata Interchange (XMI) Version 1.2. Available from http://www.omg.org [2007, 1 January]

(10) Spinellis, D. On the Declarative Specification of Models. IEEE Software, 20,2(2003):94-96.

(11) Meyer, B. Object-Oriented Software Construction, 2nd Edition, Prentice Hall, 1997.

(12) Voegele, J. Programming Language Comparison. Available on http://www.jvoegele.com/software/langcomp.html [2007, 1 January]

(13) LaLonde, W. and Pugh, J. Subclassing != subtyping != is-a. Journal of Object Oriented Programming, 3,5(1991).

(14) Taivalsaari, R. On the notion of inheritance, ACM Computing Surveys, 28,3(1996):438–479.

(15) Offutt, J.,Alexander, R.,Wu, Y.,Xiao, Q., and Hutchinson C. A Fault Model for Subtype Inheritance and Polymorphism. Proceedings of the 12th IEEE International Symposium on Software Reliability Engineering (ISSRE '01), Hong Kong, PRC, November 2001:84-95.

(16) Alexander, R. T., Offutt, J., and Bieman, J. M. Syntactic Fault Patterns in OO Programs. Proceedings of the 8th International Conference on Engineering of Complex Computer Software (ICECCS '02), Greenbelt, MD, November 2002.

(17) Harrold, M. J., McGregor, J. D., and Fitzpatrick, K. J. Incremental Testing of Object-Oriented Class Structures. Proceedings of the 14th International Conference on Software Engineering, Melbourne, Australia, 1992:68-80.

(18) Smith, M. D. and Robson, D. J. A Framework for Testing Object-Oriented Programs. Journal of Object-Oriented Programming, 5,3(1992):45-53.

(19) Jorgensen P. C. and Erickson, C. Object-Oriented Integration Testing. Communications of the ACM, 37,9(1994):30-38.

(20) Johnson Jr., M. S. A Survey of Testing Techniques for Object-Oriented Systems. Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, Toronto, Ontario, Canada, 1996.

(21) Binder, R. V. Testing Object-Oriented Systems: A Status Report. American Programmer, 7,4,(1994).

(22) Cheatham, D. J. and Mellinger, L. Testing Object-Oriented Software Systems. Proceedings of the 1990 ACM Annual Conference on Cooperation, Washington, D.C., United States, 1990:161-165.

(23) Kung, D., Gao, J., Hsia, P., Toyoshima, Y. and Chen, C. A Test Strategy for Object-Oriented Programs. Proceedings of the 19th International Computer Software and Applications Conference (COMPSAC'95), Dallas, Texas, August 09-11, 1995.

(24) Labiche, Y., Thevenod-Fosse, P., Waeselynck, H. and Durand, M.-H. Testing Levels for Object-Oriented Software. Proceedings of the International Conference on Software Engineering, Limerick, Ireland, June 4-11, 2000:136-145.

(25) Tai, K. C. and Daniels, F. J. Test Order for Inter-Class Integration Testing of Object-Oriented Software. Proceedings of the 21st Annual International Computer Software and Applications Conference (COMPSAC '97), Washington, DC, USA, August 13-15, 1997:602-607.

(26) Fraikin, F. and Leonhardt, T. SeDiTeC – Testing Based on Sequence Diagrams. Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE), Edinburgh, Scotland, UK, September 2002:261-266.

(27) Chen, H. Y., Tse, T. H., Chan, F. T., and Chen, T. Y. In Black and White: An Integrated Approach to Class-Level Testing of Object-Oriented Programs. ACM Transactions on Software Engineering and Methodology, 7,3(1998):250–295.

(28) Chen, H. Y., Tse, T. H., and Chen, T. Y. TACCLE: A Methodology for Object-Oriented Software Testing at the Class and Cluster Levels. ACM Transactions on Software Engineering and Methodology, 10,4(2001):56–109.

(29) Chen, H. Y. An Approach for Object-Oriented Cluster-Level Tests Based on UML. Proceeding of the IEEE International Conference on Systems, Man and Cybernetics, 2003.

(30) Abdurazik J. and Offutt, J. Using UML Collaboration Diagrams for Static Checking and Test Generation. Proceeding of The Third International Conference on the Unified Modeling Language (UML'00), York, UK, October 2000:383-395.

(31) Alexander, R. T. and Offutt, A. J. Criteria for Testing Polymorphic Relationships. Proceedings of the International Symposium on Software Reliability and Engineering (ISSRE00), IEEE Computer Society, San Jose CA, 2000:15-23.

(32) Alexander, R. T. and Offutt, A. J. Analysis Techniques for Testing Polymorphic Relationships. Proceedings of the Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00), Tokyo, Japan, 2000:172-178.

(33) Jin, Z. and Offutt, A. J. Coupling-Based Integration Testing. Proceedings of Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '97), Montreal, Canada, October 1996:10-17.

(34) Beizer, B. Software Testing Techniques, 2nd Edition, Van Nostrand Reinhold Co., 1990.

(35) Andrews, A., France, R., Ghosh, S. and Craig, G. Test Adequacy Criteria for UML Design Models. Software Testing, Verification, and Reliability Journal, Volume 13, Number 2, June 2003.

(36) Wu, Y., Chen, M., and Offutt, J. UML-based Integration Testing for Component-based Software. Proceedings of the 2nd International Conference on COTS-Based Software Systems (ICCBSS), Ottawa, Canada, February 2003.

(37) Briand, L. and Labiche, Y. A UML-Based Approach to System Testing. Journal of Software and Systems Modeling (Springer), 1,1(2002).

(38) Offutt J., and Abdurazik, A. Generating Tests from UML Specifications. Proceedings of the 2nd International Conference on the Unified Modeling Language (UML'99), Fort Collins, Colorado, United States, October 1999.

(39) Weyuker, E. J. The Evaluation of Program-Based Software Test Data Adequacy Criteria. Communications of the ACM 31,6(1988).

(40) Graham, S. L., Kessler, P. B., McKusick, M. K. gprof: a Call Graph Execution Profiler. Proceedings of the 1982 SIGPLAN symposium on Compiler construction, Boston, Massachusetts, United States, 1982.

(41) Eclipse Foundation. AspectJ Project Available from http://eclipse.org/aspectj/ [2007, 1 January].

(42) Interface21. Spring Framework Available from http://www.springframework.org [2007, 1 January].

(43) Apache Jakarta Project. Byte Code Engineering Library Available from http://jakarta.apache.org/bcel/ [2007, 1 January].

Appendices

Appendix A

Publication in ICEP 2004

An Instrumentation Model for Supporting Software Testing Based on UML Sequence Diagrams

Siros Supavita and Taratip Suwannasart
Department of Computer Engineering, Faculty of Engineering
Chulalongkorn University, Thailand
Siros.S@student.chula.ac.th, Taratip.S@chula.ac.th

Abstract

As UML Sequence Diagram usually represents an interaction of objects in object-oriented software in a form of a sequence of messages sending between the objects, it is a useful source as the design specification for the implementation. Likewise, testing, especially integration and system testing, potentially benefits from using it as a source of test specification. Whereas the main purpose of the testing is to verify whether the implementation conforms to the design, message sending sequence of the implementation is required to be compared to the one from the design. While the expected message sending sequence is extracted from UML Sequence Diagram, the actual message sending sequence is derived through the instrumentation of test execution. This paper presents a model for representing both message sending sequences, along with the basic guidance of how to apply the model in software testing, as test oracle and test coverage.

1. Introduction

While object-oriented paradigm provides features, like inheritance and polymorphism, that help developers to easily solve problems, it poses difficulties in testing. As the features are specific to the paradigm, testing techniques for structural programming paradigm are not adequately appropriate. Although some techniques, particularly functional testing techniques, may be applicable [1], most are usually not. An empirical study [2] shows that programs designed with object-oriented paradigm usually result in many operations with only simple intraprocedural control flow. This is significantly different from structural programs; therefore, control flow and data flow based testing techniques, which are originally designed for the structural design principle, is not suitable [3]. Moreover, unit testing in object-oriented program is inseparable from integration testing [4].

As UML (Unified Modeling Language), as a modeling language for object-oriented paradigm, is gaining more popularity, it becomes the essential part of many object-oriented software development projects. With its standard notations and semantic, designers can use UML to model their design in an expressive way, and also communicate to others effortlessly. Beside design and implementation, testing based on the model is very desirable. The goal is to verify conformance of the implementation against the design model. Nevertheless, there is no standard way of using UML notations in models and diagrams. Testability requirement for the model must be established so that the notations are uniformly used to produce test-ready model. Several researches have defined usages for their specific test approaches. While some focus on system-level specification [5,6], some focus on internal interaction or implementation [7,8,9,10].

Keeping the issues about object-oriented software testing and testing based on UML in mind, we are working on defining a test approach based on message sending sequence, modeled in UML Sequence Diagram. We focus on testing on system or integration level, as UML Sequence Diagram usually represents interaction of a system or a subsystem. This paper presents a model for representing message sending sequence which is an essential part of the test approach. Section 2 gives an overview of the test approach. Section 3 shows related work. Concept of the model is discussed in section 4, while section 5 explains the model elements. Section 6 gives an example of the model and the application of the model is given in section 7. Finally, section 8 concludes the paper.

2. Testing of Message Sending Sequence

2.1 Message Sending Sequence

Objects or components in object-oriented software interact by sending messages to each other. It is said that a sender object sends a message to a receiver object which is similar to the sender object calls an operation on the receiver object. A message sending sequence is an ordered sequence of message flying between objects in the interaction.

"Message" is a design term and is usually in higher level and conceptual; hence, it can be interpreted into several different meanings in implementation. "An object sends a message to another object" can be interpreted as "an object calls an operation on another object", "an object posts an event which causes another object, as an event handler, to be executed", or even "an object asynchronously calls an operation on another object". In our current work, we

consider only message which is a synchronous operation call.

2.2 Test Approach

From an interaction diagram like UML Sequence Diagram, message sending sequence is defined as the design specification. Programmers use the message sending sequence as the guidance of their implementation of the interaction. For test purpose, it is treated as the expected message sending sequence using as a part of test oracle, in addition to the output. The implementation under test must perform message sending equivalent to the expected message sending sequence.

According to the test approach, instrumentation of test execution is required to capture the actual message sending occurs under the execution. As stated earlier, it is compared to the expected message sending sequence to verify conformance of the implementation against the design model. An overview of how the approach works is shown in Figure 1.

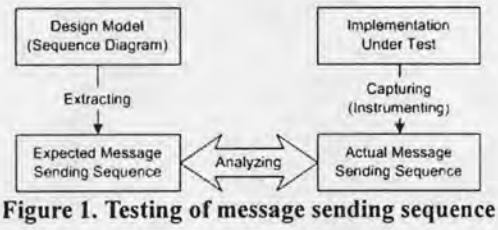

Figure 1. Testing of message sending sequence

We are currently working on defining the systematic procedure to verify the execution result (the actual message sending sequence) against the expected result (the expected message sending sequence). There are issues that need to be considered, regarding message sending sequence analysis. Ideally, the expected and the actual message sending sequence must be identical. Nonetheless, the interaction diagram, as the source of the expected message sending sequence does not usually contain interaction in a great detail like the implementation. Designers ordinarily omit the interaction with objects of common classes, like String in Java, or their own utility or framework classes. As a result, the actual message sending sequence is evidently nonidentical to the expected message sending sequence. Furthermore, polymorphism, which is a specific feature to object-oriented paradigm, is another important issue for sequence comparison.

2.3 Scope

Although our approach aims generally to UML Sequence Diagram, it apparently has limited range of usage. As stated earlier, our focus is on message which is a synchronous operation call, even though UML Sequence Diagram also supports other types of message, like an asynchronous call.

Another limitation lies in the way the interaction diagram is written. The previous section shows that the interaction diagram usually omits some classes, resulting in a gap between the interaction diagram and the implementation. As the gap grows wider, more messages in the actual message sending sequence are missing from the expected message sending sequence. Hence, the accuracy of message sending sequence verification is lessened. This becomes even worse when the diagram is written for high-level design where many parts of the interaction have been omitted from the diagram. Thus we currently limit the source of the expected message sending sequence to only the detail-level design diagram which reflects the main portion of the implementation.

3. Related Work

Fraikin et al. presented a tool, named SeDiTeC [9], to generate test for Java program from UML Sequence Diagram. UML Sequence Diagrams are taken as the test specification and test drivers are generated according to the interaction specified in the diagram. Several diagrams can be combined to form a test scenario; as a result, it allows preparation step, test execution, and result verification step to be written in separated diagrams. Test data is supplied as parameters of operation calls which are identified in the diagrams. Moreover, test stub class can be created for the incomplete implementation.

The test execution is determined as pass or fail by comparing the execution with the Sequence Diagram. The order of call, the object identity, input parameters, and return values specified by the testers, are used for the comparison. When the design model does not fully represent the implementation, the testers are required to identify whether the test is passed or failed. However, it does not provide support for polymorphic interaction.

4. Concept of the Model

4.1 Basic Requirement

Our goal is to have the expected message sending sequence and the actual message sending sequence represented on the same model to ease the sequence verification. The proposed model must have all common features of both message sending sequences, while must also avoid contradiction or confusion from both message sending sequence.

4.2 Instrumentation Model Requirement

Basic requirement of the instrumentation model is that the model must represent message sending sequence between objects which occurs in an execution of object-oriented

software. The model is significantly different from the instrumentation models which capture only function entry/exit in structural program, since object-oriented software makes uses of objects. An object is a cohesive piece of data and operations that manipulate the data. It is different from a structure type in structural programming which contains only data; the manipulation operations live apart. Moreover, each object has an identity of its own which distinguishes itself from other objects. Two objects, although contain identical values of data members, have different identities; hence, they are not identical. This difference is the major requirement of the model.

As polymorphism is a specific feature of object-oriented paradigm, discovering faults related to polymorphism is the main purpose of an object-oriented testing technique. As a consequence, the model must be capable of representing polymorphic server situation.

Beside the issues addressed above, the requirement is similar to usual function entry/exit instrumentation models. The model must be able to address call hierarchy, called function name and arguments (to identify the specific function), and context of the calling (to identify where the function is called).

4.3 Design Model Requirement

The model, which satisfies requirement in the previous section, is also capable of representing the message sending sequence from the design model, as the general information about the message sending sequence (message sender, message receiver, message action, and information about the operation and class where the execution occurs) are similar. Although for now we design the model to support message sending sequence comparison between the execution and the design model, the model should be extensible to support further analysis, for example return condition (usual return or exception thrown), and returned value verification. Since we do not yet complete the requirement for the analysis, this part of the model must be open for possible future extension.

4.4 Assumption and Restriction

First assumption is about the interpretation of message sending as described earlier; only the message sending which is an operation invocation is supported. Another assumption is about thread of execution. It is allowed to present concurrent execution in a single UML Sequence Diagram, by using notations like asynchronous call, return call, and object lifeline. However, testing for concurrent execution is complicated. Specific test techniques and instrumentation techniques are required to support concurrency. As a result, our current model supports only a single thread of execution for both the actual execution and the design model.

As UML Sequence Diagram allows guard conditions to be attached to messages in a diagram, a single Sequence diagram can represent a main scenario with several alternatives. Our model is a representative of one scenario; therefore, guard conditions are not currently considered. Further extension of this model, however, might include guard conditions for analysis purpose.

5. Design of the Model

Although UML provides semantic that covers wide array of object-oriented modeling including structural model and behavioral model, the structure of message sequence in UML, where all messages are grouped into an ordered list, is not appropriate to represent execution sequence in our case. Our model has a structure similar to dynamic call graph [11], which does not share a node when there is more than one call to an operation.

The model is presented in class diagram as shown in Figure 2. In the subsequent subsections, each element in the instrumentation model is discussed and possibly compared to UML Sequence Diagram element to give an example of how the design model might be extracted and compared to the execution.

5.1 ExecutionContext

An ExecutionContext element holds information about an execution of an operation under a specific circumstance. An execution may send messages to instances and results in executions in other contexts. Therefore, a context can be a message sender, a message receiver, or both.

As a message sender, a context has a reference to an ordered list of messages it sends. The messages are sorted in chronological order. Each message represents call and implicit return of the call. Using depth first traversal on the context hierarchy yields the sequence of message sending for entire interaction in timely order.

For the execution model, there are 2 aspects of operation attached to execution context, a direct associated operation in the execution context as an operation actually executed, and an operation associated to the stimulus message as an operation the sender intends to invoke. This is for representation of polymorphism in the execution. For the design model where polymorphic interaction is usually not explicitly presented, both operations are identical.

Unless the execution is on class operation, an execution context must be associated to an instance. The instance must be an instance of the class where the operation of the execution context and the operation of the stimulus message are defined or inherited. This is not an issue in some object-oriented programming languages (i.e. Java, C++), for they perform static type checking which already prevents undeclared operation calls on an object [12].

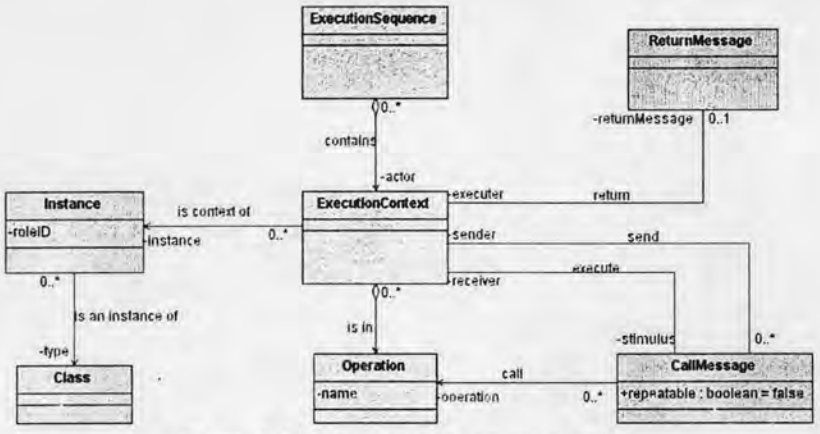

Figure 2. Message sending sequence model

5.2 Instance

An Instance element is a representative of an object in object-oriented program. It is similar to ClassifierRole in UML. The major difference is the association with type information. ClassfierRole can have more than one base Classifier, because an object in the collaboration can play roles of several Classifiers. However, an object in the implementation must have a type as a concrete class.

In an interaction, several instances of the same class possibly play different roles. It is important for the model to identify instance in the execution context, as the verification needs to compare the instances of the sender and the receiver of each message as well as the message itself. In the design model, instances are identified by assigning a role to each instance. The instrumentation model must also support instance identification by, for example, using generated object runtime ID. Although mapping between roles in the design and the execution is not possible, improper instance usage is recognizable.

5.3 CallMessage

A CallMessage element is a representative of a message sent from a sender object to a receiver object. As stated earlier, our focus is on an operation call message, which is similar to CallAction in UML semantic. A message has references to its sender and its receiver, which are both ExecutionContext elements. The sender execution context is the context where the message is issued, while the receiver execution context is the context that the receiver instance executes as the effect of the message. As an execution context element has a reference to an operation it is associated to, we can trace from a message to the operation which is the origin of the message and the operation which is the target of the execution.

In addition to the context of the sender and the receiver, a CallMessage element also has a reference, named "call", to an operation as the target of the call. The purpose of the reference is to accommodate polymorphic interaction. Along with the operation associated to the execution context of the receiver, the call association on the call message is the key to reveal defects in polymorphic implementation.

5.4 ReturnMessage

A ReturnMessage element contains information about return value or return condition. As stated earlier, the return information from both the design model and the instrumentation model are important if we want to perform analysis on the execution result or condition, in addition to message sending sequence. For design model, this element may contain information specified in the model as return message of the execution, if any. For the execution model, it may represent return value or condition captured from the execution, probably the thrown exception. However, we do not have a particular requirement on this issue yet, so the element is currently empty and optional.

5.5 ExecutionSequence

An ExecutionSequence element is a container element which wraps all other elements. It has an association to an execution context, which is the context of the actor of the interaction. The actor, as the stimulus, initiates the message sequence by sending messages to other objects. This element may contain other information which is applied to all elements in the interaction as global attributes of the interaction; for example, an identifier which is assigned to a particular execution scenario for further reference or the condition of the scenario collected from guard conditions of the diagram.

5.6 Classifier and Operation

Though sharing the same name, the Classifier element in this model is different from Classifier in UML model. In UML, a Classifier element is a base element for others, like Class, Interface, Components etc., providing general attributes to support different aspects of modeling. In our model, Classifier, as a base class for Class, Interface, and DataType elements, only provides the "name" attribute common to all of its subclasses. The only purpose of Classifier element in our model is to provide a common structure for its subclasses. There is only one usage of Classifier, as a type of each formal argument of an operation.

An operation is identified with its signature which includes its name, the list of its arguments along with their types, and the class where it is declared. The class diagram of Classifier and Operation is shown in Figure 3.

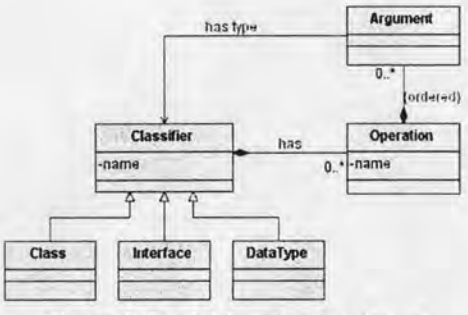

Figure 3. Model for structural part

5.7 Class, Interface, and DataType

As described in previous section, Class, Interface, and DataType elements are subclasses of Classifier element. They inherit an attribute, name, from Classifier element. The attribute represents name of class, interface, and data type respectively.

We designed these elements based on well-known object-oriented programming languages, Java and C++. Class element is apparently required as these languages are class-based. These languages, while provide support for most of object-oriented features like inheritance and dynamic binding, are classified as hybrid [12]. They do not require everything to be an object, so there are pieces of data that are defined as primitive types, which are represented by DataType elements in our model. In addition, Java has an additional construct, interface, which is considerably different from class in many senses; as a consequence, Interface element is designed to support this construct.

6. Example

This section shows how the model looks like for a given example of UML Sequence Diagram and a snippet of Java code. While the model is capable of representing complex diagrams and execution, the example is rather simple for understandability.

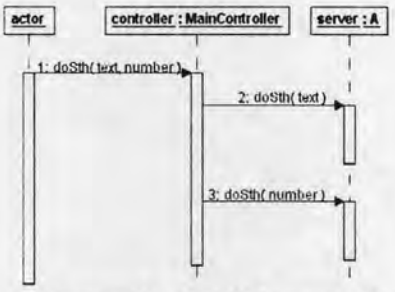

Figure 4. An example of UML Sequence Diagram

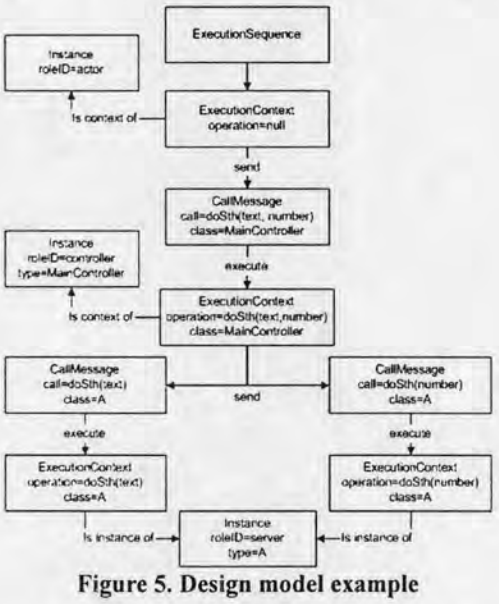

Figure 5. Design model example

Figure 4 shows a simple example of Sequence Diagram. An actor sends a message to controller, an instance of class MainController. The instance, in turn, sends 2 messages to server, an instance of class A; as a result, 2 operations on server, doSth(text) and doSth(number), are executed respectively. The message sending sequence model extracted from the diagram is shown in Figure 5. The example model shows that in the design model, as previously discussed, the operation associated to an execution context is identical to the operation associated to its stimulus message. Moreover, how overloading methods are modeled is also shown in the example model.

Figure 6 shows a snippet of Java code which is an example of implementation of the diagram in Figure 4. When the controller instance is instantiated with an instance of class A, the message sending sequence captured from the execution is identical to the one in Figure 5. However, it is slightly different, when an instance of any subclass of class A is supplied for the instantiation. Figure 7 shows the part

that is different from Figure 5 with the different elements highlighted, for the case where an instance of class B, as a subclass of class A, is supplied rather than an instance of class A.

```
class MainController {
    protected A server;

    public MainController(A server) {
        this.server = server;
    }

    public void doSth(String text, int number) {
        ...
        server.doSth(text);
        ...
        server.doSth(number);
        ...
    }
}
```

Figure 6. Java code example

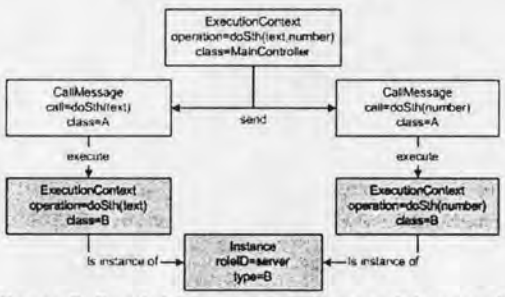

Figure 7. Partial instrumentation model example

7. Application

The model presented in this paper can be applied to several tasks in testing. Application of the model as test oracle, as emphasized in this paper, is the most important one. This model provides fundamental for defining test generation approach, where execution of generated test case yields the message sending sequence equivalent to the model.

Besides, the model is also applicable as test coverage model. When the Sequence Diagram contains branch conditions, basic coverage criteria similar to control flow based testing can be applied. As the model is capable of representing polymorphic interaction, it is also possible to define coverage criteria based on polymorphism for the interaction.

8. Future Work and Conclusion

In this paper, we propose a model for representing message sending sequence of object-oriented software. The major purpose of the model is to support testing based on message sending sequence which we are currently working on. We plan to implement extraction of the expected message sending sequence from XMI format which is a standard form of representation of metadata model, including UML, in XML. For the actual message sending sequence, Java is our target of implementation. Next, the verification rules for message sending sequence are established and evaluated with several patterns of interaction. After we accomplish this step, test environment based on the test approach will be developed. The environment will allow testers to automatically generate test cases based on UML Sequence Diagrams written during design.

9. References

[1] A. Jefferson Offutt, Alisa Irvine, "Testing Object-Oriented Software Using the Category-Partition Method", 17th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '95), Santa Barbara, CA, August 1995.

[2] Amie L. Souter, Lori L. Pollock, Dixie Hisley, "Inter-class Def-Use Analysis with Partial Class Representation", Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, Toulouse, France, 1999.

[3] Amie L. Souter, Lori L. Pollock, "OMEN: A Strategy for Testing Object-Oriented Software", Proceedings of the International Symposium on Software Testing and Analysis, Portland, Oregon, United States, 2000.

[4] Robert V. Binder, "Testing Object-Oriented Systems: models, patterns, and tools", Addison Wesley, ISBN 0-201-80938-9, 1999

[5] Jeff Offutt, Aynur Abdurazik, "Generating Tests from UML Specifications", 2nd International Conference on the Unified Modeling Language (UML'99), Fort Collins, CO, October 1999.

[6] Lionel Briand, Yvan Labiche, "A UML-based Approach to System Testing", Journal of Software and Systems Modeling (Springer) Volume 1 Issue 1, 2002.

[7] Aynur Abdurazik, Jeff Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation", The 3rd International Conference on the Unified Modeling Language (UML'00), York, UK, October 2000

[8] Ye Wu, Mei-Hwa Chen, Jeff Offutt, "UML-based Integration Testing for Component-based Software", The 2nd International Conference on COTS-Based Software Systems (ICCBSS), Ottawa, Canada, February 2003.

[9] Falk Fraikin, Thomas Leonhardt, "SeDiTeC – Testing Based on Sequence Diagrams", Proceedings ASE 2002 7th IEEE International Conference, 2002

[10] Jean Hartmann, Claudio Imoberdorf, Michael Meisinger, "UML-Based Integration Testing", Proceedings of the International Symposium on Software Testing and Analysis, Portland, Oregon, United States, 2000.

[11] S. L. Graham, P. B. Kessler, M. K. McKusick, "gprof: a Call Graph Execution Profiler", Proceedings of the 1982 SIGPLAN symposium on Compiler construction, Boston, Massachusetts, United States, 1982.

[12] J. Voegele, "Programming Language Comparison", http://www.jvoegele.com/software/langcomp.html, November 2003.

[13] The Object Management Group, "OMG Unified Modeling Language Version 1.4", http://www.omg.org, September 2001.

Appendix B

Publication in SERP 2004

Adequacy Criteria for Testing Polymorphism in the Context of Interactions

Siros Supavita and Taratip Suwannasart
Department of Computer Engineering, Faculty of Engineering
Chulalongkorn University, Bangkok, Thailand 10330
Siros.S@student.chula.ac.th, Taratip.S@chula.ac.th

Abstract

Testing is very important for polymorphic interactions, as the exact interactions or operation calls are unpredictable during design. Therefore, they are vulnerable to defects. Adequacy criteria are required to address thoroughness of testing in this context. Although there are other proposed criteria that concern about inheritance and polymorphism, none of them aims at the context of interactions. This paper proposes 5 adequacy criteria in the view of inheritance and polymorphism. Combined with other criteria from interaction view, they can be applied as criteria to generate and select test cases for object-oriented interaction testing.

Keywords: Software Testing, Test Adequacy Criteria, Object-Oriented, Interactions, Polymorphism

1. Introduction

As object-oriented software testing requires significantly different concerns from procedure-oriented software testing, specific test approaches for object-oriented paradigm are necessary [1,2,3,4,5]. As the unique features of the paradigm, including encapsulation, inheritance, and polymorphism, cause faults that do not exist in procedure-oriented software, the test approaches must aim to uncover these kinds of faults. Besides, it is addressed clearly in [2] that these features pose some difficult issues for testing and result in test adequacy criteria that are different from procedure-oriented software.

Adequacy criteria are defined specifically for each test approach, as each of them concerns with faults on different angles. In this paper, we explore an idea of adequacy of testing object-oriented software, focusing on inheritance and polymorphism features in the context of interactions, as we are working on defining a test approach for testing interactions defined in UML Sequence Diagrams. Given an interaction or, strictly speaking, a polymorphic interaction, the effects of inheritance and polymorphism under the interaction execution and testing are important to design an appropriate method for test generation and selection. Even also focused on inheritance and polymorphism, other researches about adequacy criteria focus on the problem in different ways.

The paper is organized as followed. Section 2 gives necessary background in adequacy criteria, inheritance, and polymorphism. Related works are reviewed in section 3. Our criteria are presented in section 4, while section 5 compares and discusses about their efficiency and subsumption. A case study is illustrated in section 6, and the paper is concluded in section 7.

2. Background

2.1 Adequacy Criteria

Adequacy criteria are always an essential part of software testing approaches. Without adequacy criteria, thoroughness of testing cannot be evaluated, and neither do correctness of the software under test. An adequacy criterion expresses whether a given test set is appropriate for uncovering faults in software under test according to a particular testing technique. It keeps a test set from being an exhaustive test set, which requires a very long time and a lot of effort to execute. This is not cost-effective and, sometimes, impossible in practice. Testing software with an adequate test set guarantees that the software is tested to some acceptable degree of thoroughness; as a consequence, quality or correctness of the software is trusted to the level corresponding to the criterion.

For program-based testing approaches, an adequacy criterion is defined based on basic building blocks of the artifacts under test [6]. For example, adequacy criteria of path testing (i.e. branch coverage criterion

etc.) [7] are defined based on elements in the control flow graph of the program under test.

2.2 Inheritance & Polymorphism

As very convenient features of object-oriented paradigm, inheritance and polymorphism can help to solve many problems with less complicated design. One of their important characteristics is instance substitution under a polymorphic interaction, where an instance of a class may be substituted, under the actual execution, by an instance of any subclasses of the class. With a little help from creational patterns like Abstract Factory [8], new classes can be introduced to inheritance hierarchies without any modification to the interaction code. However, it does not mean that testing for the additional classes is unnecessary [2,5].

There are variants of inheritance from different aspects. The effects of strict inheritance, subtyping, and subclassing under software testing are discussed in [5], and they all are also considered in this paper.

3. Related Works

3.1 Criteria for Testing Polymorphic Relationships

Alexander and Offutt present four test adequacy criteria for testing polymorphic relationships in object-oriented software in integration testing level [9]. These criteria are designed to support their test approach [10], which is an altered version of coupling-based testing technique [11]. Basically the criteria are defined around "defs" and "uses" like criteria for data flow testing [7]. Concerns about inheritance and polymorphism are applied, as two of the criteria require all subclasses to be tested in the interaction context of their respective superclasses.

3.2 Test Adequacy Criteria for UML Design Models

Andrews et al. propose adequacy criteria for testing UML design models, which include class diagrams and interaction diagrams [12]. Testing is performed on the design models using symbolic execution techniques rather than on the implementation of the models. The criteria are defined based on their respective diagram elements in several different aspects. For example, association-end multiplicity (AEM) criterion, which is defined based on association relationships in class diagrams, requires an adequate test set to cover scenarios where various numbers of instances are put on the opposite end of an association.

They also conducted an experiment to see whether test cases created based on the criteria are effective in detecting faults in design models. The result shows that some selected criteria are effective in revealing faults, including incorrect sequence numbering, missing flows, and data flow gaps.

4. Our Criteria

There are two important concerns in defining the criteria. The first concern is about how thorough an inheritance hierarchy is tested under a polymorphic interaction. Apparently there are two choices; whether to include all subclasses to be tested in the context of their superclass, or to just include only the subclasses which override the method under test. Although there is no practical evidence about necessity of testing all subclasses in the context of the superclass, several research studies address this issue [2,13]. From designers' point of view, inherited methods seem to be ready for use right away, as inheritance is supposed to promote reusability of the inherited method. However, the results from the studies, though counter-intuitive, give strong reasons to test all subclasses in the context of the superclass. Other researches, which focus on testing inheritance and polymorphism features, adopt this idea into their adequacy criteria [9,12]. In our research, we also embrace this idea as an adequacy criterion, as well as other more relaxed criteria as options.

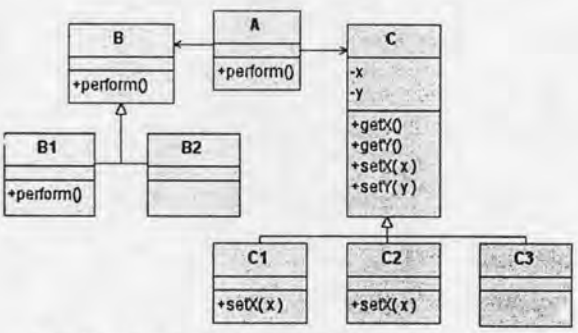

Figure 1. Class Diagram for the example

Another concern is in the situation where an interaction comprises more than one class with inheritance hierarchy. Say there are classes as shown in Figure 1, and their interaction is shown in Figure 2. From the diagrams, both class B and class C, which both have their own inheritance hierarchies, involve in the interaction. Supposed that the test adequacy criterion requires all subclasses from each inheritance

hierarchy to be tested, there will be 3 classes from class B hierarchy (B, B1, and B2) and 4 classes from class C hierarchy (C, C1, C2, and C3) to be tested. A decision is required to be made, as either the classes are tested in combination (like all paths coverage in path testing), or just cover each class at least once. It is clear to see that the first option results in a test set that grows multiplicatively, while the latter yields a test set that is limited to the maximum number of classes in each inheritance hierarchy. Indeed, the first option is more desirable in testers' perspective, as it covers all combinations of subclass substitution and results in a more thorough test set. However, it possibly results in a massive test set for situations like the example, where 12 test cases (the product of 3 and 4) are required for testing such a simple interaction without any condition or loop.

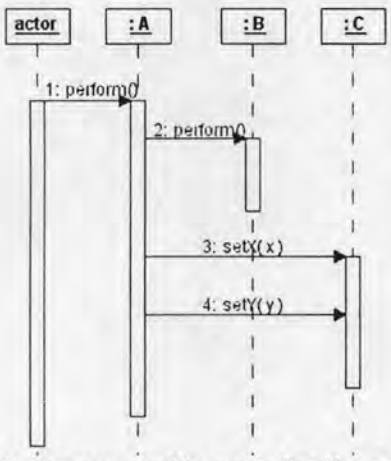

Figure 2. Sequence Diagram for the example

From these concerns, five criteria are defined as shown below. Table 1 shows an example of class selection set of test cases for the interaction in Figure 2 corresponding to each criterion, while the subsumption hierarchy of the criteria is shown in Figure 3. Note that although, inheritance coverage and strong overriding method coverage criteria are aligned on the same level, they are not comparable.

4.1 Base Class Coverage

This criterion requires no superclass-subclass substitution. Classes that must be covered under test are only those defined in the interaction.

4.2 Overriding Method Coverage

This criterion requires each class in an interaction to be substituted by its subclasses which override the method involved in the interaction.

4.3 Inheritance Coverage

Subsuming the previous criterion, this criterion requires each class in an interaction to be substituted by all of its subclasses.

4.4 Strong Overriding Method Coverage

Based on overriding method coverage criterion, this criterion is an extension to support the case where an interaction comprises more than one class with inheritance hierarchy. In addition to cover all overriding methods, this criterion requires testing for all combinations of all choices of substitution.

4.5 Strong Inheritance Coverage

Similar to strong overriding method coverage, this criterion is an extension of inheritance coverage criterion to support several subclass substitutions in an interaction. All substitutable classes are required to be covered in combinations.

Table 1. Example of class selection sets of test cases

Criteria	Class Selection Sets*
Base Class Coverage	{B,C}
Overriding Method Coverage	{B,C} {B1,C1} {B,C2}
Inheritance Coverage	{B,C} {B1,C1} {B2,C2} {B,C3}
Strong Overriding Method Coverage	{B,C} {B,C1} {B,C2} {B1,C} {B1,C1} {B1,C2}
Strong Inheritance Coverage	{B,C} {B,C1} {B,C2} {B,C3} {B1,C} {B1,C1} {B1,C2} {B1,C3} {B2,C} {B2,C1} {B2,C2} {B2,C3}

* As class A is always required for all test cases, it is left from the class selection sets.

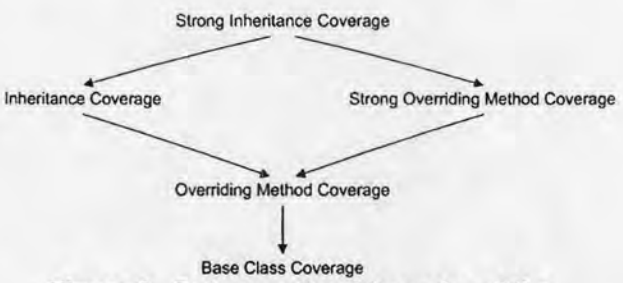

Figure 3. Subsumption hierarchy of the criteria

5. Discussion

As the weakest criterion, base class coverage criterion is apparently weak, inappropriate, and only useful in some exceptional cases. Such a case is where a subclass is the pure extension of its superclass (no overriding), and analysis result shows that the class does not involve in multiple context scenario. This situation is free from ITU (Inconsistent Type Use) fault presented by Offutt et al. [13], as the extension methods, which can cause failures, are not accessible in the context of the superclass, and the absence of multiple context scenario also keeps them from being accessed in the context of the subclass as well.

Another situation that this criterion is appropriate is where the inheritance is subclass inheritance rather than subtype inheritance. Subclass inheritance, as a relaxed version of subtype inheritance, does not preserve substitutable property of inheritance, since it rather aims at reusability of attributes and methods of the superclass. Superclass-subclass substitution does not occur at any time; consequently, no testing is required for it. Using base class coverage criterion in these situations cuts down the number of required test cases, as the additional test cases tend to uncover no additional fault.

It is intuitive that when a method is overridden, the overriding version of the method should be tested in the context of the overridden method to ensure that the overriding method is correct. According to this belief, overriding method coverage criterion seems to be appropriate for testing polymorphic interaction. However, showing in the studies, the absence of overriding method does not guarantee total correctness. While a subclass inherits a method from its superclass, correctness of the method is not inherited to the context of the subclass. There may be side effects from situations outside the scenario of the interaction under test like, for example, usage of multiple context (casting) and difference in initialization (or constructors) of superclass and subclass. Testing is usually required for the entire inheritance hierarchies to ensure complete correctness, whether overriding or not; therefore, inheritance coverage criteria is considerably better in terms of reliability and thoroughness. However, overriding method coverage criterion can provide a reasonable trade off in the situation where base class coverage criterion is inappropriate, and inheritance coverage criterion requires an unacceptable amount of effort.

Advancing the regular criteria by exercising all combinations of substitutable classes, strong overriding method coverage and strong inheritance coverage criteria are designed to cope with an interaction, which includes more than one class with inheritance hierarchy. While they are very effective to reveal faults which occur in the situation of a particular combination of class substitution, they may require a tremendous test set. In the situation where, in an interaction, two objects with inheritance hierarchies do not interact with each other directly, there is a very little chance that faults could occur from the combinations of class substitution. In such a case, either one of the regular criteria is possibly more cost-effective than its strong version criterion, as the excessive test cases are eliminated while comparable test effectiveness is achieved.

6. A Case Study

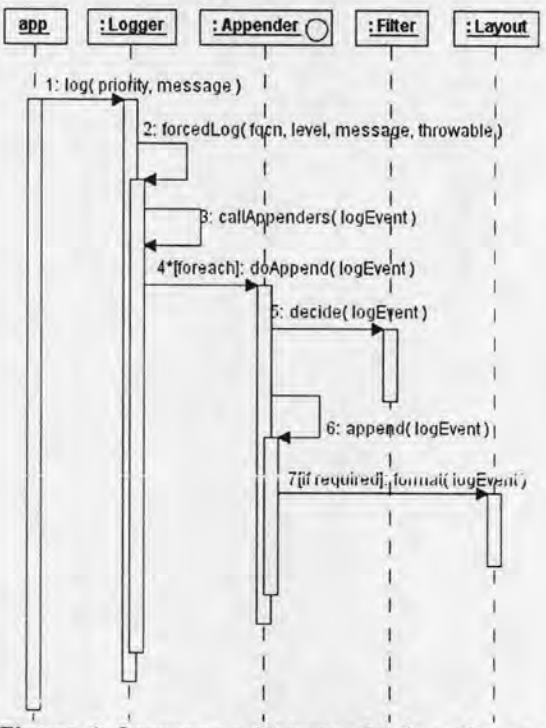

Figure 4. Sequence diagram for Log4j case study

We select Jakarta Log4j, an Apache open source project [14], to be analyzed as our case study. Log4j is a framework providing logging mechanism to any application developed under Java platform. Underneath its simple API, the logging mechanism is configurable, as the logging interaction is designed to be polymorphic. Basically, the logging interaction comprises several main classes as shown in UML Sequence Diagram in Figure 4. The diagram represents an overview of the logging interaction. (Although there are a lot more classes involved in the actual interaction,

only the classes that we focus are shown to simplify the example). Appender, Filter, and Layout have their own inheritance hierarchies shown in Figure 5 and 6, as any instance of their concrete subclasses can be substituted in the interaction to form any desirable logging capability. For example, substitution of an instance of ConsoleAppender class in the place of Appender interface causes log messages to go to the console of the application. Each implementation of Appender, Filter, and Layout must override append, decide, and format methods respectively to implement their specific functionality.

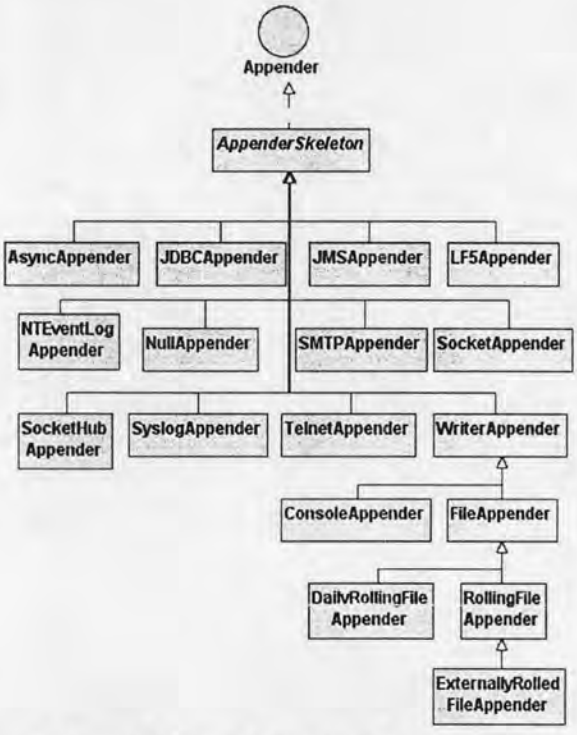

Figure 5. Class diagram of appenders

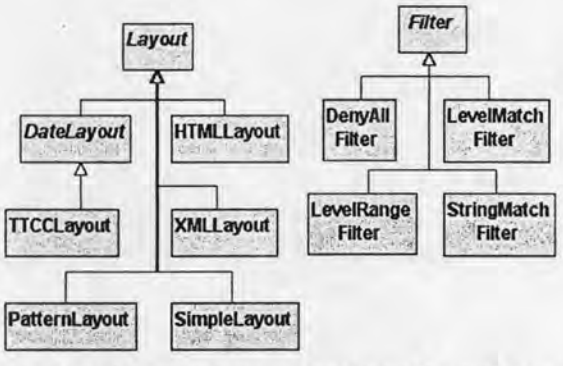

Figure 6. Class diagram for layouts and filters

6.1 Base Class Coverage

Testing this interaction using base class coverage criterion requires only one test case. Since Appender is an interface, and Layout and Filter are abstract classes, it is impossible to have direct instances of these types. Therefore, the test case must rather use instances of their concrete subclasses/implementations instead. As the criterion does not consider polymorphic interaction, any single selection of the classes satisfies the criterion. This criterion is obviously inappropriate for testing the interaction, as subclass substitution is fundamental under actual execution.

6.2 Overriding Method Coverage and Inheritance Coverage

All concrete subclasses of Appender, Filter, and Layout have overriding methods; consequently, applying overriding method coverage and inheritance coverage criteria yield similar test sets. The numbers of concrete implementations of Appender, Filter, and Layout are 17, 4, and 5 respectively; hence, the total number of required test cases is 17.

6.3 Strong Overriding Method Coverage and Strong Inheritance Coverage

Strong overriding method coverage and strong inheritance coverage criteria require significantly more test cases than overriding method coverage and inheritance coverage criteria. Considering only the concrete implementations, 340 test cases (17 x 4 x 5) are required to satisfy either one of the strong criteria. Although either overriding method coverage or inheritance coverage criterion requires a reasonably small amount of test cases, they seem unable to uncover faults from various combination of class substitution. Either strong overriding method coverage or strong inheritance coverage criterion seems to be more effective in detection of these faults, but the massive amount of required test cases does not seem to be cost-effective in practice.

6.4 Improvement

Selectively applying the criteria to different parts using some domain knowledge could yield a more effective test set, in both terms of cost and possibility of defect detection. From analysis of the interaction and the implementation, it is clear that Appender implementations deal with any Filter implementations in only doAppend method in AppenderSkeleton

abstract class, the superclass of all Appender implementations. If the interaction between AppenderSkeleton and Filter is tested with one of their implementation selection, it is less likely that other selections would introduce a new defect. Testing combinations of these classes seems to uncover no additional defect; consequently, some test cases can be removed without significant effect.

However, it is different for the interaction between Appender and Layout. Each implementation of Appender interface differently interacts with Layout; for example, they call the format method on Layout in different sequence or, in some cases, the method is not called at all (as some implementation of Appender does not require Layout, as marked in guard condition in Figure 4). As a result, it is still necessary to test combinations of these classes. However, another improvement comes from this fact that some Appender implementations do not require Layout. Testing them together is fruitless, let alone testing various combinations of them; therefore, some unnecessary test cases can be removed. This is one of limitations of our proposed criteria, as conditions in interactions are not considered.

The case study also shows another limitation of the criteria. Log4j allows a log event to go to more than one appender as shown in the sequence diagram as iteration condition of doAppend method call. The scenarios where more than one instance of appender attached to Logger are not covered by the criteria. Criteria based on interaction diagrams with this information would be more effective in addressing these issues.

7. Conclusion

In this paper, we address the importance of concerns of inheritance and polymorphism in testing. Our proposed criteria provide several useful guidelines for adequacy of testing polymorphic interactions. However, the case study shows that these criteria alone are not comprehensive enough to test interactions, as they leave some vulnerable parts of the interaction untested. Adequacy criteria from other aspects; for example interaction condition and multiplicity of association ends, are required to complement our criteria to form better test adequacy criteria.

As we are working on defining an approach for testing object-oriented interactions based on UML Sequence Diagrams, we plan to incorporate the proposed criteria as a part of adequacy criteria for the approach. As addressed above, we will blend these criteria with other criteria based on UML Sequence Diagrams to strengthen the proposed criteria.

8. Acknowledgement

This research is supported by TJTTP-OECF (Thai-Japan Technology Transfer Project - Japanese Overseas Economic Cooperation Fund) and Department of Computer Engineering – Industry Linkage Research Project, year 2004, Chulalongkorn University. The authors would like to thank Prof. Koichiro Ochimizu of JAIST (Japan Advance Institute of Science and Technology) for his useful comments and suggestions.

9. References

[1] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 1999.

[2] D. E. Perry and G. E. Kaiser, "Object-Oriented Programs and Testing", *Journal of Object Oriented Programming*, January/February 1990

[3] J. H. Hayes, "Object-Oriented Programming Systems (OOPS): A Fault-Based Approach", *Proceedings of International Symposium on Object-Oriented Methodologies and Systems (ISOOMS)*, Palermo, Italy, September 1994, pp. 205-220.

[4] E. V. Berard, "Issues in the Testing of Object-Oriented Software", *Proceedings of Electro'94 International*, IEEE Computer Society Press, 1994, pp. 211–219.

[5] S. Barbey and A. Strohmeier, "The Problematics of Testing Object-Oriented Software", *Proceedings of SQM '94, Second Conference on Software Quality Management*, Edinburgh, Scotland, UK, volume 2, 1994, pp. 411-426.

[6] E. J. Weyuker, "The Evaluation of Program-Based Software Test Data Adequacy Criteria", *Communications of the ACM*, Volume 31, Issue 6, June 1988.

[7] B. Beizer, *Software Testing Techniques*, 2nd Edition, Van Nostrand Reinhold Co., 1990.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, 1st Edition, Addison-Wesley, 1995.

[9] R. T. Alexander and A. J. Offutt, "Criteria for Testing Polymorphic Relationships", *Proceedings of the International Symposium on Software Reliability and Engineering (ISSRE00)*, IEEE Computer Society, San Jose CA, 2000, pp.15-23.

[10] R. T. Alexander and A. J. Offutt, "Analysis Techniques for Testing Polymorphic Relationships", *Proceedings of the*

Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00), Tokyo, Japan, 2000, pp. 172-178.

[11] Z. Jin and A. J. Offutt, "Coupling-Based Integration Testing", *Proceedings of Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '97)*, Montreal, Canada, October 1996, pp. 10-17.

[12] A. Andrews, R. France, S. Ghosh, and G. Craig, "Test Adequacy Criteria for UML Design Models", *Software Testing, Verification, and Reliability Journal*, Volume 13, Number 2, June 2003.

[13] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson, "A Fault Model for Subtype Inheritance and Polymorphism", *Proceedings of Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE '01)*, Hong Kong, PRC, November 2001, pp. 84-95.

[14] Apache Software Foundation, Jakarta Log4j, http://logging.apache.org/log4j/docs/, 2004.

Appendix C

Publication in ITCC 2005

Testing Polymorphic Interactions in UML Sequence Diagrams

Siros Supavita and Taratip Suwannasart
Department of Computer Engineering, Faculty of Engineering
Chulalongkorn University, Bangkok, Thailand 10330
Siros.S@student.chula.ac.th, Taratip.S@chula.ac.th

Abstract

Nowadays UML based testing becomes more and more noteworthy, as UML is broadly accepted as de facto standard for object-oriented modeling language. A lot of researches propose test approaches for several types of UML diagrams, focusing on various viewpoints. An approach for testing polymorphic interactions, which are defined in UML Sequence Diagrams, is presented in this paper. From our observation, there are several forms of polymorphic interactions, which have their polymorphic behavior controlled by distinct factors. A method to test each form is determined from these factors. If an interaction is in one of the polymorphic forms defined in this paper, the factors that affect polymorphic behavior of the interaction can be addresses. Finally test cases for the interaction can be created for testing its polymorphic behavior.

Keywords: Software Testing, Object-Oriented, Polymorphism, UML, Sequence Diagram

1. Introduction

UML [1] is getting more popular as a standard modeling language for object-oriented analysis and design. With its well-understood notations, it provides a wide array of diagrams, which help designers to effectively communicate with developers in many different aspects. As an interaction diagram, UML Sequence Diagram shows a design of an interaction among a group of objects under a particular scenario. Basically, an interaction is represented as a sequence of messages sending between objects. Conformance-directed testing [2] aims at uncovering any discrepancy between specification and implementation. Based on UML Sequence Diagrams, this kind of testing is to determine whether the implementation produces the equivalent message sending sequences as defined in the UML Sequence Diagrams as the specification.

In addition to control flow, testing object-oriented interactions also needs to take into account the effect of polymorphism. In our previous work [3], we present an instrumentation model for message sending sequences. The model supports comparison of the expected message sending sequence against the actual message sending sequence, which is not necessarily exactly identical to each other, due to the effect of polymorphism. However, it is required that a polymorphic interaction is tested with particular scenarios to ensure that particular subclasses are tested in the interaction as instances of their superclass, which is defined in the interaction.

For a polymorphic interaction, there are factors that influence its polymorphic behavior. Test cases must be designed around these factors in order to execute required classes under particular scenarios. This paper discusses several forms of polymorphic interactions and their factors. For a polymorphic interaction under test, it is necessary that the factors are identified. This paper shows the basic pattern and factors of each form along with guideline for test design.

This paper is organized as follow. Related background knowledge about object-oriented software testing is explained in Section 2, while Section 3 reviews our test approach. In Section 4, the forms of polymorphic interactions are presented along with some examples of the forms. A discussion about variations of the forms is given in Section 5. The paper is concluded in Section 6 with potential future work.

2. Object-Oriented Software Testing

2.1 Polymorphism and Testing

Polymorphism means the ability to take several forms [4]. In other words, an instance of a class can be handled as an instance of one of its superclass. In

cooperation with dynamic binding, it is a very useful feature, as it allows a program to have dynamic behavior according to the actual type of object under execution. A number of design patterns [5] are formalized based on this concept.

Despite its usefulness, a number of researches address that polymorphism does not prevent the software from defects [2,6,7,8]. Perry and Kaiser state that both inherited and overriding operations require new test sets, which are different from the test set for their superclass [6]. Furthermore, an object reference in runtime can be substituted by a reference to the class, which is not the one defined in the code. This could make behavior of the program unpredictable, which is not good in testing point of view [7].

There are several terms defined by Meyer [4] which are used in this paper. "Polymorphic assignment" is the situation when a reference is assigned with an object which is not its exact type, but one of its subclasses. "Polymorphic entity" is a reference which appears in a polymorphic assignment.

2.2 Testability

Testability is always one of the most important issues in testing. In [9], Binder states that testability consists of 2 important factors: controllability and observability.

Controllability is the capability of controlling inputs of the program under test. Lack of controllability results in low testability, as it is difficult to test the program under test as intended. The term "input" also includes data that are not explicitly passed to the program. As an object can have its own internal state, an operation on the object can have execution logics according to the state. Usually encapsulation and information hiding prevent object states from being accessed directly. This could become a big obstacle in testing, since object states as ones of inputs cannot be controlled by test drivers. A common workaround for this problem is to temporarily break encapsulation by adding public methods to access and mutate the states during the test.

Observability is the ability to examine intermediate outputs of the program under test. It is important to have access to these outputs in order to determine whether the test passes or fails. Similar to controllability, encapsulation is also the major obstacle for observability.

Controllability is critical in the discussion of this paper, since it is required that a polymorphic interaction under test is controlled to be executed with a particular set of classes. From now on, it is assumed that whatever input is discussed, it is always accessible to test drivers. Therefore, there is no controllability issue.

3. Message Sending Sequence Testing

3.1 Concept

Message sending is a way to express object communication in an interaction. A sender object sends a message to request a receiver object to fulfill a particular service [4]. A message sending sequence is a sequence of messages flying between objects in an interaction under a specific scenario.

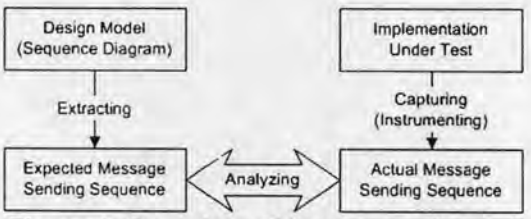

Figure 1. Overview of the comparison of message sending sequences

In addition to checking the actual test result with test oracle, the comparison of message sending sequences is the key of this test approach. The approach aims to detect any discrepancy between interactions in the design model and the implementation. This test approach is conformance-directed testing rather than fault-directed testing [2], as there is no specific fault model. Any difference between the message sending sequences is considered as a defect. As shown in Figure 1, the expected message sending sequence, which is extracted from the design model, is analyzed against the actual message sending sequence, which is instrumented from the execution of the implementation. A model which represents both types of message sending sequences is presented in our previous work [3]. The model captures all information necessary for comparison of the message sending sequences, particularly polymorphism related information.

3.2 Refinement Issue

In design model, all detailed information is not usually presented. Refinement is an action of implementing the software slightly differently from the design model usually by adding implementation level detail, which is intentionally omitted from the design model. With refinement, the message sending sequence from the implementation is not identical to the one

from the design model, i.e. UML Sequence Diagram, although the implementation is correctly implemented based on the diagram.

For example, common classes, which provide elementary services like String and Integer in Java, are omitted from the design model for simplicity in the design model, although they are presented and required in the implementation. Comparison of message sending sequence must take care of identifying refinement messages; therefore, the actual message sending sequence of an implementation with refinement can be compared against its design model.

4. Forms of Polymorphic Interactions and Testing

This section discusses forms of polymorphic interaction and how to test them. To thoroughly test a polymorphic interaction requires that all subclasses are substituted to the polymorphic entity; as a consequence, each test case must be designed to execute a particular subclass.

Polymorphism could only happen when there is polymorphic assignment in the interaction. Forms discussed in this section differ from each other in the patterns of their polymorphic assignments. The form of non-polymorphic interaction is introduced before the 3 forms of polymorphic assignments to identify and exclude non-polymorphic interaction from our further discussion.

4.1 Non-Polymorphic Interaction

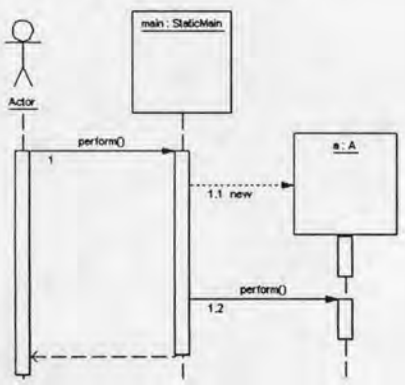

Figure 2. Example of non-polymorphic interaction

Basically non-polymorphic interaction is an interaction which does not have any polymorphic assignment, although a reference in the interaction is defined as a class with one or more subclasses. In other words, it is the situation where the classes of the instances in an interaction are unconditionally fixed to the same set of classes for every execution. Figure 2 shows an example of this form of interaction. Although class A may have subclasses, it is not possible for polymorphism to occur.

4.2 Simple Polymorphic Assignment

The first form of polymorphic assignment is the simplest one. An instance which is assigned to the polymorphic entity is passed as a parameter to the interaction; therefore, polymorphic behavior is directly controlled by the parameter. Figure 3 shows the UML Class Diagram of classes in the class hierarchy used in the examples in this section. An example of an interaction with simple polymorphic assignment is shown in Figure 4.

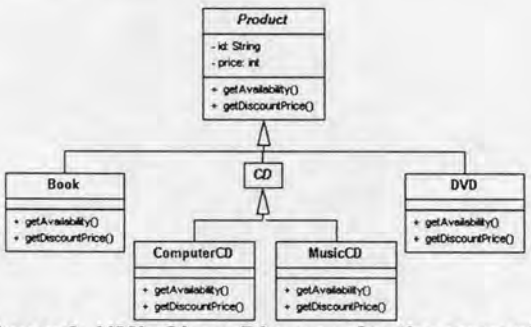

Figure 3. UML Class Diagram for the example

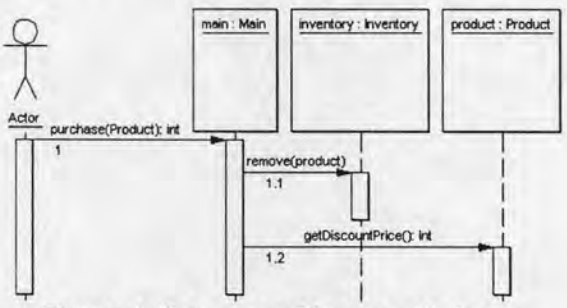

Figure 4. Sequence Diagram of simple polymorphic assignment

Testing an interaction with this form of polymorphic assignment requires each test case to pass in an instance of the class intended to be tested as test input. This form of polymorphic assignment is easy to test, as the parameter to the interaction is directly assigned to the polymorphic entity.

From Figure 3, there are 4 concrete subclasses of abstract class Product (classes with italic names are abstract): Book, ComputerCD, MusicCD, and DVD. In order to test this interaction with all 4 subclasses, 4 test cases must be created; one for each class. The

skeletons of test cases for this example are shown in table 1.

Table 1. Test cases for interaction in Figure 4[1]

No.	Test Input
1	An instance of Book
2	An instance of ComputerCD
3	An instance of MusicCD
4	An instance of DVD

4.3 Parameter-Influenced Polymorphic Assignment

Parameter-influenced polymorphic assignment, as its name suggests, is a polymorphic assignment which has its behavior dependent on one or more parameters. The common form of this type of assignment is one of the variations of "factory method" design pattern, known as "parameterized factory method" [5]. As a creational pattern, a "parameterized factory method" is responsible for creating instances of various classes from the same inheritance hierarchy. One or more parameters passed to the method are used to determine from which class an instance is created.

An example of an interaction with parameter-influenced polymorphic assignment, still based on inheritance hierarchy in Figure 3, is shown in Figure 5. The polymorphic assignment is in step 1.1, where the reference "product" is assigned to the return result from method "getProduct" on finder object as an instance of class ProductFinder. The method "getProduct" is a parameterized factory method, as it creates an instance of one of concrete subclasses of class Product according to the parameter "id" (it may determine the type of product by looking up in data storage, e.g. database, using the given id.)

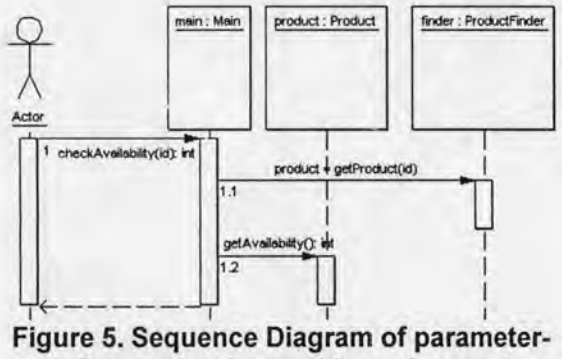

Figure 5. Sequence Diagram of parameter-influenced polymorphic assignment

[1] Note that the table does not show complete test cases. Some information, e.g. exact test data and expected result, is omitted, as it is not relevant to the discussion. This is also true for the subsequent subsections.

This form of assignment is similar to simple polymorphic assignment discussed in the previous subsection in that they both are dependent on parameters. Testing this form of assignment, however, is not as straightforward as the other. Each test case must be designed with specific test input which causes the polymorphic assignment to result in the particular class.

Table 2 shows skeletons of test cases for this example. Notice that these test cases are more abstract than the ones from table 1. It is not possible to determine exactly what values of the test inputs should be just from the interaction diagram. Additional information and, sometimes, human decision are required to identify the exact appropriate test input values. As a result, testing this form of polymorphic assignment is less likely to be automated.

Table 2. Test cases for interaction in Figure 5

No.	Test Input (id)
1	Id of a book
2	Id of a computer CD
3	Id of a music CD
4	Id of a DVD

From Figure 5, the parameter "id" is identified as the parameter which affects the polymorphic assignment. It is required that this parameter is also one of parameters of the interaction or is directly dependent on parameters of the interaction, in order to fulfill testability requirement. As this example falls into the first case, the test cases are designed based on the parameter that influences the polymorphic assignment (product id). For the latter case, test case design is more complicated, if not impossible.

4.4 Configuration-Influenced Polymorphic Assignment

The last form is configuration influenced polymorphic assignment. For this type of assignment, one or more types of configuration are the factor that influences polymorphic assignment. Configuration in this context includes all types of set-up and environment, which is external to the interaction. Different from parameter-influenced polymorphic assignment, this form of assignment is not dependent on any parameter passed to the interaction; thus, test cases for this form of assignment are not designed solely around test inputs.

Also based on inheritance hierarchy in Figure 3, Figure 6 shows an example of an interaction with configuration-influenced polymorphic assignment. The polymorphic assignment is in step 1.1, which is the

result of the execution of method "getTopSelling" on finder object as an instance of class ProductFinder. Notice that it is different from the interaction in Figure 5 in that the method "getTopSelling" does not take any argument. The method "getTopSelling" returns an instance of the best selling product, which could be one of concrete subclasses of class Product. The behavior of this polymorphic assignment is based on external set-up, not any of parameters of the interaction. Testing this interaction with different subclasses of class Product requires set-up to have different types of product as the best selling product. The skeletons of test cases for this example are shown in table 3.

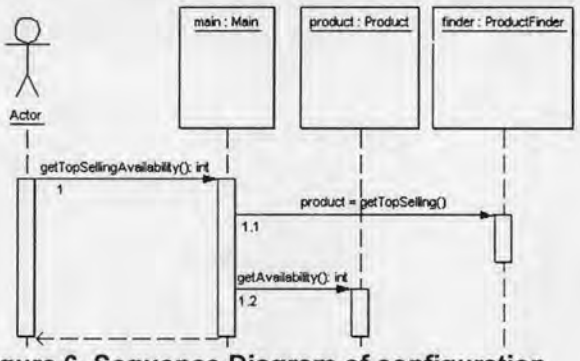

Figure 6. Sequence Diagram of configuration-influenced polymorphic assignment

Table 3. Test cases for interaction in Figure 6

No.	Test Set-up
1.	Set the top selling product to be a book
2.	Set the top selling product to be a computer CD
3.	Set the top selling product to be a music CD
4.	Set the top selling product to be a DVD

The interaction in Figure 6 does not have any argument, and the polymorphic assignment is not dependent on any parameter either. Test cases for this interaction must be designed around configuration, as it is the factor that affects the polymorphic assignment. Testing this form of assignment is more complicated than testing parameter-influenced polymorphic assignment, as test cases with configuration set-up or, best known as, test set-up are difficult to be designed systematically. Moreover, automated test for this form is difficult due to the variety of configuration implementation.

There are a number of possible implementations which falls into configuration-influenced polymorphic assignment. From the example in Figure 6, some content, which is persisted in a file or a database, affects the polymorphic assignment; therefore, changing sales record may result in change of the best selling product. For each test case execution on this interaction, set-up steps must be performed to achieve the required configuration state.

Another example of this form is inversion of control or dependency injection pattern presented by Fowler [10]. There are several Java implementations of this concept, e.g. Spring [11], PicoContainer [12] etc. With dependency injection, associations between objects are not statically fixed in the compiled code, but injected into the code in runtime. Dependency injection could result in polymorphic assignment. Another alternative is "Service Locator" patterns presented by Sun Microsystems [13].

5. Discussion

5.1 Variations

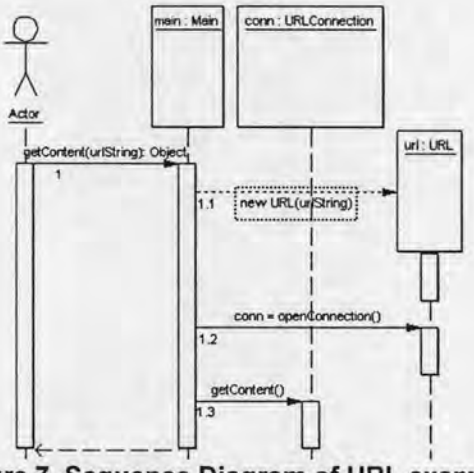

Figure 7. Sequence Diagram of URL example

In the previous section, forms of polymorphic assignments are discussed in basic patterns. However, it is not uncommon to see different patterns of polymorphic assignments in the real world. Figure 7 shows an example of parameter-influenced polymorphic assignment, which does not strictly follow the basic pattern. The polymorphic entity is "conn" as a reference to class URLConnection, which has subclasses for specific URL protocols (i.e. HttpURLConnection for HTTP connection and FtpURLConnection for FTP connection etc.). However, the polymorphic assignment in step 1.2 does not take any parameter. The parameter that affects the polymorphic assignment is "urlString", supplied in step 1.1 when an instance of URL is created. Then test case design for this interaction must focus on the parameter "urlString".

This example shows that, although does not follow the pattern exactly, the concept of testing can also be applied. The problem is how to identify the form of

these interactions. Stereotypes in the design model may be required in order to automate test case creation.

5.2 Combination

Another possible situation is when there is more than one polymorphic entity in an interaction. In addition, it is possible that different forms of polymorphic assignments happen in the same interaction. The concept in this paper is still applicable in this case, although it does not deal with the issue of combination. In our previous work [14], we present adequacy criteria for testing polymorphic interaction which address the situation where more than one polymorphic entity is involved.

6. Conclusion

In this paper, 3 forms of polymorphic assignments are presented along with their test case design approach. Given a polymorphic interaction with an inheritance hierarchy, it is possible to apply the relevant approach to design test cases for the interaction. Although test data generation is not considered in this paper, automated test case generation is possible with some help from additional information in the design model.

The finding in this paper complements our previous works [3,14]; a tool can be implemented to aid testing of OO software based on its sequence diagrams. From UML Sequence Diagrams with additional information for inheritance hierarchy from, for example, UML Class Diagram, adequacy criteria presented in [14] can be applied to decide which classes (or, strictly speaking, subclasses) are to be included under test and how many test cases are required to cover these classes. Combined with conventional control flow testing, the concept in this paper can be applied to identify test input and/or test set-up for each test case, and finally an instrumentation model in [3] is applied to check whether test execution results in the same message sending sequence, including the order of messages and classes of instances under test, as designed in the design model and the test case.

7. Acknowledgement

This research is supported by TJTTP-OECF (Thailand-Japan Technology Transfer Project - Japanese Overseas Economic Cooperation Fund) and Department of Computer Engineering – Industry Linkage Research Project, year 2004, Chulalongkorn University. The authors would like to thank Prof. Koichiro Ochimizu of JAIST (Japan Advance Institute of Science and Technology) for his useful comments and suggestions.

8. References

[1] The Object Management Group, "OMG Unified Modeling Language Version 1.4", http://www.omg.org, September 2001.

[2] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 1999.

[3] S. Supavita and T. Suwannasart, "An Instrumentation Model for Supporting Software Testing Based on UML Sequence Diagrams", *Proceedings of the 4th Information and Computer Engineering Postgraduate Workshop (ICEP 2004)*, Phuket, Thailand, January 2004.

[4] B. Meyer, *Object-Oriented Software Construction*, 2nd Edition, Prentice Hall, 1997.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, 1st Edition, Addison-Wesley, 1995.

[6] D. E. Perry and G. E. Kaiser, "Object-Oriented Programs and Testing", *Journal of Object Oriented Programming*, January/February 1990.

[7] M. D. Smith, D. J. Robson, "Object-Oriented Programming - the Problems of Validation", *Proceedings of Conference on Software Maintenance*, San Diego, CA, USA, 26-29 November 1990, pp. 272-281.

[8] S. Barbey and A. Strohmeier, "The Problematics of Testing Object-Oriented Software", *Proceedings of SQM '94 Second Conference on Software Quality Management*, Edinburgh, Scotland, UK, Volume 2, 1994, pp. 411-426.

[9] R. V. Binder, "Design for Testability in Object-Oriented Systems", *Communications of the ACM*, Volume 37, Issue 9, September 1994.

[10] M. Fowler, "Inversion of Control Containers and the Dependency Injection Pattern", http://www.martinfowler.com/articles/injection.html, January 2004.

[11] "Spring – Java/J2EE Application Framework", http://www.springframework.org/, November 2004.

[12] "Pico Container", http://www.picocontainer.org/, November 2004.

[13] Sun Microsystems, "Core J2EE Patterns - Service Locator", http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html, November 2004.

[14] S. Supavita and T. Suwannasart, "Adequacy Criteria for Testing Polymorphism in the Context of Interactions", *Proceedings of The International Conference on Sofware Engineering Research and Practice (SERP'04)*, July 2004.

BIOGRAPHY

Mr. Siros Supavita was born in Bangkok at March 16, 1978. He got a bachelor degree, Bachelor of Engineering in Computer Engineering from Faculty of Engineering, King Mongkhut's Institute of Technology Ladkrabang in 1999. He has been working in software development industry since then. He got a master degree, Master of Engineering in Computer Engineering from Faculty of Engineering, Chulalongkorn University in 2007.