# CHAPTER II

# LITERATURE REVIEWS

## 2.1 Concepts and Theories

### 2.1.1 Object-Oriented Software

Object-oriented analysis and design is a paradigm for design and development of software. The basic building block of the paradigm is a class which is a template of objects. A solution to a particular problem is achieved by an interaction of a group of objects via messages. In other words, objects collaborate with each other by sending messages. A sender object sends a message to request a receiver object to fulfill a particular service [11]. A message sending sequence is a sequence of messages flying between objects in an interaction under a specific scenario.

Inheritance and polymorphism are the features that make object-oriented paradigm unique. They are among features that defined object-oriented programming languages [12]. Inheritance occurs when a class, called a subclass, derives some or all of the members from another class, called a superclass, and makes them become its own members. Inheritance relationships can be organized into a hierarchy, called inheritance hierarchy, since a superclass of a class may have its own superclass and so on. Members inherited from upper classes in an inheritance hierarchy are also indirectly inherited to lower classes in the hierarchy. There are programming languages, for example C++ and Eiffel, which allow a class to have more than one direct superclass. This feature is called multiple inheritance. In this case, members from all superclasses are inherited to the subclass. Figure 2.1 shows an example of inheritance in a UML class diagram. There are 3 classes: A, B, and C. A is the superclass of B, and B is the superclass of C. From the example, class C has method n (which is defined in class C itself), attribute y (which is defined in class B), attribute x, and method m (which are defined in class A).

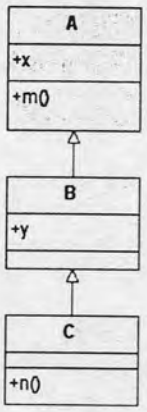


Figure 2.1 Example of inheritance

There are several types of inheritance. Some are specific to a particular programming language. Most common types are subtype inheritance and subclass inheritance. Subtype inheritance requires that an instance of a subclass must be a kind of the superclass, since inheritance relationship is treated as an "is-a" relationship. For example, class Rectangle is a subclass of class Shape, as an instance of class Rectangle is also considered a Shape object. The instance has the characteristic of Shape class as well as Rectangle class. Subclass inheritance is more relaxed than subtype inheritance. Its major purpose is to allow members on a superclass to be reused in subclass; consequently, a subclass is not necessarily a kind of its superclass. It is arguable whether to use subclass inheritance or not [13, 14].

Overriding is a feature that complements inheritance. In addition to inheriting a feature, a subclass has another option to override a feature from its superclass. Most object-oriented programming languages provide supports for only instance method overriding. A method is generally overridden by re-defining a member in the subclass with the same name (or, in other words, signature) as a member in the superclass. In this case, the newly defined member in the subclass is called an overriding method, and the method in the superclass is called an overridden method. When a method is overridden, it is completely excluded from the context of the subclass where it is overridden. An instance of the subclass responds to a call to the method by executing

the overriding version of the method. Similar to inheritance, overriding goes along an inheritance hierarchy. A subclass can override methods defined in its direct superclass and also other indirect superclasses. Also an overriding method can be overridden again by another lower subclass in the inheritance hierarchy. An example of overriding is shown as a UML class diagram in figure 2.2.

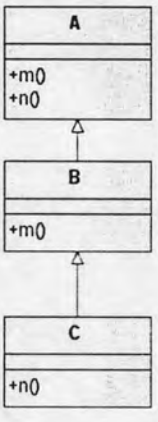


Figure 2.2 Example of overriding

From figure 2.2, class B is a direct subclass of class A, and class C is a direct subclass of class B. Class A defines 2 instance methods: m and n. Class B overrides method m, and class C overrides method n. Calls to instances of different classes results in executions of different versions of methods. For example, a call to method n on an instance of class B results in an execution of method n defined in class A, while a call to method n on an instance of class C results in an execution of method n defined in class C.

Polymorphism means an ability to take several forms [11]. This feature complements inheritance and overriding, as an instance of a class can be handled as an instance of one of its superclass. For example, the inheritance hierarchy in figure 2.2 shows that an instance of class C can be also treated as an instance of class B and an instance of class A. This case is illustrated as a UML sequence diagram in figure 2.3 and Java code in figure 2.4. A method, doSomething, takes an instance of class A as its

parameter, where the three lowermost lines show calls to the method with instances of class A, B, and C respectively.

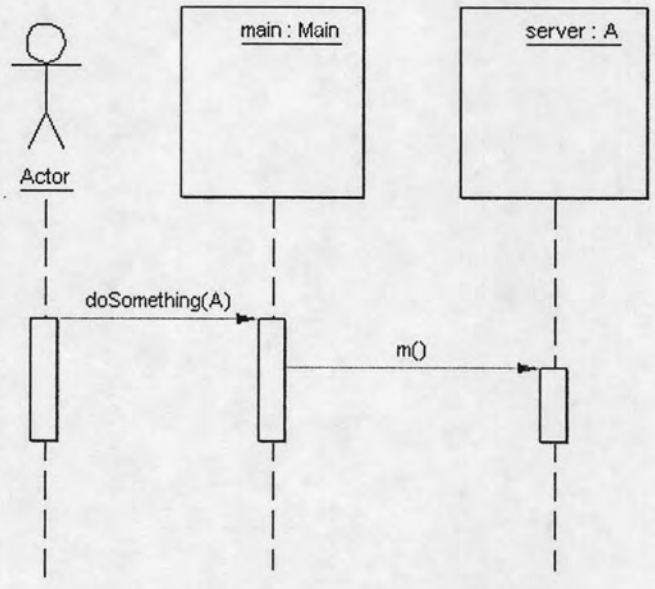


Figure 2.3 Example of polymorphism in UML sequence diagram

```
public static void doSomething(A server) {
      ...
      server.m();
      ...
}

...

doSomething(new A());
doSomething(new B());
doSomething(new C());
```

Figure 2.4 Example of polymorphism in Java code

Complementing inheritance and polymorphism, dynamic binding ensures that a polymorphic call goes to the correct version of method corresponding to the class of the receiver object. From figure 2.4, somewhere during the execution of method doSomething, method m is called on the parameter object "server". The parameter

object "server" can be an instance of either class A, B, or C; therefore, the call would result in execution of the version of the method corresponding to the actual class of the instance of parameter "server".

Regarding Meyer's definition [11], a "descendant" of a class C is any class that inherits from C, directly or indirectly, including class C itself. A "proper descendant" of a class C is a descendant of class C other than itself. The definitions of "ancestor" and "proper ancestor" are also defined in the same fashion as "descendant" and "proper descendant".

A "polymorphic assignment" is a situation when a reference is assigned with an object which is not its exact type, but one of its subclasses. A "polymorphic entity" is a reference which appears in a polymorphic assignment. From figure 2.4, the parameter object "server" is a polymorphic entity. The formal parameter object "server" at the definition of method doSomething is a polymorphic assignment, as it is where the parameter object "server" is assigned with an instance of either class A or any of its subclasses.

In this research, a "polymorphic interaction" is referred to as an interaction which contains one or more polymorphic call. In other words, there is at least one polymorphic assignment in the interaction. The UML sequence diagram in figure 2.3 and the code in figure 2.4 are both polymorphic interactions.

### 2.1.2 Object-Oriented Software Testing

During late 80's and early 90's, researchers tried to identify whether object-oriented features could keep programmers and designers from making mistakes, which causes faults. Furthermore, they tried to answer whether object-oriented software requires specific techniques for testing or existing test methods could be applied. As the first publication that addressed issues in object-oriented software testing with formal analysis, Perry and Kaiser [1] presented adequacy criteria for object-oriented software testing. Adequacy criteria play an essential role in every usual test process. They are generally a measure of thoroughness of testing. After a test set, as a set of test cases, completely passes, an adequacy criterion are applied to determine the efficiency of the test set. If the criterion is satisfied, the test set is adequate, and the software under test

(SUT) is adequately tested. Otherwise, the test set is inadequate, and more test cases are required to make the test set become adequate. This process is shown in figure 2.5. Another use of adequacy criteria is for guiding test case generation. Test case generation should be aware of relevant adequacy criteria in order to generate an adequate test set.

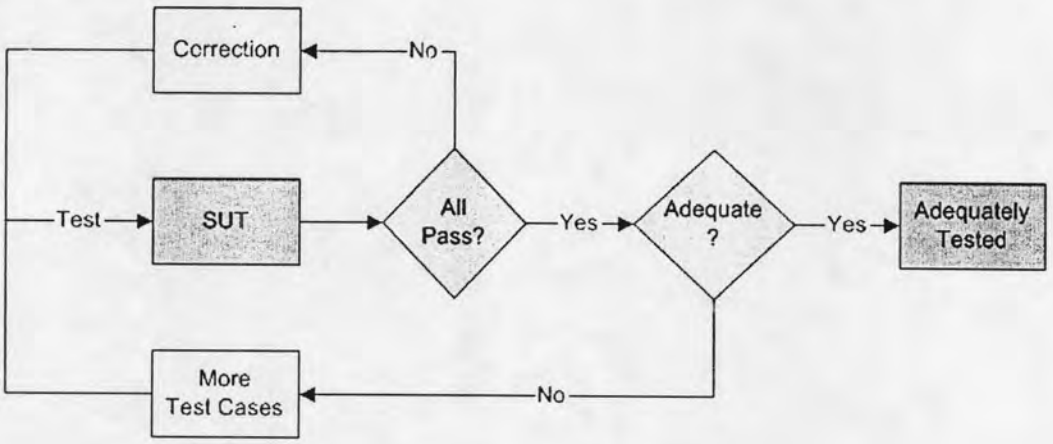


Figure 2.5 Adequacy criteria in test process

The result from Perry and Kaiser is quite surprising, as most parts of it are counter-intuitive. Object-oriented features, such as encapsulation and inheritance which ease several tasks in software development, including design, coding, and maintenance, are not really directly beneficial as far as testing is concerned.

Smith and Robson [2] stated that polymorphism and genericity are among the obstacles in object-oriented software testing. Applying these features, an object reference in runtime can be substituted by an object reference of the class/type different than the one defined in the code. This could make behavior of the program in runtime partly unpredictable, which is not good in testing point of view. In spite of the good effect in design and coding, these features aggravate the problem of object-oriented software testing, as it is required that all substitution should be tested to ensure its correctness and integrity. This finding was also supported by Barbey and Strohmeier [4]. In his book [8], Binder discussed the problem of polymorphism along with testing related problems of other object-oriented features.

Regarding testing techniques, one question always arises among researchers: whether existing testing techniques are effective for object-oriented software. As discussed in previous paragraphs, features in object-oriented software tend to pose new kinds of faults that are unique to the paradigm. Existing traditional testing techniques are not effective in detecting these faults, as they are not designed to do so. This idea is supported by a number of publications [1, 2, 3, 4, 5, 6, 8, 15, 16]. Furthermore, some defects in traditional procedural software are not likely to occur in object-oriented software, due to differences in their nature. While the complexity of procedural programs is in control flow inside procedures, testing techniques for this type of programs, for example control flow testing and data flow testing etc., are designed based on that assumption that faults are usually related to control flow of programs. However, object-oriented programs usually have simple control flow inside operations. Complexity in object-oriented programs lies in relationships between classes and their interactions.

Another difference between testing of procedural and object-oriented software is test level. Traditional unit, integration, and system test levels do not fit well with object-oriented software. It was arguable whether the basic unit for testing is an operation [17, 18, 19, 20] or a class [4, 8, 21]. Binder stated that a class is the basic unit for testing, as an operation is meaningless outside the context of its enclosing class [8, 21]. This idea becomes more acceptable later on, as researchers and practitioners find that it is more reasonable and more practical to test on class level than to test on operation level.

Smith and Robson [2] pointed that, as classes are designed to represent templates of abstraction of real-world problems, they are not organized as trees, like functional decomposition trees in procedural programming. Techniques for integration testing, such as top-down or bottom-up testing, are not applicable to object-oriented software testing, and neither is the concept of integration testing level itself. Cheathem and Mellinger [22] called the second test level as cluster testing rather than integration testing. Like Smith and Robson, they explained that traditional integration testing techniques are not applicable to object-oriented software. Berard [5] also supported this idea. He addressed that object-oriented software does not have a "top"; therefore, integration testing approach based on tree hierarchies cannot be applied.

However, later researches and practices focus more on cluster level testing. Testing a class alone is not likely to be effective, as an instance of a class usually collaborates with other instances of the same class or other classes to complete its operations. Testing an individual instance of a class requires test stubs to be created to replace the missing required instances. This task consumes a significant amount of effort and is not considerably cost-effective, as test stubs are usually thrown away after testing of the class is complete. A strategy presented by Kung et al. [23] is an example of an attempt to reduce effort in creating test stubs by testing a cluster of classes by incrementally integrating new classes into the cluster. Rather than testing each class individually with its stubs, a class with dependencies to other classes is integrated with tested classes to eliminate the need for stubs of those classes. There are also refinements of this technique to improve efficiency of test order [24] and to solve cyclic dependency [25].

### 2.1.3 Unified Modeling Language

Unified Modeling Language or UML [7] is a standard set of notations for modeling object-oriented systems. It is nowadays common in object-oriented software development. With its wide array of diagrams, software engineers can represent their systems in various views.

The most well known UML diagram is UML class diagram. A class diagram shows a static structural view of a system. Elements in this type of diagram are classes and their relationships. A class diagram is usually used to represent class structures (attributes and operations), inheritance hierarchies, association relationships, and dependencies. A class diagram can represent either conceptual class structure or implementation level class structure. An example of class diagram is shown in figure 2.6.

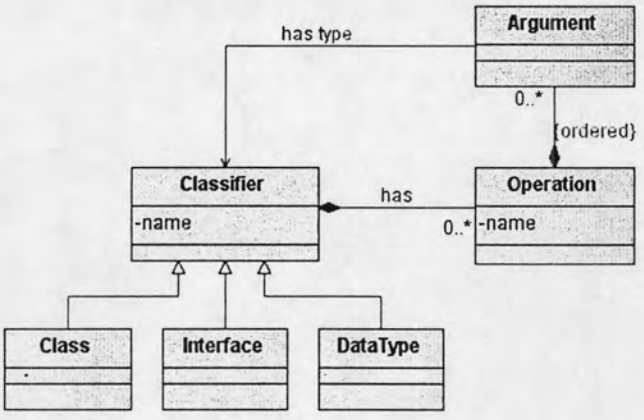


Figure 2.6 Example of class diagram

An interaction of an object cluster is represented by either UML sequence diagram or UML collaboration diagram. While both types of diagrams are possible for modeling object interactions, they have different focuses. A sequence diagram represents an interaction of a group of objects over time. It presents the order of time when messages are sent between objects. Figure 2.7 shows an example of a sequence diagram. From the diagram, upper messages are sent before lower messages.

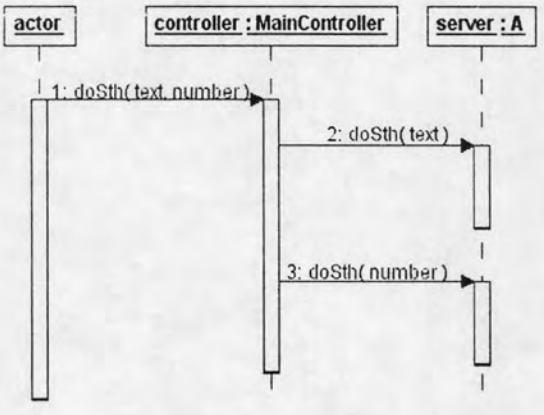


Figure 2.7 Example of sequence diagram

A collaboration diagram can represent the same information as a sequence diagram; however, its focus is not the time of message sending. The main focus of a collaboration diagram is object roles. It shows an interaction in form of messages

sending between objects of various roles. Both sequence diagram and collaboration diagram represents dynamic behavior of a system.

### 2.1.4 Fault Model & Verification

Binder [8] classifies software testing into 2 categories, fault-directed testing and conformance-directed testing. Fault-directed testing is a category of software testing which focuses on revealing defects from a particular fault model. This type of testing relies on verifying test execution output to identify whether there is a failure occurred. Test cases for this type are generated based on the fault model. For example, in path-based testing, test cases are generated from control flow graph, since the fault model assumes that failures occurred from invalid control flow.

Conformance-directed testing is different. The major purpose of conformance-directed testing is to ensure conformance of the software under test against its requirements or specification. In other words, this type of software testing is for revealing discrepancy between the software under test and its requirement or specification. There is no particular fault model. On the other hand, anything that causes discrepancy is a fault. These two categories are not always mutually exclusive. A test method may cover both aspects at the same time.

### 2.1.5 Polymorphism & Adequacy Criteria

Polymorphism along with its based feature, inheritance, is the major obstacle in object-oriented software testing. A lot of early research in object-oriented software testing area tries to address whether software with inheritance and polymorphism requires specific test techniques.

Perry and Kaiser [1] presented adequacy criteria for object-oriented software testing using formal analysis based on the axioms for test adequacy evaluation proposed by Weyuker [39]. Their result is rather counter-intuitive; for example, their result shows that a test set for a particular method is not adequate for testing the same method inherited by a subclass. This is quite unusual, since two identical methods should be adequately tested with the same test set.

Barbey and Strohmeier [4] discussed polymorphism in similar fashion. Polymorphism usually relies on dynamic binding. Since dynamic binding allows the target of a call to be decided on runtime, it is impossible to identify the target of the call during compile-time. Testing is required to exercise every possibility of binding. This means that for a call to a class in a program, all of its subclasses must be tested against this call in order to say that it is practically tested.

## 2.2 Related Works

### 2.2.1 SeDiTeC

SeDiTeC [26], presented by Fraikin and Leonhardt, is a tool that creates and executes test for Java program based on UML sequence diagrams. The diagrams are applied as test specification, which the tool uses to generate test drivers. In addition to verification of return values, the Java program under test is instrumented to check whether the program sends messages, or "call operations", according to what is specified in the diagrams. Since several diagrams can be combined to form a test scenario, sharing of a diagram, which represents a common interaction, is possible.

Strong features of SeDiTeC are message path verification and ability to automatically generate test driver and test stub. Shown in figure 2.8 is an example of a sequence diagram for SeDiTeC. The test driver as the actor of the interaction is generated by SeDiTeC. However, SeDiTeC is not capable for examining results in a case where some refinement is applied to the implementation under test. In this case, human decision is required to identify whether the test execution is a failure or not. Moreover, its verification process does not take into account of the possibility of polymorphic interactions of the implementation under test.

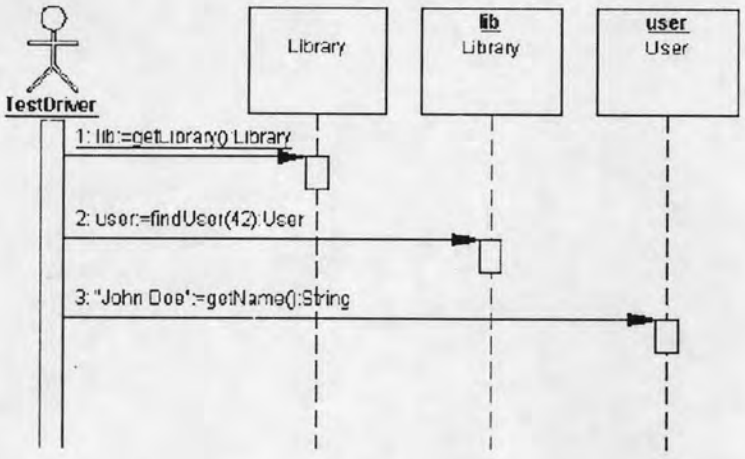


Figure 2.8 Example of sequence diagram for SeDiTeC

SeDiTeC is an effective test driver generation. Nevertheless, it is not a test case generation tool. Inputs of this tool must be sequence diagrams of a particular scenario. Test inputs are required to be supplied manually by test designers. Moreover, test output validation procedure is also required to be included as sequence diagrams.

### 2.2.2 TACCLE and Sequence Diagram Based Testing

From TACCLE approach presented by Chen et al. [27, 28], they expressed how to apply the approach to test object-oriented software based on UML sequence diagrams and UML collaboration diagrams [29]. TACCLE is an approach for testing object-oriented software on class level and cluster level using formal specification. In cluster level testing, the approach uses Contract, a formal specification language, as test specification. Contract represents interactions between objects in a cluster.

For either UML sequence diagrams or UML collaboration diagrams, a diagram corresponding to a use case is converted into specification by rules which are presented by Chen. Characteristics like message path, conditional message, and message iteration are extracted from the diagram to the specification. Once the specification is created, test is performed based on the specification as in TACCLE approach.

This approach is effective due to the automated power of formal specification. However, it also requires formal class level testing based on algebraic specification, which may not be available in most cases.

### 2.2.3 Using UML Collaboration Diagrams for Static Checking and Test Generation

Abdurazik and Offutt suggested several ways to generate test cases from collaboration diagrams in [30]. They noticed that the main benefit of generating test cases from design artifact is that defects could be detected earlier than generating test cases from implementation, which occurs later in development process. They also addressed that mistakes in design artifacts are probably detected while test cases are generated. They gave some basic definitions of elements in collaboration diagrams. The focus is on links and pairs of links. Several possible ways to perform static checking on the information gathered from collaboration diagrams are also shown in their paper.

Using test cases generated from collaboration diagrams, dynamic testing is also possible to be performed. However, they did not describe it in the paper. Instead, they showed that the implementation can be instrumented, and coverage analysis could be performed based on collaboration diagrams.

### 2.2.4 Criteria for Testing Polymorphic Relationships

Regarding polymorphic adequacy criteria, Alexander and Offutt presented four test adequacy criteria for testing polymorphic relationships in object-oriented software in integration testing level [31]. These criteria are designed to support their test approach [32], which is an altered version of coupling-based testing technique [33]. Basically the criteria are defined around "defs" and "uses" like criteria for data flow testing [34]. Concerns about inheritance and polymorphism are taken into account, since two of the criteria require all subclasses to be tested in the interaction context of their corresponding superclasses.

### 2.2.5 Test Adequacy Criteria for UML Design Models

Andrews et al. proposed adequacy criteria for testing UML design models, which include class diagrams and interaction diagrams [35]. 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 asscciation 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.

### 2.2.6 Other UML Based Testing

In [36], Wu et al. present an approach to perform integration testing on COTS (Commercial Off-The-Shelf) software components. Due to absence of source code, the technique is based on interaction diagrams (UML collaboration/sequence diagrams) and UML statechart diagrams. Their focus is on covering all state transitions, message paths, and data dependency (similar to def-use in data flow testing [34]).

Briand and Labiche propose another approach for testing on system level based on various UML diagrams, including use case diagrams, activity diagrams, and sequence/collaboration diagrams [37]. An approach to generate test case from UML statechart diagrams is presented by Offutt and Abdurazik in [38]. These researches focus on different aspects of testing, but their common characteristic is that they are specification level testing. This means that test specification is derived mostly from design specification, while source code is virtually ignored.