INTEGRATION TESTING

AN APPROACH OF INTEGRATION TESTING
FOR
OBJECT-ORIENTED PROGRAMS

By

ZHE (JESSIE) LI, B.E.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

McMaster University

©Copyright by Zhe (Jessie) Li, May 2007

MASTER OF SCIENCE (2007)
(Computer Science)

McMaster University
Hamilton, Ontario

TITLE: An Approach to Integration Testing for Object-Oriented Programs
AUTHOR: Zhe (Jessie) Li, B.E. (McMaster University)
SUPERVISOR: Professor Tom Maibaum
NUMBER OF PAGERS: ix, 126

Abstract

Object-oriented programs have many unique features that are not present in conventional programs, such as Inheritance, Polymorphism, Dynamic binding and Encapsulation, etc. Hence, testing object-oriented programs using only traditional techniques is unlikely to find the faults associated with these features. A study shows that approximately 40% of software errors can be traced to component interaction problems discovered during integration. Integration testing is an important part of the testing process. However, few integration testing techniques have been systematically studied or defined.

The goal of this research is to develop an approach for automatic test case execution at the integration level. The approach is based on the concept of Coordination Contract, which was developed by J. Fiadeiro and L. Andrade. A coordination contract is a connection between a group of objects. Through contracts, rules and constraints are superposed on the behavior of the participants without interfering with their implementation. Due to the contract's character, integration test case execution and test result evaluation are suitably implemented by contracts. A tool has been developed to automatically generate the relevant contracts to implement integration test cases generated by some mechanism for test case generation.

In recent years, more and more software developers use the Unified Modeling Language (UML) and corresponding visual modeling tools to design and develop their application software. So we are using UML sequence diagrams and class diagrams as integration testing specifications. Actually, there are few practical tools to generate test cases from UML, and even fewer tools to execute test cases. Therefore, the result of this research will play an important role in testing object-oriented programs at the integration level. Our accomplishment makes some progress in the integration testing for object-oriented programs.

Acknowledgement

This thesis is the result of nineteen months of work whereby I have been accompanied and supported by many people. I would like to express my warmest gratitude to all those who gave me the possibility to complete this thesis.

I am deeply indebted to my supervisor Prof. Dr. Tom Maibaum from McMaster University. I am grateful for his offering me a chance to be one of his students. His wide knowledge and his logical way of thinking have been of great value for me. His understanding, encouraging and personal guidance have provided a good basis for the present thesis. His kind help, stimulating suggestions helped me in all the time of research for and writing of this thesis. Without his help, this thesis could not be possible to get there.

I would like to express my deep and sincere gratitude to Prof. Dr. Kamran Sartipi from McMaster University. His extensive discussions around my work have been very helpful for this thesis.

I wish to express my warm and sincere thanks to Prof. Dr. Alan Wassyng from McMaster University for his valuable advices and friendly help. His kind support and guidance have been of great value in this thesis.

I would also like to thank Dr. Marcelo Frias from University of Buenos Aires for offering a tool to generate test data, and Georgios Koutsoukos from ATX Software for providing latest information about the tool CDE.

Especially, I would like to give my special thanks to my husband Jeremy whose patient love and help enabled me to complete this work, and my parents who are always encouraging and supporting me.

The financial support of the department of computing and software in McMaster University and my supervisor is gratefully acknowledged.

I wish I could thank everyone who helped me. But I cannot list all the names here. In one word: Thank You, Everyone!!

Sincerely,
Jessie Li
Hamilton, Canada
May, 2007

Contents

Abstract ii

Acknowledgement iv

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Related Work . . . 1
- 1.3 Contribution . . . 2
- 1.4 Thesis Contents . . . 3

2 Object-Oriented Programming 5
- 2.1 Introduction . . . 5
- 2.2 Features . . . 5
- 2.3 Design Patterns . . . 7
  - 2.3.1 Introduction . . . 7
  - 2.3.2 Examples . . . 8

3 Software Testing 11
- 3.1 Introduction . . . 11
  - 3.1.1 Software Testing vs. Formal Verification . . . 11
  - 3.1.2 Validation vs. Verification . . . 12
  - 3.1.3 Testing Level . . . 12
  - 3.1.4 Black Box vs. White Box . . . 14
- 3.2 Integration Testing . . . 15
  - 3.2.1 Introduction . . . 15
  - 3.2.2 Integration Testing Patterns . . . 15
  - 3.2.3 Integration Faults . . . 16
- 3.3 Related Work: A Survey of OO Integration Testing Techniques 17
  - 3.3.1 State-based Testing . . . 18
  - 3.3.2 Mutation-based Testing . . . 18
  - 3.3.3 Data-flow based Testing . . . 19
  - 3.3.4 Control-flow based Testing . . . 19
  - 3.3.5 Event-based Testing . . . 20

3.3.6 Formal Specification based testing 20
3.3.7 UML-based Testing 21

4 Unified Modeling Language 25
4.1 Introduction 25
4.2 Sequence Diagrams 26
4.2.1 The Basics 27
4.2.2 Advanced 29
4.3 Class Diagrams 30
4.3.1 Elements 30
4.3.2 Relationship 31
4.4 UML Tools 32

5 Coordination Contract 35
5.1 Concepts 35
5.2 An Example of Coordination Contract 36
5.3 Contract Specification for CDE1.1 38
5.3.1 Syntax 38
5.3.2 Semantics 39
5.4 Micro-Architecture 41
5.5 CDE 44
5.6 Other Notions of Contract 45

6 Test Approaches 47
6.1 Introduction 47
6.2 Test Case Generation 49
6.2.1 Testing Sequence of Message Calls 49
6.2.2 Testing Parameters 51
6.2.3 Testing Return Value 51
6.3 Test Case Coverage 54
6.3.1 Test Coverage for Integration Testing 54
6.3.2 Test Coverage Criteria in Our Approach 58

7 Test Design by Contracts 59
7.1 Introduction 59
7.2 Contract Design 60
7.2.1 Contract for Testing Sequences of Message Calls 60
7.2.2 Contract for Testing Parameters 76
7.2.3 Contract for Testing Returned Value 78
7.3 Test Case Execution 83

8 Prototype 85
8.1 Introduction 85
8.2 Algorithms 88

9 Case Study 106
9.1 Test Process 106
9.2 Test Coverage 111
9.3 Another Case 111

10 Conclusion and Future Work 115
10.1 Conclusion 115
10.2 Future Work 116

Bibliography 118

Appendices 122

A Contracts for Testing "transfterTo(ca, amount)" 123

B Contract One for Testing "cashCheck(check)" 127

List of Figures

2.1 Structure of Proxy Design Pattern [19] 9
2.2 Structure of Chain of Responsibility Design Pattern [19] . . . 9

3.1 V Lifecycle Model 13

4.1 The Thirteen Standard UML Diagrams [23] 26
4.2 An Example of Simple Sequence Diagram. 28
4.3 An Example of Class Diagram. 31

5.1 Coordination Contract Design Pattern [4] 42

6.1 Architecture of Integration Testing Approach. 48

7.1 A Simple Sequence Diagram Example 62
7.2 Control Flow Graph Examples 66
7.3 Control Flow Graph Examples 69
7.4 Control Flow Graph Examples 74
7.5 Sequence Diagram of "transferTo" 82
7.6 Test Execution Process 84

8.1 Class Diagram of the Main Data Structures 86
8.2 Architecture of Integration Testing Approach. 87
8.3 Sequence Diagram XML Node Tree 91
8.4 Sequence Diagram Example - No Frame 95
8.5 Example of Sequence Diagram with Option Frame 95
8.6 Example of Sequence Diagram with Alt Frame 96
8.7 Example of Sequence Diagram with Loop Frame 97
8.8 Class Diagram XML Node Tree 99
8.9 Relationship between three classes. 104

9.1 Sequence Diagram of "transferTo" 107
9.2 Class Diagram of Bank Accounts 107
9.3 Case Study Folder Tree Structure. 110
9.4 Class Diagram of Bank Subsystem 112
9.5 Sequence Diagram of "cashCheck(check)" 113

List of Tables

2.1 Design Patterns Classification by Purpose [19] 7

3.1 The Methodologies of Test Case Design 15

6.1 Coverage criteria based on Sequence Diagrams [33] 57

8.1 Tabular to Define the Location of the Messages to Frames . . 89
8.2 Relationship between symbols and types 105

9.1 Test Data One for Bank Account Integration Testing 109
9.2 Test Data Two for Bank Account Integration Testing 109

Chapter 1

Introduction

1.1 Motivation

Object-oriented programming is a programming paradigm that uses "objects" to design applications and computer programs. It has been commonly used in mainstream software application development since the 1990s. It has many useful features, such as information hiding, encapsulation, inheritance, polymorphism and dynamic binding. These object-oriented features facilitate software reuse and component-based development. However, some types of faults associated with these unique object-oriented features are difficult to detect using only traditional testing techniques.

A lot of research has been done in the field of object oriented testing and various techniques have been developed for testing of object oriented programs. However, only a small part of them address integration testing. A study [29] shows that approximately 40% of software errors can be traced to component interaction problems discovered during integration. Therefore, integration testing is very important for software quality. But the problem is few integration testing techniques have been systematically studied or defined.

1.2 Related Work

The common approaches in the integration testing of object-oriented programs include state-based testing, event-based testing, formal and semi-formal techniques. For example, Gallagher and Offutt [18] extended an existing intra-class testing technique to inter-class testing. This testing approach relied on finite state machines, database modeling and processing techniques, and algorithms for analysis and traversal of directed graphs. H.Y.Chen, T.H.TSE and T.Y.Chen [15] introduced a methodology TACCLE for object-oriented software **T**esting **A**t the **C**lass and **C**luster **LE**vels. This methodology in-

cludes algebraic specifications for the class testing and Contract specifications for the cluster testing. Offutt and Abdurazik [40] first proposed a mechanism that adapted their previously developed criteria for generating test cases from Software Cost Reduction (SCR) specifications to UML statecharts, and a tool named UMLTest has been built to automatically generate test cases from UML statecharts. More techniques are reviewed in Chapter 3.

1.3 Contribution

Our work addresses integration testing for object-oriented programs. We present an approach that effectively tests object-oriented programs at the integration level. We accomplish the automation of both test case generation and test execution for object-oriented programs integration testing.

Our approach is completely based on UML (Unified Modeling Language). We generate test cases from UML sequence diagrams and class diagrams. A test case consists of three parts: the first part is to test the sequence of message calls; the second part is to test parameters; and the third part is to test object interactions post-conditions.

The automation of test execution is achieved by applying the concept of coordination contracts and the well-developed Coordination Development Environment (CDE) tool. Coordination contract is related to the idea of active association in UML. It defines a connection among a group of objects, through which interactions (rules and constraints) can be dynamically superposed over system components without interfering with their implementations. So we implement test cases using the concept of coordination contracts. Contracts are created independently and explicitly so that they can be added and deleted in a "plug and play" mode. CDE supports the use of coordination contracts for Java applications. It generates relevant Java implementation of the contracts on top of the components under test and integrates the contracts with the components into a larger executable test framework.

OO integration test cases are well designed and the test is automatically executed through our approach because of the following reasons:

- The test cases are generated from UML sequence diagrams and related class diagrams. A sequence diagram depicts the sequences of interaction among different objects over a period of time. Therefore, test cases derived from the sequence diagrams will reveal software faults caused by the component's integration effectively through detecting sequences of the message calls, parameters in the messages and post-conditions of the component's interaction.

  UML specifications can be used for both requirement and design. In our approach, UML is a design specification. For the sake of testing

correctness, UML design specification is complete and accurate.

- The test case is implemented using the concept of coordination contract which was introduced by L. Andrade and J. Fiadeiro. A coordination contract defines a connection among a group of objects without interfering with their implementations. Integration testing deals with the interaction among components. Therefore, integration test cases are well suited to be realized by coordination contract.

- CDE is a tool which generates the Java code that adapts components for coordination and that implements the contracts. The test cases automatic execution is achieved by using CDE to transform the generated contracts into a Java implementation of the contracts and to combine the Java version of the contracts with the components under test into a larger executable test framework.

We have developed a tool to take UML sequence diagrams and class diagrams and generate test cases in terms of coordination contracts automatically based on the mechanism of test case generation. The CDE tool generates the contracts and the components together to form a test framework. We use the tool JAT [17] to generate test data based on the Branch Coverage criterion. The test case execution can also be implemented automatically using a test driver to run the test framework with the generated test data.

There are few practicable tools to generate test cases from UML directly, even fewer tools to execute test cases. Our testing methodology implements the automation of test case generation and test case execution. Therefore, our research will play an important role in testing object-oriented programs at the integration level. Our research will fill a gap between the need for integration testing techniques and the lack of such techniques by developing a method that implements automation of test case generation and test execution for object-oriented programs at the integration level.

1.4 Thesis Contents

The remainder of this thesis is organized as follows. In Chapter 2, we provide an overview of object-oriented programming, focusing on the main features and give a general introduction to design patterns, especially about the "Proxy" and "Chain of Responsibility" patterns which are used in Chapter 5. Chapter 3 outlines software testing, especially integration testing, and gives a survey of current object-oriented integration testing techniques. In Chapter 4, we present the basics of the Unified Modeling Language and list some UML modeling tools. Chapter 5 gives a brief introduction to coordination contracts, including the underlying concepts and methodology, contract specification and

CDE. In Chapter 6, we present our test approaches. We introduce the mechanism to generate test cases and how to evaluate test result. Chapter 7 presents the detailed design of test cases in terms of the concept of coordination contract. In Chapter 8, we give an overview of the automated testing tool. We also list the main algorithms used in the tool and their justifications. Chapter 9 is a case study which helps understand the whole test approach. Conclusions and future work are presented in Chapter 10.

Chapter 2

Object-Oriented Programming

In this chapter, we provide an overview of object-oriented programming focusing on the main features and give a general introduction to design patterns, especially the "Proxy" and "Chain of Responsibility" patterns which are used in Chapter 5.

2.1 Introduction

Object-oriented programming is a way of thinking about the process of decomposing problems and developing programming solutions. It views a program as a collection of objects. An object is an encapsulation of state (data values) and behavior (operations). Each object is responsible for specific tasks. It is by the interaction of objects that computation proceeds. The behavior of objects is dictated by the object's class. Every object is an instance of some class. An object will exhibit its behavior by invoking a method in response to a message [12].

Productivity gains from object-oriented development come not only from reusability, but from the reduction of the semantic gap between the "real world" and the program [11].

2.2 Features

There are three major features in object-oriented programming: encapsulation, inheritance and polymorphism.

Encapsulation Encapsulation refers to the creation of self-contained modules that bind processing functions to the data. These user-defined data types are called "classes" and one instance of a class is an "object". For example, in a payroll system, a class could be Manager, and Pat and Jan could be two

instances (two objects) of the Manager class. Encapsulation ensures good code modularity, which keeps routines separate and less prone to conflict with each other.

Inheritance Inheritance is the capability of a class to use the properties and methods of another class while adding its own functionality. The benefit of inheritance is software reusability and code sharing. When behavior is inherited from another class, the code that provides that behavior does not have to be rewritten. Reusable code increases reliability and decreases maintenance cost because of sharing by all users of the code.

Dynamic Binding Dynamic binding is sometimes called "late binding". It is the linking of a routine or object at runtime based on the conditions at that moment. Dynamic binding enables applications and developers to defer certain implementation decisions until run-time. It facilitates a decentralized architecture that promotes flexibility and extensibility. For example, it is possible to modify functionality without modifying existing code.

Dynamic binding allows new objects and code to be interfaced with or added to a system without affecting existing code and eliminates switch statements. This removes the spread of knowledge of specific classes throughout a system, as each object knows what operation to support. It also allows a reduction in program complexity by replacing a nested construct (switch statement) with a simple call. It also allows small packages of behavior, improving coherence and loose coupling. Another benefit is that code complexity increases not linearly but exponentially with lines of code, so that packaging code into methods reduces program complexity considerably, even more than removing the nested switch statement.

Polymorphism Polymorphism means many shapes. A definition given by Meyer [34] is the ability of a variable or argument to refer at run-time to instances of various classes. More precisely, in object-oriented programming, it indicates a language's ability to handle objects differently based on their runtime type. The programmer and the program do not have to know the exact type of the object in advance, so this behavior can be implemented at run time. For example, the command to show the cursor on screen displays a different icon due to its current location on the screen.

Two types of polymorphism are overloading and overriding. Overloading occurs when an object has two or more behaviors that have the same name. The methods are distinguished only by the messages they receive (that is, by the parameters of the method). Overriding allows a subclass to provide a specific implementation of a method that is already provided by one of its

Creational	**Structural**	**Behavioral**
Factory Method	Adapter	Interpreter
Abstract Factory	Bridge	Template Method
Builder	Composite	Chain of Responsibility
Prototype	Decorator	Command
Singleton	Facade	Mediator
	Proxy	Memento
		Flyweight
		Observer
		State
		Strategy
		Visitor

Table 2.1: Design Patterns Classification by Purpose [19]

super classes. The implementation in the subclass overrides the implementation in the superclass. The third type of polymorphism is dynamic binding, mentioned above.

Polymorphism is a very powerful concept that allows the design of amazingly flexible applications. It separates interface from implementation. It allows programmers to isolate type specific details from the main part of the code.

2.3 Design Patterns

2.3.1 Introduction

Design patterns are convenient ways of reusing object-oriented code between projects and between programmers. The idea behind design patterns is to write down and catalog common interactions between objects that programmers have frequently found useful [16].

Design patterns are classified by the criterion purpose which reflects what a pattern does, as shown in Table 2.1. Patterns can have a creational, structural, or behavioral purpose. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility [19].

2.3.2 Examples

Two design patterns, *Proxy* and *Chain of Responsibility*, will be used in coordination contracts underlying micro-architecture described in Chapter 5. We will give more details about these two design patterns in the following.

Proxy. The Proxy pattern is used when you need to represent a complex object by a simple one. The intent of this pattern is to provide a surrogate or placeholder for another object to control access to it [19]. Proxy is applicable in the following cases:

- Remove Proxy. A remove proxy provides a local representative for an object in a different address space.
- Virtual Proxy. A virtual proxy creates expensive objects on demand. This object will not be created until it is really needed.
- Copy-on-write Proxy. A copy-on-write proxy defers copying a target object until required by client actions. This is a special case of the "virtual proxy" pattern.
- Protection Proxy. A protection proxy controls access to the original object.

The structure of the Proxy pattern is represented in Figure 2.1. In the structure, Subject defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected. RealSubject defines the real object that the proxy represents. Proxy maintains a reference that lets the proxy access the real subject. Any request to Proxy is forwarded to RealSubject when it is appropriate, depending on the kind of proxy, mentioned above.

Chain of Responsibility The Chain of Responsibility pattern allows a number of classes to attempt to handle a request, without any of them knowing about the capabilities of the other classes. It provides a loose coupling between these classes; the only common link is the request that is passed between them. The request is passed along until one of the classes can handle it [16].

The structure of the Chain of Responsibility pattern is represented in Figure 2.2. In the structure, Handler defines an interface for handling requests. A client initiates the request to a ConcreteHandler object in the chain. ConcreteHandler handles requests it is responsible for. It can access its successor, another ConcreteHandler. If the ConcreteHandler can handle the request, it does so; otherwise, it forwards the request to its successor [19].

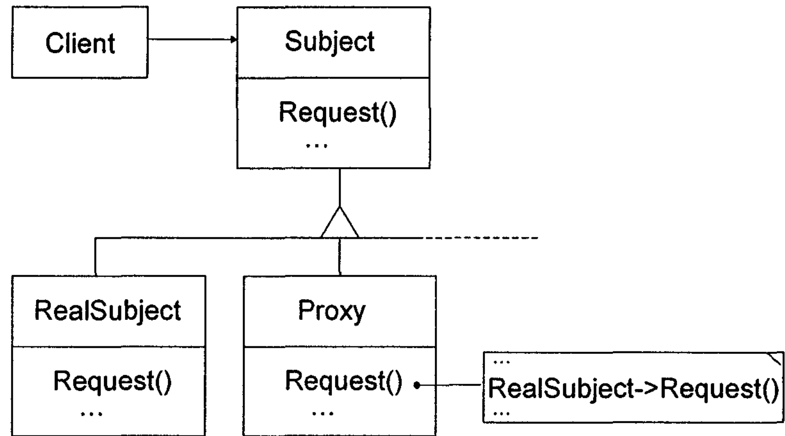


Figure 2.1: Structure of Proxy Design Pattern [19]

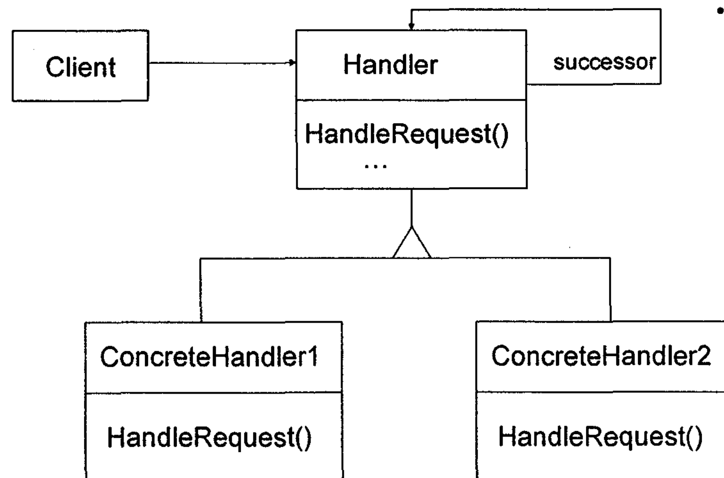


Figure 2.2: Structure of Chain of Responsibility Design Pattern [19]

When a client sends a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.

To summarize, the Proxy pattern, a structural design pattern, is used when you need to represent a complex object by a simple one. The Chain of Responsibility pattern, a behavioral design pattern, uses a chain of objects to handle a request, which is typically an event. Objects in the chain forward the request along the chain until one of the objects handles the event. Processing stops after an event is handled.

In this chapter, we introduced the unique features of object-oriented programming, like Encapsulation, Inheritance, Dynamic Binding, Polymorphism, etc. These object-oriented features facilitate software reuse and component-based development. We also presented design patterns which provide convenient ways of reusing object-oriented code between projects and between programmers. We particularly introduced the "Proxy" structural design pattern and the "Chain of Responsibility" behavioral design pattern because these two patterns are used in coordination contracts underlying micro-architecture, which will be presented in Chapter 5. In the next chapter, we will give a brief introduction of software testing, integration testing, object-oriented programming integration testing and existed testing techniques.

Chapter 3

Software Testing

In this chapter, we outlines software testing especially software integration testing. We present four patterns in integration testing: top-down, bottom-up, big bang and backbone. We list the common faults associated with software integration. Lastly we give a survey of current object-oriented integration testing techniques.

3.1 Introduction

Software testing is an important and integral part of the software development process. It is used to reveal bugs in a system, to assure that the system complies with its specification and to verify that the system behaves in the intended way. Various definitions have been presented for software testing [8, 9]. Myers [38] defines it as:

> "... *the process of executing a program [or system] with the intent of finding errors.*"

Software testing is a vital part of the software development process. It can be used for the purposes of quality assurance, reliability estimation and verification and validation [24]. However, software testing is extremely costly and time consuming. Studies indicate that more than 50% of the cost of software development is devoted to testing [24]. Therefore, there is a need for effective testing strategies.

3.1.1 Software Testing vs. Formal Verification

Formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics [49]. A formal

program verification is a mathematical proof that the program executed according to its intended model of execution meets the specification. It proves the algorithms implemented in the program are correct in the technical sense of being consistent with the specification.

Software testing alone can not prove that a system does not have a defect. Neither can it prove that a system does have a property. Only the process of formal verification can prove that a system does not have a certain defect or does have a certain property, but only if we do not make any mistake in the mathematics.

3.1.2 Validation vs. Verification

IEEE defines validation as "Confirmation by examination and provisions of objective evidence that the particular requirements for a specific intended use are fulfilled". It demonstrates that the software implements each of the software requirements correctly and completely. In other words, the "right product was built".

IEEE defines verification as "Confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled". It is the activity which ensures the work products of a given phase fully implement the inputs to that phase, or "the product was built right".

3.1.3 Testing Level

Generally, there are four levels of software testing carried out: unit testing, integration testing, system testing and acceptance testing. We will introduce each level in details in the following.

The software development life cycle can be represented as a V model, as shown in Figure 3.1. It begins with the identification of a requirement for software and ends with the formal verification of the developed software against that requirement.

On the left side of the diagram, the first three phases produce software specifications.

- The Requirements phase, in which the requirements for the software are gathered and analyzed, to produce a complete and unambiguous specification of what the software is required to do.

- The Architectural Design phase, where a software architecture for the implementation of the requirements is designed and specified, identifying the components within the software and the relationships between the components.

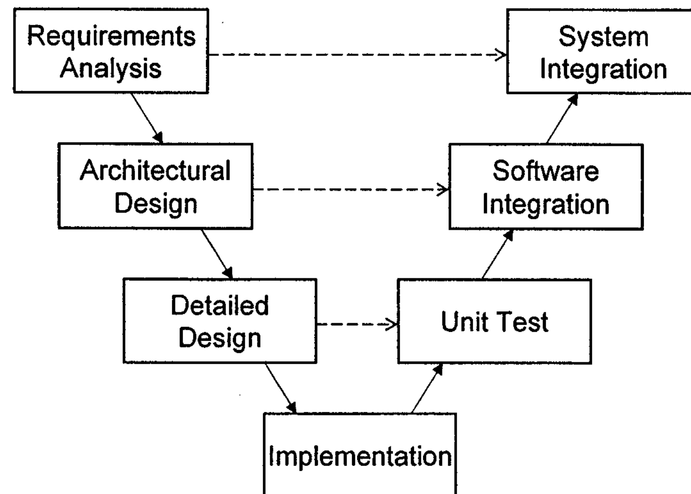


Figure 3.1: V Lifecycle Model

- The Detailed Design phase, where the detailed implementation of each component is specified.

The remaining three phases on the right side of the diagram all involve testing the software at various levels, requiring test specifications (produced from the first three phases) against which the testing will be conducted as an input to each of these phases, horizontally correspondingly.

- The Unit Test phase, in which each component of the software is tested to verify that it faithfully implements the detailed design.

- The Software Integration phase, in which progressively larger groups of tested software components are integrated and tested until the software works as a whole.

  The integration testing of software modules and components is especially concerned with the detection of interface errors. The assumption is made that during unit testing program parts have been tested sufficiently. Therefore the aim of integration tests is to uncover errors which are not detectable during unit testing.

- The System Integration phase, in which the software is integrated to the overall product and tested to show that all requirements are met. It is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

There is another test phase (not shown in the diagram):

- The Acceptance Test phase, in which tests are applied and witnessed to validate that the software faithfully implements the specified user requirements. It can be conducted by the end-user, customer, or client to validate whether or not to accept the product. Acceptance tests represents the customer's interests. Acceptance tests can grow as the system grows, capturing user requirements as they evolve which they always do.

Our work focuses on the integration testing. We will give a brief introduction to software integration testing, the patterns used in software integration testing, the faults associated with software integration and the common integration testing techniques in the following.

3.1.4 Black Box vs. White Box

Black-box and white-box are test design methods. In black-box, the test views the program as a black box. The test is completely unconcerned about the internal behavior and structure of the program. Rather, the tester is only interested in finding circumstances in which the program does not behave according to its specifications. Test data are derived solely from the specifications without taking advantage of knowledge of the internal structure of the program [36]. Synonyms for black-box include: behavioral, functional, opaque-box, and closed-box.

White-box test design allows one to examine the internal structure of the program. The tester derives test data from an examination of the program's logic and often unfortunately, at the neglect of the specification [36]. Synonyms for white-box include: structural, glass-box and clear-box.

It is important to understand that these methods are used during the test design phase, and their influence is hard to see in the tests once they're implemented. Note that any level of testing, we introduced in the previous section, can use any test design methods. For example, unit testing is usually associated with structural test design.

Myers pointed out that exhaustive black-box and white-box testing are, in general, impossible, but he also stated that a reasonable testing strategy might use elements of both. Myers [36] introduced the methodologies for designing test cases, shown in Table 3.1.

Our approach uses black-box testing method. We generate test cases from UML specifications. We will give a detailed introduction in the following chapters.

Black Box	White Box
Equivalence partitioning	Statement coverage
Boundary-value analysis	Decision coverage
Cause-effect graphing	Condition coverage
Error guessing	Path coverage

Table 3.1: The Methodologies of Test Case Design

3.2 Integration Testing

3.2.1 Introduction

Definition 1 by Myers [37] in 1976, p173:

> "*Integration testing is the verification of the interfaces among system parts (modules, components and subsystems).*"

Definition 2 by Beizer [7] in 1984, p141:

> "*Integration testing is aimed at showing inter element consistency under the assumption that elements themselves satisfy element requirements and have passed element level testing.*"

The integration testing of software modules and components is especially concerned with the detection of interface errors. The interfaces between separate parts of a system are determined, for example, by the calling of other system parts, by the use of parameters in procedure calls, by data files or by common global variables. Such connections in a complex system must be specified in detail and tested after realization.

The assumption is made that during unit testing program parts have been tested sufficiently. Therefore the aim of integration tests is to uncover errors which are not detectable during unit testing. In other words, integration testing is the testing of the interactions among components in a subsystem.

In object-oriented programs, unit testing can be considered as intraclass testing, which is the testing of one class only in a component. Interclass testing is the testing of a set of classes composing a system or subsystem. Typically, such classes are not stand-alone entities, but mutually cooperation in several ways.

3.2.2 Integration Testing Patterns

Top-Down Top-down Integration interleaves component integration and integration testing by following the application control hierarchy. Testing and

integration begin early when the top-level components are coded. The cost of test driver development is reduced. There is only one driver to maintain, instead of a driver for every sub-tree. The test cases can be reused to drive lower-level tests. However, stub development and maintenance are the most significant costs and it may be difficult to exercise lower-level components sufficiently.

Bottom-Up Bottom-Up integration interleaves component integration and integration testing by following usage dependencies. This pattern is used for almost any scope or architecture. The advantage of this pattern is that testing may begin as soon as any leaf-level component is ready. However, driver development is the most significant cost. Any revision to a previously tested component is error-prone, costly, and time-consuming.

Big Bang Big Bang Integration attempts to demonstrate system stability by testing all components at the same time. This pattern is used for small to medium systems. Big Bang Integration can result in quick completion of integration testing under favorable circumstances, like a small, well-structured system whose components have received adequate testing, or an existing system where only a few changes have been made. The disadvantages of this pattern is that debugging is difficult due to fewer clues about fault locations.

Backbone Backbone Integration combines top-down integration , bottom-up integration, and big bang integration to reach a stable system that will support iterative development. Backbone integration mitigates the disadvantages of top-down integration and bottom-up integration by curtailing their use at the point at which they lose effectiveness. Top-down integration is used only on the upper control levels and bottom-up integration is restricted to the application subsystems. Big Bang integration of the backbone is preceded by component testing. But careful analysis of system structure and dependencies is necessary.

3.2.3 Integration Faults

A study [29] shows that approximately 40% of software errors can be traced to component interaction problems discovered during integration. Therefore, integration testing is very important for software quality. Most of these detected errors are due to misinterpretation of module specifications [9]. The following is a list of these errors.

- Configuration/version control problems.
- Missing, overlapping, or conflicting functions.

- An incorrect or inconsistent data structure used for a file or database.
- Conflicting data views/usage used for a file or database.
- Violations of the data integrity of a global store or database.
- The wrong method called due to coding error or unexpected runtime binding.
- The client sending a message that violates the server's preconditions.
- The client sending a message that violates the server's sequential constraints.
- Wrong object bound to message (polymorphic target).
- Wrong parameters, or incorrect parameter values.
- Failures due to incorrect memory management allocation/deallocation.
- Incorrect usage of virtual machine, ORB, or OS services.
- Attempt by the Implementation under Test (IUT) to use target environment services that are obsolete or not upward-compatible for the specified version/release of the target environment.
- Attempt by the IUT to use new target environment services that are not supported in the current version/release of the target environment.
- Intercomponent conflicts: thread X will crash when process Y is running, for example.
- Resource contention: the target environment can not allocate resources required for a nominal load. For example, a use case may open up to six windows, but the IUT crashed when the fifth is opened.

3.3 Related Work: A Survey of OO Integration Testing Techniques

Object-oriented programming has been used widely, since it increases software reusability, extensibility, interoperability, and reliability, compared with conventional programming. Software testing is necessary to realize these benefits by uncovering as many programming errors as possible at a minimum cost.

While object-oriented programming poses new challenges for software testing, since objects may interact with one another with unforeseen combinations and invocations, which are much more complex to simulate and test than the hierarchy of modules in conventional programs.

A lot of research has been done on testing object-oriented programs. The following is a survey of common integration testing techniques for object-oriented programs. They include state-based, data-flow based, control-flow based, event-based, formal specification and UML-based testing.

3.3.1 State-based Testing

State-based testing techniques rely on the construction of a finite state machine (FSM) to represent the change of states of the program under test. But for integration testing, the construction of the global finite state machines may become very huge and subject to the state explosion problem. One solution is to design the components into an FSM hierarchy by reducing composite FSMs at each level by means of abstraction. From the testing point of view, test cases can also be selected based on graph traversal.

Gallagher and Offutt [18] extended an existing intra-class testing technique to inter-class testing. This testing approach relied on finite state machines, database modeling and processing techniques, and algorithms for analysis and traversal of directed graphs.

In general, the state-based approach uses interacting finite state machines to model an integrated system. The difficulty of the technique increases when the number of concurrent units increases.

3.3.2 Mutation-based Testing

Mutation testing is used to test the quality of the test suite. This is done by mutating certain statements in the source code and checking if the test code is able to find the errors.

Mutation-based integration testing techniques are based on State-based Mutation Test Criteria (SMTC). The SMTC is a test case selection criterion which can differentiate the state diagram from the mutant state diagram. But SMTC only covers inter-method testing and intra-class testing. Yoon, Choi and Jeon [50] extend it to inter-class testing by applying mutation analysis to the state diagram. It can test the interactions between the public methods from the inheritance and polymorphism. Their techniques include two procedures: test identification procedure and test case selection procedure. Test Identification uses Inheritance Call Graph, drawn from the source code and class diagram, and for the use of the Taxonomy. The test case procedure can cover the inter-class testing by using SMTC. But they don't give more anal-

ysis on the fault coverage. We do not know how efficient this criterion could be. And the ICGraph needs to be drawn from the source code and the class diagrams, which is not an easy job for inexperienced programmers.

3.3.3 Data-flow based Testing

Data flow analysis of a software can be accomplished statically by inspecting the source code and tracking the sequence of the uses of the variables of the program under test without running it, or dynamically by executing it and tracking the sequence of actions. Dynamic data flow analysis is a method for analyzing the sequence of actions on data in a program as it is running. To detect data flow anomalies dynamically, Boujarwah et al [10] introduced a method that inserts software probes into the original source code program to gather information during the program execution. The problem is that conventional probing techniques alone may not be adequate for object-oriented programs. More research in this area is required.

Jorgensen [28] used the decision-to-decision paths (DD-paths) approach from unit testing for integration testing. Module-to-Module paths (MM-paths) were defined as combinations of DD-paths. Leung and White [30] applied extremal values testing concepts to integration testing.

Martena et al [47] extended their previous work in the automatic generation of test cases for a single class to address the problem of inter-class testing. They used data flow analysis to derive a feasible set of test case specifications for interclass testing. A tool is available for the automatic generation of test cases based on the presented technique. But their technique only addressed those problems related to the objects' state. So this technique should be accompanied by other techniques which address the problems related to other object-oriented programs' features, like inheritance, polymorphism and dynamic binding.

3.3.4 Control-flow based Testing

Control-flow based testing is a traditional form of white-box testing. Test cases are designed to cover certain elements of the graph created from the source code to describe the flow of control. Control-flow based and data-flow based technologies are often used together to supplement each other. Spillner [45] developed a pair of integration testing techniques based on unit testing methods. One technique adapted control flow technology to test software modules by testing as many different sequences of calls as possible. The other adapted data flow technology to test for data flow anomalies across procedure calls. Linnenkugel and Müllerburg [31] also used data flow and control flow technology to develop criteria for selecting integration test data.

3.3.5 Event-based Testing

Event-flow technique is an adaption of data-flow techniques to the inter-class level. Whereas data-flow focuses on the effect of definitions on subsequent uses of variables, event-flow focuses on the effect of inter-class level events on subsequent events. Liu and Dasiewicz introduced an event-flow techniques to select system level test cases using a hierarchical state machine model of the system [32]. The strategy is to require test cases which stress the interactions between related events rather than to simply exercise each transition in the model. The advantages of the technique are that selected test cases are meaningful and similar to the way that human testers would select them.

Event-flow handles hierarchical and concurrent specification in a natural way. However, for systems with a large number of concurrent events, the number of selected test cases grows exponentially.

3.3.6 Formal Specification based testing

A lot of research work has been done for the testing of object-oriented programs at the intra-class level using formal specifications. However, relatively little work has been done on the inter-class level testing.

Contract Specification The contract specification, proposed by Helm et al in 1990 [26], describes the behavioral dependencies and the interactions among the cooperating classes in a given cluster (a group of classes). The main syntax of a contract specification is the message-passing rule (mp-rule). Each mp-rule in the contracts is individually used for cluster-level testing. Chen et al [15] introduced a methodology of Testing at the Class and Cluster LEvels (TACCLE) for object-oriented software. This methodology includes algebraic specification as a basis for the class testing and contract specification as a basis for the cluster testing.

For the cluster testing, they defined two testing procedures for individual mp-rules and for composite mp-rules. A **TIM approach** for **T**esting the interactions using **I**ndividual **M**p-rules was presented for individual mp-rules. The implementation of the TIM approach need only write a sub-module AMP to **A**nalyze the body of the given **MP**-rule to find the messages passing across different classes in the cluster. The whole system integrating all the modules and algorithms has not been considered yet.

The other part analyzes the interactions according to composite message-passing sequences [15]. The TACCLE approach, however, has to detect the contract specification and find the test cases properly. In their system, some steps can only be done manually for general situations. This requires much more test effort. And it does not consider the non-deterministic and concurrence issues in Java programs.

Other specifications Besides the contract specification, some research has been done on other formal specifications. Different specifications have different strengths and weaknesses in supporting object-oriented program testing. As a research trend, more than one formal specification is combined together to test one program. For example, state-based specification such as finite state machines and Petri Nets, model-based specification such as Object Z, and process algebra such as CSP. Change et al [13] generate test scenarios based on FSM specifications and Object Z; Smith and Derrick [43] combined data structure modeling in Object Z and communication behavioral descriptions in CSP. The combined models are very comprehensive for testing all the features of object-oriented programs, but it is really a challenge for the software testers because of the increased complexity.

3.3.7 UML-based Testing

The Unified Modeling Language(UML) is a language for specifying, constructing, visualizing and documenting artifacts of software-intensive systems. More and more software developers like to use UML and associated visual modeling tools as a basis for the design and implementation of their component-based applications. Even though UML is widely employed in industry and research, only a little part of the reported literature has addressed its use in the testing phase so far. These methods generates test cases for various testing levels from various UML diagrams. The following are examples of the integration testing approaches based on different UML diagrams.

- Statecharts. UML statecharts are based on finite state machines using an extended Harel state chart notation, and are used to describe the behavior of an object.

  Offutt and Abdurazik [40] adapted their previously developed criteria for generating test cases from Software Cost Reduction (SCR) specifications to UML statecharts, and a tool named UMLTest has been built to automatically generate test cases from UML statecharts. This tool, UMLTest, is the first tool that can generate test cases from UML specifications. But the tool is not available any more. Furthermore, this approach is only able to generate test cases for a single component.

  Hartmann, et al [25] also introduced an approach to generate test cases automatically from the UML Startcharts diagrams. They construct a global behavioral model from the multiple statecharts, each one representing a component. Test cases can be derived from the model by using the test generation engine and executed with the help of the test execution tool. But the global behavioral model used in their approach can

not support internal data conditions of these state machines influencing the transition behavior. Concurrent states are not supported as yet.

- Sequence Diagrams. Basanieri and Bertolino [6] introduced an approach to generate test cases based on the use case diagrams and sequence diagrams, called Use Interaction Test (UIT). The use case drives the integration testing by incremental strategy. They start by analyzing the low-level functionalities described in a sub-Use Case, and then progressively put them together, until the whole system described in the main Use Case is obtained. And for each Use Case, they derived the message sequence from the corresponding sequence diagram. Then they analyzed the message sequence by using the Category Partition method. This testing approach is not supported by a tool to automate the process.

- Collaboration Diagrams. It is suitable to consider collaboration diagrams for integration testing because collaboration diagrams specify the interactions among a set of objects. The benefits of using collaboration diagrams are generating test data using data flow or control flow techniques before the code generation, and stack checking of specification and code.

  Abdurazik and Offutt [1] presented an approach to generate test data to check the software that is presented by collaboration diagrams. Testing can be either static or dynamic. For dynamic checking, they assumed that each operation has a collaboration diagram, which represents a complete trace of messages during the execution of the operation. They also introduced an algorithm to insert instrumentation into the code for testing the events sequences produced by the system match to the message sequences derived from the collaboration diagram. But they only focused on the test criteria and they did not investigate test generation.

- Activity Diagrams. Activity diagrams can be used to model dynamic aspects of a group of objects, or the control flow of an operation.

  Wang Linzhang et al [48] proposed an approach to generate test cases directly from UML activity diagrams using a gray-box method. A gray-box method, from the designer's viewpoint, generates test cases from the high level design model which represents the expected structure and behavior of the system under test. Gray-Box methods can overcome the shortcomings of black-box testing methods and white-box testing methods. First, they traverse the activity diagram to generate the test scenarios. And for each test scenario, they derive test cases based on the category partition method. Test suites are composed from the test cases for all the test scenarios. A tool named UMLTGF was developed

to automate the most part of this method. This method is not for integration testing.

- Transforming Interaction Diagrams. Chen [14] developed an approach for Object-Oriented cluster-level tests based on UML. This approach uses guidelines to transform the sequence diagram or collaboration diagram into contract specification, proposed by Helm et al in 1990 [26]. Then use TACCLE methodology, introduced by Chen et al in 2001 [15], for the cluster tests of Object-oriented softwares. This approach makes the cluster-level testing easier for those object-oriented softwares specified by UML. But this method need to transform UML description into another formal description and then derive the test from the latter. This approach need to consider the transformation cost.

We have presented an overview of research work on integration testing for object-oriented program. In summary, the state-based approach uses interacting finite state machines to model an integrated system. The difficulty of the technique increases when the number of concurrent subsystems increases. The traditional techniques on data-flow and control-flow have been adapted to integration testing. But one technique cannot address all the problems related to object-oriented programs' features. Event-flow handles hierarchical and concurrent specification in a natural way. However, for systems with a large number of concurrent events, the number of selected test cases grows exponentially. Formal specification based testing techniques play an important role in software testing. However, the complexity of the combined formal models may be a serious problem for software testers. UML-based techniques conduct testing by representing UML specification with a formal notation. With the variety of the UML diagrams, the integration of formal and practical techniques is a promising area.

It seems like there are a reasonable amount of integration testing techniques for object-oriented programs. However, most of work still remains in research phase. Few integration testing techniques has been systematically designed or studied. Some of them only proposed approaches to integration testing but not practical at all. Furthermore, they don't have practical tools available to apply their methods in the real application. A need for systematic and practical methodologies to integration testing for object-oriented programs becomes exigent.

In this chapter, we gave a brief introduction to software testing. We presented the difference between software testing and formal verification, and compared verification with validation. There are four software testing phases in the software development life cycle. We focused on software integration testing and introduced object-oriented programming integration testing. We gave

a survey of the reported literature about the methods generating test cases for integration test from various UML diagrams. In the next chapter, we will present the basics of the Unified Modeling Language and list some UML modeling tools.

Chapter 4

Unified Modeling Language

In this chapter, we give an overview of UML and its thirteen diagrams. An sequence diagram is used primarily to show the interactions between objects in the sequential order that those interactions occur. Our approach uses sequence diagrams to generate test cases for integration level testing. We introduce sequence diagrams and class diagrams from the thirteen UML diagrams in detail. We also present a short list of UML-based tools on the market and why we choose Omondo EclipseUML tool in our approach.

4.1 Introduction

The Unified Modeling Language (UML) is "a language for visualizing, specifying, constructing and documenting the artifacts of software systems". UML is an object-oriented analysis and design language from the Object Management Group (OMG). UML can be used for business modeling and for modeling other non-software systems too. Using any one of the large number of UML-based tools on the market, one can analyze a future application's requirements and design a solution that meets them, representing the results using UML 2.0's thirteen standard diagram types [21]. A diagram is a model in a view, a view consisting of one or more models.

What can one Model with UML? UML 2.0 defines thirteen types of diagrams, shown in Figure 4.1, divided into three categories: six diagram types represent static application structure; three represent general types of behavior; and four represent different aspects of interactions [21]:

- **Structure Diagrams** include class diagram, object diagram, component diagram, composite structure diagram, package diagram, and deployment diagram.

- **Behavior Diagrams** include the use case diagram (used by some method-

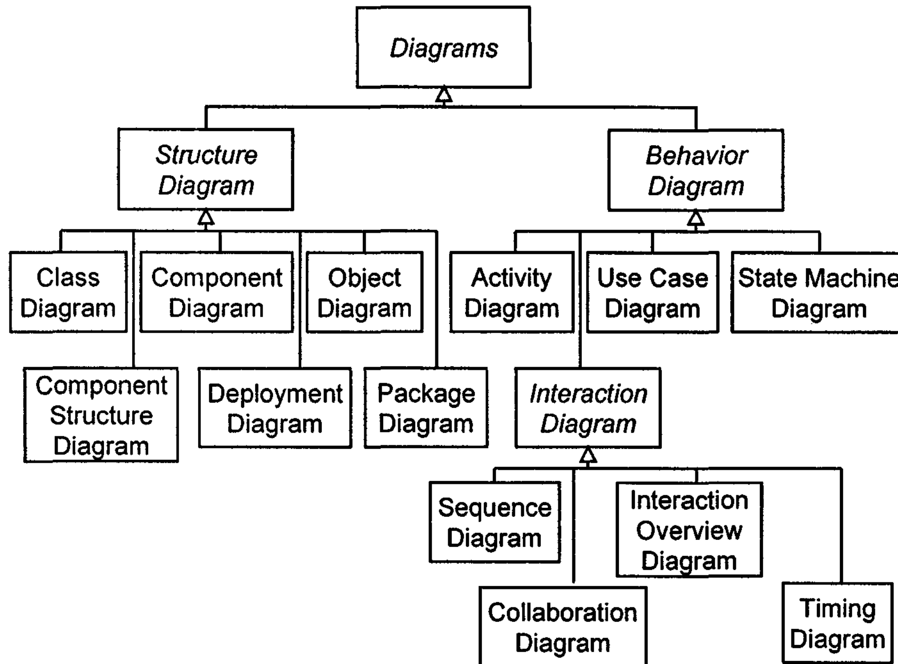


Figure 4.1: The Thirteen Standard UML Diagrams [23]

ologies during requirements gathering); activity diagram, and state machine diagram.

- **Interaction Diagrams**, all derived from the more general behavior diagram, include the sequence diagram, communication diagram, timing diagram, and interaction overview diagram.

Our approach generates test cases from sequence diagrams and class diagrams, so we will introduce these two diagrams in detail in the following.

4.2 Sequence Diagrams

A sequence diagram depicts an interaction by focusing on the sequence of messages that are exchanged, along with their corresponding event occurrences on the lifelines of the objects introduced in the interaction. A sequence diagram includes time sequences but does not include object relationships. A sequence diagram can exist in a generic form (describes all possible scenarios) and in an instance form (describes one actual scenario). Sequence diagrams and communication diagrams express similar information, but show it in different ways [23].

The sequence diagram is used primarily to show the interactions between objects in the sequential order that those interactions occur. Another primary use of sequence diagrams is in the transition from requirements expressed as use cases to the next and more formal level of refinement. Use cases are often refined into one or more sequence diagrams. In addition to their use in designing new systems, sequence diagrams can be used to document how objects in an existing system currently interact.

4.2.1 The Basics

Most sequence diagrams will communicate what messages are sent between a system's objects as well as the order in which they occur. The diagram conveys this information along the horizontal and vertical dimensions: the vertical dimension shows, top down, the time sequence of messages as they occur, and the horizontal dimension shows, left to right, the object instances that the messages are sent to.

- **Lifelines** Lifelines represent either roles or object instances that participate in the sequence being modeled. Lifeline notation elements are placed across the top of the diagram.

- **Messages** The first message of a sequence diagram always starts at the top and is typically located on the left side of the diagram for readability. Subsequent messages are then added to the diagram slightly lower than the previous message. To show an object sending a message to another object, you draw a line to the receiving object with a solid arrowhead (if a synchronous call operation) or with a stick arrowhead (if an asynchronous signal). The message name is placed above the arrowed line. The message that is being sent to the receiving object represents an operation or method that the receiving object's class implements. Return messages are an optional part of a sequence diagram. A return message is drawn as a dotted line with an open arrowhead back to the originating lifeline, and above this dotted line is placed the return value from the operation. To indicate an object calling itself, we draw a message and connect the message back to the object itself instead of connecting it to another object.

- **Combined Fragments** A combined fragment is used to group sets of messages together to show conditional flow in a sequence diagram. So we could draw a control-flow graph corresponding to a sequence diagram, which will be introduced in chapter 7. A combination fragment element is drawn using a frame. A keyword *alt* (*opt* or *loop*) is placed inside the frame's namebox, representing an alternative combination fragment

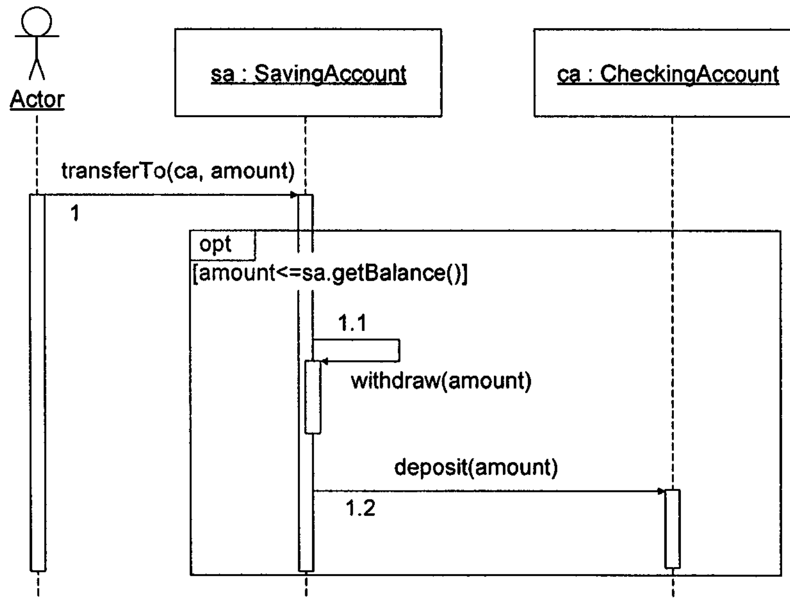


Figure 4.2: An Example of Simple Sequence Diagram.

element (or option or loop). Inside the frame's content area, the alt (opt or loop) guard is placed towards the top left corner, on top of a lifeline. More details of alternatives, options and loops are presented in the following:

- **Alternatives** Alternatives are used to designate a mutually exclusive choice between two or more message sequences. Alternatives allow the modeling of the classic "if then else" logic.
- **Options** The option combination fragment is used to model a sequence that will occur, given a certain condition; otherwise, the sequence does not occur. An option is used to model a simple "if then" statement.
- **Loops** The loop combination fragment models a repetitive sequence. Loops allow the modeling of the "while" logic.

An example of basic sequence diagram is shown in Figure 4.2, which is in an instance form. The sequence diagram depicts the interactions of transferring *amount* many money from the saving account to the checking account under the condition of the saving account's balance greater than *amount*. We have three lifelines: "driver", "sa : SavingAccount" and "ca

: CheckingAccount". The lifeline "driver" is a role. The lifelines "sa : SavingAccount" and "ca : CheckingAccount" are object instance *sa* of class *SavingAccount* and object instance *ca* of class *CheckingAccount*. The first message is $transferTo(ca, amount)$. The message $withdraw(amount)$ indicates the object calling itself. There is a combined fragment of type "options" in the diagram. $amount < sa.getBalance()$ is the condition of the control flow, meaning if $amount < sa.getBalance()$, then call message $withdraw(amount)$ followed by message $deposit(amount)$.

4.2.2 Advanced

In addition to the basic elements, which should depict most interactions taking place in a common system, there are more advanced notion elements that can be used in a sequence diagram.

- **Referencing Another Sequence Diagram** The "Interaction Occurrence" element is introduced in UML2.0. Interaction occurrences add the ability to compose primitive sequence diagrams into complex sequence diagrams. With these we can combine (reuse) the simpler sequences to produce more complex sequences. This means that we can abstract a complete, and possibly complex, sequence as a single conceptual unit. An interaction occurrence element is drawn using a frame. The text "ref" is placed inside the frame's name box, and the name of the sequence diagram being referenced is placed inside the frame's content area along with any parameters to the sequence diagram.

- **Gates** Gates can be an easy way to model the passing of information between a sequence diagram and its context. A gate is merely a message that is illustrated with one end connected to the sequence diagram's frame's edge and the other end connected to a lifeline.

- **Combined Fragments (Break and Parallel)** The break combined fragment is almost identical in every way to the option combined fragment, with two exceptions. First, a break's frame has a name box with the text "break" instead of "option". Second, when a break combined fragment's message is to be executed, the enclosing interaction's remainder messages will not be executed because the sequence breaks out of the enclosing interaction. In this way the break combined fragment is much like the break keyword in a programming language like C++ or Java. Breaks are most commonly used to model exception handling. Today's modern computer systems are advancing in complexity and at times perform concurrent tasks. When the processing time required to complete portions of a complex task is longer than desired, some systems handle

parts of the processing in parallel. The parallel combination fragment element needs to be used when creating a sequence diagram that shows parallel processing activities.

Our approach generates test cases from sequence diagrams, which do not have the advanced notion elements. We will consider those sequence diagrams with advanced notion elements in our future work.

4.3 Class Diagrams

Definition: A class diagram is a diagram showing a collection of classes and interfaces, along with the collaborations and relationships among classes and interfaces.

A class diagram consists of a group of classes and interfaces reflecting important entities of the business domain of the system being modeled, and the relationships between these classes and interfaces. The structure of a system is represented using class diagrams. A class diagram is a static view of a system.

4.3.1 Elements

A class diagram consists of the following elements that represents the system's entities:

- Class A class represents an entity of a given system that provides an encapsulated implementation of certain functionality of a given entity. The UML representation of a class is a rectangle containing three compartments stacked vertically, as shown in Figure 4.3. The top compartment shows the class's name. The middle compartment lists the class's attributes. The bottom compartment lists the class's operations.

- **Interface** An interface is a variation of a class. An interface provides only a definition of business functionality of a system. A separate class implements the actual business functionality. An interface is drawn like a class, but the top compartment of the rectangle also has the text "<< *interface* >>".

- **Package** A package provides the ability to group together classes and/or interfaces that are either similar in nature or related. Drawing package starts with a large rectangle with a smaller rectangle (tab) above its upper left corner, the package name is written in the smaller rectangle area. Then you have two ways to display package's membership: place all the members within the larger rectangle; place all the members outside the rectangle, a line is drawn from each class/interface to a circle that has a plus sign inside the circle attached to the package.

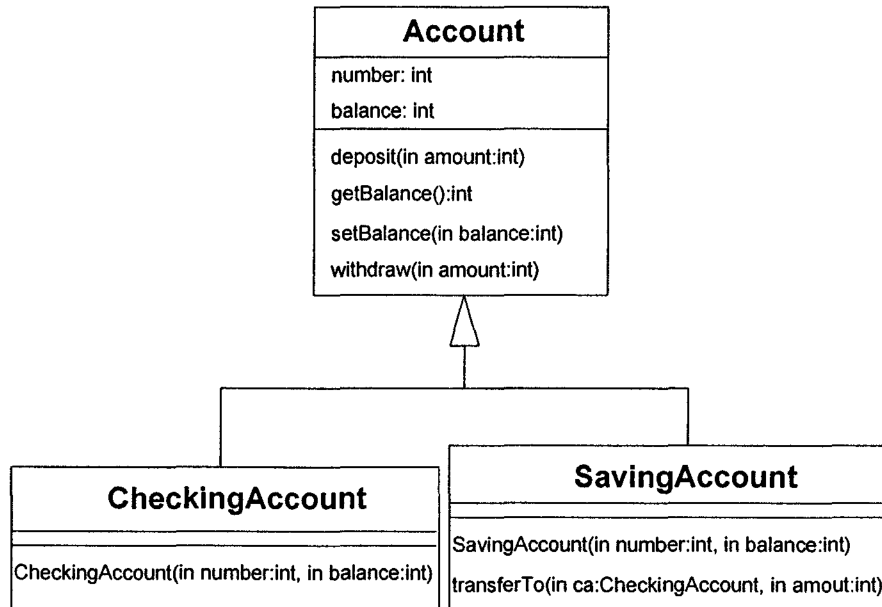


Figure 4.3: An Example of Class Diagram.

4.3.2 Relationship

In class diagram, we can see the relationship between the classes. The following shows the kinds of relationships between classes:

- **Association** When two classes are connected to each other in any way, an association relationship is established.
  - Multiplicity Multiplicity association is indicated by a solid line between the two classes. At either end of the line, you place a role name and a multiplicity value.
  - Directed Association Association between classes is bi-directional by default. You can define the flow of the association by using a directed association. The arrowhead identifies the container-contained relationship.
  - Reflexive Association Reflexive association models a class which has a variety of responsibilities. To represent a reflexive association relationship, you could draw a solid line from the class to itself.
- **Aggregation** When a class is formed as a collection of other classes, it is called an aggregation relationship between these classes. It is also

called a "**has-a**" relationship. Aggregation is a special type of association used to model a "whole to its parts" relationship. In basic aggregation relationships, the life cycle of a part class is independent from the whole class's life cycle. To represent an aggregation relationship, you draw a solid line from the parent class to the part class, and draw an unfilled diamond shape on the parent class's association end.

- **Composition** The composition aggregation relationship is just another form of the aggregation relationship, but the child class's instance life cycle is dependent on the parent class's instance life cycle. Composition relationship is drawn like the aggregation relationship, but the diamond shape is filled.

- **Inheritance/Generalization** Inheritance is an very important concept in Object-Oriented design. It refers to the ability of one class (child class) inherits the identical functionality of another class (super/parent class) and then add new functionality of its own. It is also called "**is-a**" relationship. Inheritance is also sometimes called generalization, because the is-a relationships represent a hierarchy between classes of objects. To model an inheritance on a class diagram, a solid line is drawn from the child class with a closed, unfilled arrowhead pointing to the super class.

- **Realization** In a realization relationship, one entity (normally an interface) defines a set of functionalities as a contract and the other entity (normally a class) "realizes" the contract by implementing the functionality defined in the contract. Realization relationship is drawn like the inheritance relationship, but the line is dotted instead of solid one.

4.4 UML Tools

There are a lot of UML development tools, most of which implement a particular methodology. One may find a tool which is suitable for the application one is developing or the organization one is working for. The following introduces some of UML tools:

- **IBM®Rational®Software Architect and Modeler** IBM Rational Software Modeler is a Unified Modeling Language 2.0-based visual modeling and design tool for architects, systems analysts and designers who need to ensure that their specifications, architectures and designs are clearly defined and communicated. IBM®Rational®Software Architect is an integrated design and development tool that leverages model-driven development with the UML for creating well-architected applications and services [44]. It is a commercial software. It represents the combined

fragment notion elements in sequence diagram by its "note" element, which can only provide a graphical description instead of any logical relationship description.

- **OMONDO EclipseUML** Omondo EclipseUML is a modeling software offering full native integration with Eclipse. Omondo EclipseUML solution has been developed especially and uniquely for Eclipse, which enables one design and implement software easily by using Eclipse along. It uses Eclipse Graphical Editing Framework (GEF) and an optimized plug-in. Omondo EclipseUML is now the best Java UML modeling tool on the market. It provides full supports to combined fragments notion elements in sequence diagram. And the diagrams are saved as a standard XML file format. It is free for non-commercial use.

- **ArgoUML** ArgoUML is the leading open source UML modeling tool and includes support for all standard UML 1.4 diagrams. But ArgoUML can not support combined fragments notion elements in sequence diagrams.

- **No Magic's MagicDraw UML** MagicDraw is a visual UML modeling and CASE tool with teamwork support. Designed for Business Analysts, Software Analysts, Programmers, Quality Assurance Engineers, and Documentation Writers, this dynamic and versatile development tool facilitates analysis and design of Object-oriented systems and databases [39].

- Others.

Our approach uses the sequence diagrams and class diagrams which are drawn in Omondo EclipseUML for the following reasons:

1. Omondo EclipseUML is the best Java UML modeling tool on the market.

2. Omondo EclipseUML saves UML diagrams in the standard XML files which are easy to be parsed using Simple XML API or DOM API.

3. Omondo EclipseUML provides full supports to the combined fragment notation elements in sequence diagrams; it represents combined fragments using frames of types 'opt', 'alt', 'loop', 'par' and 'region'.

4. Omondo EclipseUML is free for non-commercial use. Everyone can use it for non-commercial purpose.

In this chapter, we presented the OMG's UML, which helps one specify, visualize, and document models of software systems. We introduced two standard

diagrams: sequence diagram and class diagram. We also introduced some UML-based tools on the market. Next chapter is a brief introduction to coordination contracts, which is a very important idea in our approach. We realize the test cases in the concept of contracts and make test execution automation with the aid of coordination development environment.

Chapter 5

Coordination Contract

This chapter presents the concept of coordination contract and Coordination Development Environment (CDE). The coordination contract, proposed by L. Andrade and J. Fiadeiro, is related to the idea of the association relationship in UML. The CDE, developed by ATX software SA, supports the use of coordination contracts for Java applications. We also introduce the syntax and semantics of the contracts in the current version of CDE.

5.1 Concepts

This methodology emerges from a particular concern of separation between "computation" and "coordination". "Computation is the mechanisms through which the functionality of services is ensured at the local level of the components of the system. Coordination is the mechanisms through which interactions between components can be established so that the global properties that are required of the system can emerge from the local computations and the interconnections established between them" [4].

One of the main reasons for advocating the separation is that it facilitates the evolution of systems. Changes that do not require different computational properties can be brought about by adding new connectors that regulate the way existing objects operate, instead of performing changes in the objects themselves. This can be achieved by superposing, dynamically, new coordination mechanisms on the objects.

L. Andrade and J. Fiadeiro proposed a concept of coordination contract which provides the mechanisms for the coordination to be modeled and implemented in compositional way.

A coordination contract is a connection that is established between a group of objects or participants. Through the contract, rules and constraints are superposed on the behavior of the participants. A contract is related to the concept of the association relationship in UML, but the way interaction is

established between the participants in contracts is more powerful than what can be achieved within the UML and Object-oriented languages because it relies on a mechanism of superposition that overrides direct, explicit method invocation, and replaces it with an external trigger or reaction kind of interaction [2].

The Coordination Development Environment (CDE) is a pair of tools to help develop Java applications using coordination contracts. We give an example to show the concept of coordination contract in the next section. We describe how contracts can be edited in the CDE to allow the implementation of the micro-architecture in Java in the latter sections.

5.2 An Example of Coordination Contract

Firstly, we introduce the contract by presenting an example of banking application. Secondly we give the formal specification of a contract in the latter section. We have two parts in this example: components and contracts:

- Components. The application banking has two classes: Customer and Account. It allows the withdrawal and deposit of money in a given account, and the choice of whether a given customer can overdraw a given account or not, and by how much. The class Account has two methods: withdraw and deposit, which have no constrains other than amounts must not be negative.

- Contracts. We defined two contracts, Traditional and Credit, each of them between a Customer and an Account to restrict the availability of operation "withdraw" in component Account. Contract Traditional does not allow to withdraw more money than available in the account; contract Credit allows overdrawing an account up to a given amount, which is an attribute of the contract.

The following is the contract Traditional. It has a name: Traditional, and two sections: the participants section lists two participants: customer and account, and the coordination section contains one rule: TraditionalRule. Each participant must be of a class in the components listed above. The rule has a trigger (after when), an optional guard (after with), and an optional body (after do). A trigger is a call to a method of one of the participants and (possibly) additional conditions. The Java code in the body is executed if the trigger occurs and the guard is true. If the trigger occurs and the guard is false, then the code after failure will execute. That code should end either by throwing an exception declared in the signature of the trigger method or by returning a value. So the rule guarantees that a given customer can not overdraw a given account.

```
contract Traditional
participants
  customer : Customer;
  account : Account;
coordination
  TraditionalRule:
    when *−≫ account.withdraw(amount, c) && (customer == c)
    with (account.getBalance() >= amount)
    failure {
      // throw an exception;
    };
    do account.withdraw(amount,c)
end contract
```

The following is the contract Credit, which relaxes the availability condition a bit. The Credit contract allows the participating customer to overdraw the participating account up to a limit, given by a local variable defined in attributes session of the contract.

```
contract Credit
participants
  customer : Customer;
  account : Account;
attributes
  double limit;
coordination
  CreditRule:
    when *−≫ account.withdraw(amount, c) && (customer == c)
    with (account.getBalance() + limit >= amount)
    failure {
      // throw an exception;
    };
    do account.withdraw(amount,c)
end contract
```

Through these two examples, we introduced contracts informally. In the next section, we will give a formal contract specification.

5.3 Contract Specification for CDE1.1

The Coordination Development Environment (CDE), developed by ATX software SA [3], supports the use of coordination contracts for Java applications. The current version is 1.1. This section presents the general format of a contract in the current version of the CDE. The CDE is introduced in detail in the latter section.

5.3.1 Syntax

The general format of a contract in the current version of the CDE consists of a mixture of abstract specifications and Java source code sections. This means that Java statements are parsed and generated as defined. The CDE compiler does not detect Java syntax or semantic errors. The following is a contract template, it includes all possible elements in a contract.

```
contract contractName
participants
  participant_1: Component;
    ...;
  participant_n: Component;
attributes
  JavaType name_1;
  JavaType name_k;
operations
  JavaType contractOperation() {
    // operation body in Java
  }
coordination
  TriggerRuleName:
    when *->> participant_i.operation(args) && (trigger conditions)
    with (JavaGuardConditions)
      failure {
      // Java guard failure actions
      // throw an exception or return a value
      }
    before {
      //operations to be executed before participant_i.operation(args)
    };
    do {
      //operations to be executed instead of participant_i.operation(args)
    };
    after {
```

//operations to be executed after $participant_i$*.operation(args)*
};
StateConditionRuleName:
when ? (condition in Java) **on** $participant_1$, ..., $participant_n$
do {
// set of operations of the participants or the contract
};
end contract [4]

5.3.2 Semantics

The semantics of the various constructs in a contract is as follows [4]:

- **contract** contractName: This is the name of the contract as specified by the user when creating a new contract.
- **participants**: This is a sequence of formal parameter declarations that identify the classes that become associated through the contract. This sequence can consist of one ore more Java classes that have been added to the current CDE project. The classes should have been added to the project prior to being declared as contract participants. Otherwise, a compilation error will occur. There are two ways to define participants: either by using the syntax *participantName:Class*; or by using *Class participantName*
- **attributes**: These are the private attributes of the contract. The attributes are generated as written by the user in this section and therefore they have to be specified using Java syntax; The contract file generated by the CDE defines, by default, two public methods for each attribute: one to set values and the other to get the current value.
- **operations**: These are private operations of the contract. They should be edited in Java and are generated as specified. Therefore they should be in correct Java syntax and have correct semantics in order to avoid compilation or behavior errors when integrated in the generated code for the rest of the application. The Java syntax error in the operation body will not cause a compilation error in the CDE, but the generated Java code may give rise to a compilation error. Therefore, it pays off to be careful to use correct Java syntax and semantics in the definition of these operations.
- **coordination**: This section defines the coordination rules that will be superposed on the participants. There are two types of coordination rules

that are currently supported by the CDE: rules on calls to operations of the participants and state condition rules.

- Trigger Rule. The syntax of this coordination rule for an operation invocation is a statement of the following form:

$$when\ *-\gg\ participant.operation(args)$$

  * The symbol $*-\gg$ applied to an operation means "any call to that operation". The operation arguments should be consistent with the signature of the operation.
  * The && section defines the trigger conditions of the rule, i.e under which conditions the call to the operation constitutes a trigger. In other words, if the operation is called when this condition is false, the contract will not react.
  * The **with** section defines the guard condition for the trigger: when this condition is false, an exception is raised and a failure is reported to the object that called the operation. This condition should also be written in Java.
  * The **failure** statement for the with guard specifies the Java actions that should be executed in case the guard fails. Because failure is not a Java feature, it is necessary to model the rule failure in case the with condition is false, by either throwing an exception or returning a value to the operation client. Hence, the last statements inside the failure block should be either a throw Exception or a return statement.
  * When the trigger corresponds to the call for an operation, three types of actions may be superposed on the execution of the operation: **before** (to be performed before the operation), **do** (to be performed instead of the operation), and **after** (to be performed after the operation). In the case in which an object participates in multiple contracts with the same trigger, the sequence of execution is the following: first, all the before actions are performed, then one replace, and finally all the after actions.
- State Condition Rule. The syntax of this state condition rule is a statement of the following form:

$$when\ ?\ (condition\ in\ Java)\ on\ participant_1, ..., participant_n$$

  State condition rules are declared by using **?** with the condition statement specified inside parenthesis and in Java syntax. The

statement, *on* $participant_1, participant_2, \ldots$, is used to declare which participants the rule refers to. Only statements that do not change the state of the participants may be defined as conditions on state rules. For instance, a state condition rule can be of the form:
when *? participant*$_1$*.getBalance()>100* ***on*** *participant*$_1$

- **end contract**: This defines the end of the contract.

5.4 Micro-Architecture

Manual implementation of coordination contracts, or tools such as the CDE, require an underlying micro-architecture (design pattern). The micro-architecture presented herein is only one among several possible alternatives for implementing coordination contracts. There may well be a different architecture or different implementation of contracts as long as they adhere to the general principles of contracts [4].

The coordination contract Design Pattern consists of two parts: The component part and the coordination part. The former consists of the features that have to be provided for each component so that it can become coordinated by a contract. The latter concerns the mechanisms that allow for the coordination of a given component through the contracts that are in place for it. The classes that participate in the proposed pattern are shown in Figure 5.1.

The detailed functionality of the various classes is as follows [4]:

Component Part

- **SubjectInterface**. It is an abstract class (type) that defines the operations under potential coordination. In fact, it is the common interface of services provided by *SubjectToProxyAdapter* and *ISubjectProxy.*

- **Subject**. This is the real component, candidate for coordination, that provides the concrete implementation of the various services and inherits from *SubjectToProxyAdapter.*

- **SubjectToProxyAdapter**. Defines the ability to, alternatively, use a proxy or internal methods for the implementation of a given Subject interface. It is a concrete class that allows, at run time, and using the polymorphic entity proxy, for delegating received requests to *ISubjectProxy* in the case in which Subject is under coordination. Such requests are then delegated to *ISubjectPartner* that links the subject to the contracts that coordinate it. If no contract is involved, *SubjectToProxyAdapter* may forward requests directly to Subject. In order to achieve

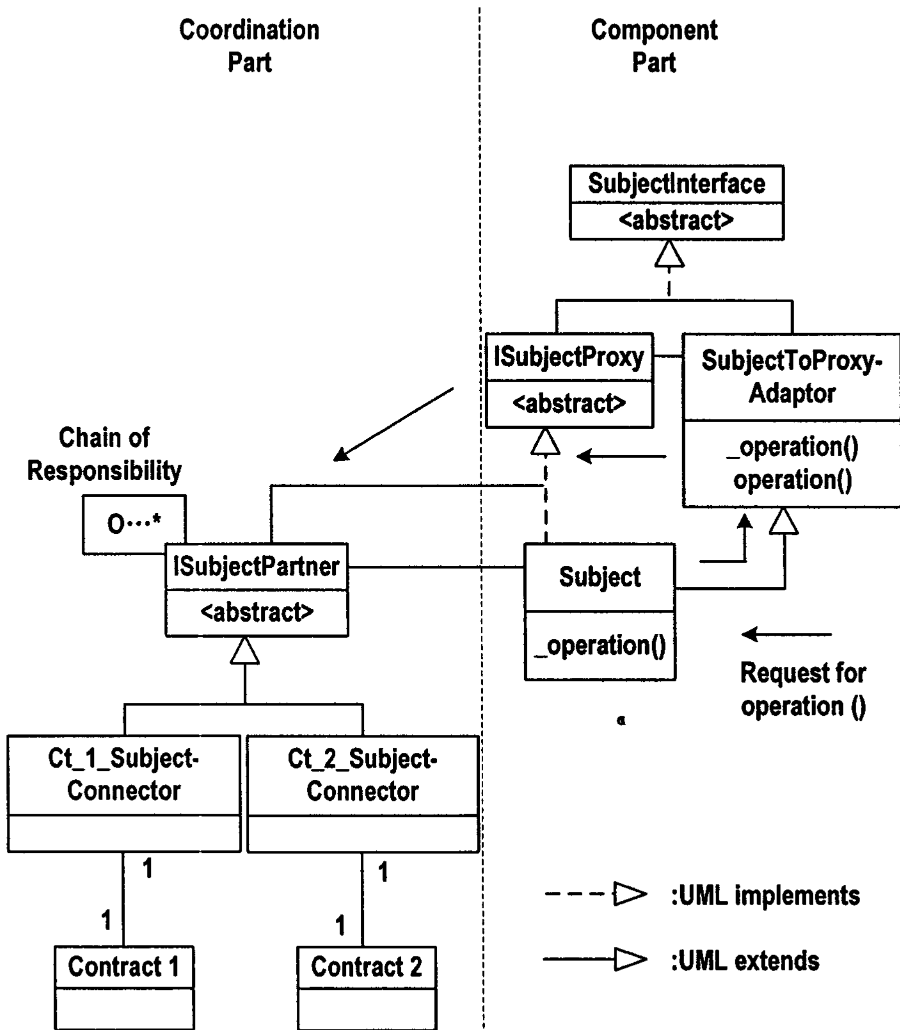


Figure 5.1: Coordination Contract Design Pattern [4]

this, two actions are necessary. Firstly, Subject inherits from *SubjectToProxyAdapter*. Secondly, the operations of Subject are renamed in such a way that the operations with the initial names are moved to *SubjectToProxyAdapter* as concrete operations, and the new operations occurring in *SubjectToProxyAdapter* are abstract operations. For instance, operation( ) of Subject exists now as operation() in *SubjectToProxyAdapter* and as _operation( ) in Subject. Moreover, _operation() is also declared as an abstract operation in *SubjectToProxyAdapter*. Requests for operation() are made to Subject. However, due to renaming, the operation does not, in fact, exist in Subject but in *SubjectToProxyAdapter* from which Subject inherits. In the case that there is no contract (no proxy) involved, operation() in *SubjectToProxyAdapter* forwards the request to the corresponding real implementation, _operation(), in Subject. Otherwise, as already stated above, it delegates the request to *ISubjectProxy*.

- **ISubjectProxy**. It represents an object with the capability of implementing the Subject interface. It is an abstract class that defines the common interface of Subject and *ISubjectPartner*. The interface is inherited from *SubjectInterface* to guarantee that all these classes offer the same interface as Subject with which real subject clients have to interact.

Coordination Part

- **ISubjectPartner**. Defines the general abilities of a concept to be under coordination. It maintains the connection between the real object (Subject) and the contracts in place for it. The class is responsible for delegating received requests to *CtSubjectConnectors* according to a chain of responsibility. The class contains operations for managing the chain of responsibility. Alternatively, the required management operations can be included in an abstract class, *ContractPartner*, from which *ISubjectPartner* inherits. However, this is rather a "low-level" design issue and therefore such a class is not presented in the pattern.

- **Ct_i_SubjectConnector**. A partner that represents the specificities of Subject coordination for a given contract in which Subject is a participant. For each pair contract-participant there is exactly one *Ct_i_SubjectConnector*. The class implementation may be responsible for the execution of the rules defined in the coordination part of the contract and for ensuring satisfaction of the contract semantics.

- **Contract-i**. A coordination object that defines the rules that will be superimposed on Subject.

The contracts micro-architecture allows coordination contracts to be directly implemented using object-oriented languages, while providing the following advantages [4]:

1. Components are not aware of the presence of contracts and, therefore, any number of contracts can be added/removed without having to modify the components.
2. Contracts can be added/deleted in a "plug and play" mode, even in run time.
3. Even existing components can be easily adapted to accept contracts without making any modifications elsewhere in the application, thus allowing for easier reengineering/evolving of existing applications.

5.5 CDE

The Coordination Development Environment (CDE) is a pair of tools to help develop Java applications using coordination contracts. One of the tools is an editor to write contracts and translate them into Java classes, which can then be compiled with the other classes that form the application. The other tool is an animator, with some reconfiguration capabilities, in which the run-time behavior of contracts and their participants can be observed using sequence diagrams, thus allowing rapid testing of different scenarios. The Java classes generated for coordination contracts provide public methods that allow the applications to dynamically reconfigure themselves by creating and deleting contracts, and changing the values of their attributes. The CDE is written in Java and can be downloaded for free from *www.atxsoftware.com/CDE* [4].

The CDE generated Java code provides a specific implementation of the general contracts micro-architecture. In this context, the implementation points we wish to present are the following:

- In the general micro-architecture *Subject* and *SubjectToProxyAdapter* appear as two different classes. However, for reasons explained in the micro-architecture document, the previous two classes are merged in one class, Subject.
- In the general micro-architecture class *ISubjectProxy* is supposed to implement *SubjectInterface* and also *Subject* and *ISubjectPartner* are subclasses of (implement) *ISubjectProxy*. However, in the CDE generated code instead of *ISubjectProxy*, a general, not specific to the Subject, class *CrdIProxy* is used. Each Subject either has *CrdIProxy* or inherits it from its parent class. The presence or no presence of contracts

is determined by controlling *CrdIProxy_proxy*. If *_proxy == null* there are no contracts involved, otherwise contracts exists and calls to Subject are delegated to contracts [4].

- The chain of responsibility management operations are provided by a framework class *CrdContractPartner* from which *ISubjectPartner* inherits. For each call on a participant that is forwarded to the chain, all active partners are determined, the first one takes responsibility and executes its actions, then it forwards to the next one and so on until no active partners are left. The current implementation does not allow the setting of priorities between contracts in the chain. Therefore, priorities of contracts is currently a matter of configuration i.e the order contracts are established between participants is also the execution order in the chain of responsibility. In the future versions of CDE, however, the ability to set the contracts priorities in the chain management will be provided [4].
- Contract's state conditions rules are evaluated whenever a call to an operation that changes the state of a participant occurs. This is accomplished by invoking an operation named *stateChange*() that is defined on each contract.

Unfortunately, the current version of the CDE can not coordinate classes of objects for which the source code is not available. One can only coordinate components for which the source code is available. Therefore, for instance, one may not define contracts that superpose behavior on operations that belong to a Java class library. ATX Software SA said they will implement this important feature that the CDE will support the coordination of components for which the source code is not available such as Java .class files soon.

5.6 Other Notions of Contract

In addition to the coordination contract introduced above, there are several other notions of contract. The term "contract" is somewhat overloaded. There are contracts in the sense of Meyer [35] in "Applying Design by Contract"; R.Helm et al [26] proposed a notion of "Contract" in "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems". A notion of contract can also be found in "Analysing UML Use Cases as Contracts" [5]. We introduce the different notions of contract as follows.

- **Applying Design by Contract**. They fulfil a different, but complementary, role to coordination contracts: their purpose is to support the

development of object methods in the context of client-supplier relationships between objects. Therefore, they apply, essentially, to the "in-the-small" construction of systems rather than the "in-the-large" dimension that cooperation contracts have chosen as target and which is concerned with architecture and evolution.

- **Contracts: Specifying Behavioral Compositions in Object-Oriented Systems.** A notion of contract that applies to behaviors and not to individual operations or methods is the one developed in "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems". The aim of contracts as developed therein is to model collaboration and behavioral relationships between objects that are jointly required to accomplish some task. The emphasis, however, is in software construction, not so much in evolution.

- **Analysing UML Use Cases as Contracts.** RJ.Back et al presented another notion of contract in "Analysing UML Use Cases as Contracts", that emerged in the context of the action-systems approach. Like cooperation contracts, it promotes the separation between the specification of what actors can do in a system and how they need to be coordinated so that the computations required of the system are indeed part of the global behavior. The architectural and evolutionary dimensions are not explored as such.

In this chapter, we introduced the coordination contract and the CDE. The coordination contract is a mechanism which superpose behaviors to the components without interfering with their implementations. The CDE supports the use of coordination contracts for Java applications. The coordination contract and the CDE play an important role in our approach because we design tests in the concept of the contracts and execute tests automatically with the aid of the CDE. In the next chapter, we will present our test approaches in detail.

Chapter 6

Test Approaches

In this chapter, we introduce our test approaches, object-oriented programs integration tests by testing the sequence of the message calls, testing parameters and testing post-conditions. We describe the test case generation from UML sequence diagram, test oracle from class diagram, and test coverage criteria we use in our approach.

6.1 Introduction

The purpose of our test approach is to detect faults related with the interactions among objects in a system. Test cases are derived from UML sequence diagrams and class diagrams. Sequence diagrams are used primarily to show the interactions among objects in the sequential order that those interactions occur. Thus sequence diagrams are one of the most suitable specifications to guide our integration testing. Class diagrams consist of classes and the relationships among them. By using a class diagram specification, we can get enough information for verifying the test result, like an oracle. Given sequence diagram and class diagram specifications, our integration test process can be generally described as follows:

1. Generate Test Case from sequence diagram and class diagram. The test case generator tool parses the sequence diagram and class diagram XML files and gets the useful information for the integration testing, see details in next section "Test Case Generation".

2. Realize Test Case in terms of Contracts. What to test, how to test and how to verify result in terms of contracts can be generated by the tool, see details in Chapter 7.

3. Create a Test Framework by Generating Contracts and Components in CDE. We import the components under test and the contracts generated

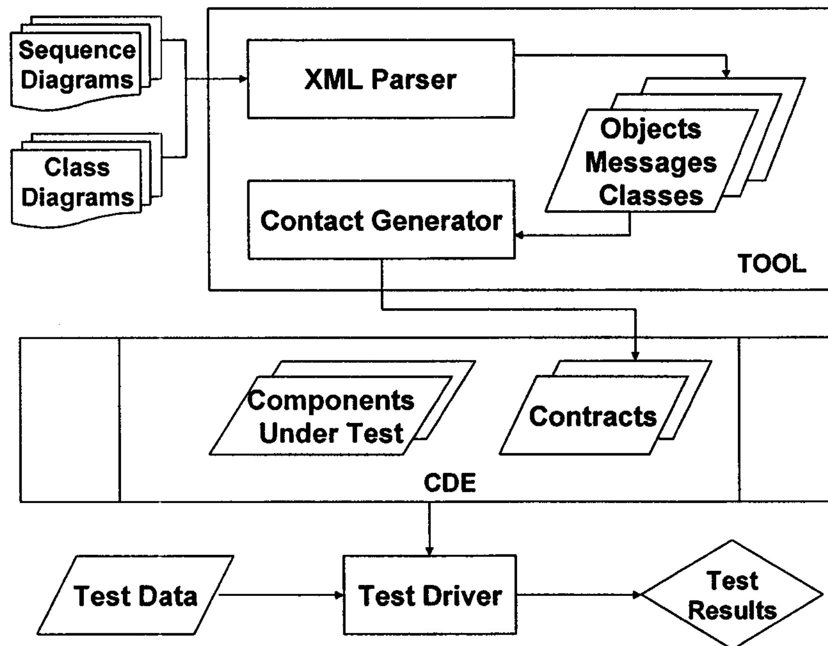


Figure 6.1: Architecture of Integration Testing Approach.

by the tool into the CDE. To generate a contract and component is to produce the Java code that implements the micro-architecture that we introduced in Chapter 5 for allowing coordination contracts to be superposed on components without the latter being aware of the contracts existence.

4. Generate Test Data by the Tool JTA [17], provided by professor Marcelo Frias. The JTA Tool takes a sequence diagram, selects the component code corresponding to the sequence diagram and generates test data for this part code by Branch Coverage Criteria, see details in the latter section in this chapter.

5. Develop Test Driver. A test driver is developed to run the test framework by taking the generated test data.

6. Generate the test result. Test results are generated after running the test framework with the generated test data.

The whole process is represented in Figure 6.1. The rectangle part at the right top corner in the figure is the core of the Tool we have developed in

Java. It takes UML sequence diagram and class diagram and generates the contracts by the mechanism of test case generation introducing in the next section. Please refer to Chapter 8 for the implementation details of the Tool.

6.2 Test Case Generation

How should we do the integration testing for object-oriented programs based on UML specifications? As described above, there are some techniques or approaches to test object-oriented programs at the integration level based on UML specifications. We developed an alternative approach by using UML sequence diagrams and class diagrams. Through a sequence diagram, we know the interactions among objects in sequential order represented as sending some message to some object in sequential order, and the parameters taken from the first message and when and by which messages are taken again in the later. Sequence diagrams are primarily used to generate test cases; through a corresponding class diagram, we know each parameter's type, each class's attributes and the relationships among those classes. Class diagrams are primarily used to create test oracles. Therefore in order to test object-oriented programs at the integration level, we test the interactions among objects in three parts: sequence of method calls, parameters and post-conditions. Our approach will reveal those faults related with incorrect sequence of method calls, inconsistent parameters and unexpected behavioral operations. We present each part of testing as follows.

6.2.1 Testing Sequence of Message Calls

Given a sequence diagram, we want to make sure all the methods in the sequence diagram are called in correct sequential order with respect to the values of the conditions in the control flows. How should we do this? The general idea of the approach is that we introduce an integer variable **step** and each method in the sequence diagram is assigned a unique step value. The value of step at one time is set to the value related with the message which is the latest executed. We assert the correctness of the sequence by checking the current value of step before a certain message is called is what we expect by the sequence diagram.

If the method which is going to be called is the first method in the sequence diagram, we do not check the step value, we only assign a value related with that method to **step**. The sequential order of the messages presented in the sequence diagram depends on the control-flows: different test data indicate different control-flow for the test data are generated by Branch Coverage Criteria, as a result methods are called in a different order. By checking the value of **step** when each message is called, we know if the current message is

called in the expected sequential order or not. We summarize our approach as follows:

Approach to Testing Sequence of Method Calls (TSMC)

1. Create an integer variable called **step**. Initialize **step** $step = 0$.
2. Assign each method a unique sequential value, starting from 1, saved in **step**.
3. Assure that each method is called in the right sequential order by checking the current value of **step** is the value corresponding to the method which is expected to execute before the currently executed method.
4. Reset **step** to the value corresponding to the currently executed method.
5. At the end, check the value of **step**; if it is the value we expected (the step value of the last method executed by the sequence diagram), the methods are called in the right sequential order. Otherwise, the test fails.

LEMMA: TSMC succeeds if expected and actual sequences of method calls are equal, and fails the test if they are not.

PROOF: Proof by Induction.

- Base Step: Suppose there are only two messages: $message_1$, $message_2$ and $message_1$ is followed by $message_2$. By applying approach TSMC, initially, $step = 0$, after $message_1$ is invoked, $step = 1$; when $message_2$ is invoked, and now $step = 1$, reset $step = 2$. The expected value of $step$ is 2; therefore, the test passes if $message_1$ is followed by $message_2$ in the actual message invocations; the test fails if it is not.
- Induction Step: Suppose TSMC asserts that expected and actual sequences of message calls are equal when there are n messages invoked. That is, the test passes when $step = n$ and the test fails when $step \neq n$ at the last. If there are $n+1$ messages, the expected result is $message_{n+1}$ following $message_n$. When the $n+1st$ message is invoked, there are two cases: 1. $step = n$, which shows the first n messages are called in the right sequence, reset $step = n + 1$, test passes; 2. $step \neq n$, which shows the first n message are not called in the right sequence, $step$ will not be reset to $n + 1$, test fails.

In conclusion, the procedure TSMC guarantees correctness of sequences of message calls.

6.2.2 Testing Parameters

Given a sequence diagram and a corresponding class diagram, make sure the type signatures are consistent within the diagrams. We only check the parameters introduced in the first message in the sequence diagram. For any parameter in the first message, if it is used in the later messages, we check both the value and the type of the parameter. For each parameter under test in a sequence diagram, we get the name from the sequence diagram and we can find the type from the relevant class diagram. A class diagram is the specification from which we create a test oracle for testing parameters.

Approach to Test Type Signatures (TTS)

1. Read the first message in the sequence diagram, save the parameters appearing in the message one by one.

2. Scan the relevant class diagram, find the message appearing the first in the sequence diagram, and get the type of the each parameter in the message.

3. Navigate each message except the first one in the sequence diagram sequentially. If any parameter in the first message appears in the following messages, compare the value and type of the parameter between the current message and the first one.

4. If the compared parameters have the same value and type, the test passes; otherwise, the test fails.

In testing parameters, we just check all the parameters taken by the first message are consistent in a sequence diagram. We do not test those parameters which are not appearing in the first message.

6.2.3 Testing Return Value

Suppose we have a sequence diagram, depicting the interactions among a group of classes. We have an assumption that class-level testing has been done on each class sufficiently. The approach Testing Object Interactions (TOI) described below is a general approach to test objects interactions using simulation technique to check the post-conditions of each participant (object) after the interactions. The purpose of the approach TOI is to detect the faults related with the interactions among objects.

The basic idea of the approach is to simulate the execution of the programs under test. We start from the same states with the program by making a copy of each object; we simulate the execution of the interactions among

the objects by manipulating the copies of the objects in a such a way that the messages, according to the sequence diagram, are sent to the corresponding copies of the objects sequentially; on the other side, the program is running as it developed. Finally, we compare each pair of the object and its copy. If they are equivalent observationally, then the behaviors of each pair are equivalent; we say the interactions take place correctly; otherwise, test fails. We describe each step in detail as follows.

Approach to Test Object Interactions (TOI) Suppose we have a sequence diagram, depicting the interactions among a group of classes: $class_1$, $class_2$,..., $class_n$. Let $object_1$, $object_2$,..., $object_n$ denote the objects of each class, respectively. Let's assume class-level testing has been done on each class. Given test data generated by the Tool JAT[17] (the test data cover each branch of the control-flow in the sequence diagram), the following operations will be executed sequentially by the sequence diagram specification:

$$\begin{aligned} &object_1.operation_1(parameter_1);\\ &object_2.operation_2(parameter_2);\\ &\vdots\\ &object_n.operation_n(parameter_n). \end{aligned}$$

1. Clone Objects. Before the first operation is invoked on the $object_1$ in the sequence diagram, clone each of $object_1$, $object_2$,..., $object_n$ and rename new objects as pre_object_1, pre_object_2,..., pre_object_n using the Approach to Clone an Object (**CAO**), see below.

2. Execute Operations on the Cloned Objects as We Expect. Run

   $$pre_object_i.operation_i(parameter_i), i = 1, 2, ..., n$$

   sequentially.

3. Execute Operations on the Objects in Actual Program. Continue to invoke the first operation on the $object_1$, it will trigger all the message invocations in the sequence diagram.

4. Use the Approach to Determine Objects Behavioral Equivalent (**DOBE**) to examine whether the program is executed as we expected by determining if $object_i$ is equivalent to $pre_object_i, i = 1, 2, ..., n$.

The TOI is a general approach to test objects interactions by checking the post-conditions of each participant after the interaction. It consists of the other two approaches: CAO and DOBE, see below.

Approach to Clone an Object (CAO) Suppose the attributes of class **C** are a_1, a_2, ..., a_n and a_i is public for $i = 1, 2, ..., n$. Furthermore, that $object.a_i$ denotes the value of a_i of *object* for $i = 1, 2, ..., n$. We have an object instance of class C: "object". We will make a copy of "object" by the following two operations.

1. Create a new object instance named "pre_object" with the same type of "object" using the default construction function, as follows:

$$pre_object = new\ ClassOfObject();$$

2. Initialize the value of each attribute in "pre_object" using the assignment below:

$$pre_object.a_i = object.a_i\ for\ i = 1, 2, ..., n$$

We have a copy of the object instance "object" of class C, named "pre_object", because both "object" and "pre_object" are object instances of class C and they have the same value for each attribute, representing by the equal values in the following two tuples:

$$(object.a_1, object.a_2, ..., object.a_n)$$

$$(pre_object.a_1, pre_object.a_2, ..., pre_object.a_n)$$

For example, in a banking system, we have a class Account with two attributes: *number* and *balance*, both are the type integer. The following is an example of Java code:

```
public Class Account {
  int number;
  int balance;
  //operations;
}
```

Now we have an object instance "account" with the value of *number* 1 and the value of *balance* 200. We will use the approach CAO to make a copy of "account" using the following Java statements:

1. pre_account = new Account();

2. pre_account.number = account.number;

3. pre_account.balance = account.balance;

Thus, object instance "pre_account" is a copy of object instance "account". Both are the object instances of class Account and have the same value for each attribute.

Approach to Determine whether Objects are Behaviorally Equivalent (DOBE) [15] Suppose the attributes of the implemented class C are a_1, a_2, ..., a_n and a_i is public for $i = 1, 2, ..., n$. Suppose, further, that $object_k.a_i$ denotes the value of a_i of $object_k$ for $i = 1, 2, ..., n$ and $k = 1, 2$. Check whether the following two tuples are equal:

$$(object_1.a_1, object_1.a_2, ..., object_1.a_n)$$

$$(object_2.a_1, object_2.a_2, ..., object_2.a_n)$$

If yes, we have

$$object_1 \approx object_2$$

How should we compare the actual result by running the program under test with the expected result derived from the given specification? The comparison is based on the objects involved in the interactions. We determine if each object is behaviorally equivalent with our expected object by the above approach **DOBE**.

The approaches TOI, CAO and DOBE are used together to test the interactions among objects. The approach TOI uses the idea of the simulation technique to simulate the execution of the program under test; the approach CAO makes a copy of an object in the program; the approach DOBE compares the equivalence of two objects behaviorally.

6.3 Test Case Coverage

One important aspect of software testing is deciding when enough testing has been done. Goodenough and Gerhart [20] first presented the idea of a test adequacy criterion, which is a criterion that defines what makes an adequate test. Adequacy criteria play an important role in the testing process. They can be used as a stopping rule. Testing stops when enough test cases have been produced to satisfy the criterion. They can also be used as a measurement of test quality. They also provide a basis for deciding what test cases to use during testing, making it more likely that faults will be found in the system [33].

6.3.1 Test Coverage for Integration Testing

Coverage can be used to measure the extent to which an adequacy criterion is satisfied. Test coverage is usually given in terms of percentage of the chosen structures covered at lease once. Coverage criteria are a type of adequacy criteria that specify the percentage of requirements that must be covered. We introduce several coverage criteria based on data-flow, control-flow and UML

below. A test strategy can be based on coverage of one or more of the following [33].

Data Flow Coverage Criteria

Linnenkugel and Müllerburg (1990)[31]defined the following criteria for data flow based integration testing. All the criteria are defined for communication variables. A communication variable (CVar) contains data which is shared between a calling and a called procedure (imported operation) via parameter parsing or a global variable.

- INT-all-defs criterion. Every definition of a CVar within the calling module which may affect the call has to be read at least once within the imported operation.
- INT-all-c-uses/some-p-uses criterion. Every definition of a CVar within the calling module which may affect the call has to be used for every possible computation within the imported operation. Predicate uses have to be tested only if computations do not exist within the operation.
- INT-all-p-uses/some-c-uses criterion. The same as above except that predicative uses have priority.
- INT-all-uses criterion. Every definition of a CVar within the calling module which may affect the call has to be tested for every possible use (c-use and p-use) within the imported operation.
- INT-all-du-paths criterion. This is the enlargement of the INT-all-uses criterion. Different paths (e.g. decision branches) between definition and use of a CVar (without loops) are taken into consideration.

A disadvantage of all these criteria is that they are either met or not, there is no level in between. However, coverage measures are easily defined. Ratios may be defined for each of the criteria. The advantage is that complete coverage of a specific criterion is not necessary any more, different levels of coverage may be defined. For example, the all-defs integration test coverage measure is defined in the following:

$$IC_{ad} = \frac{number\ of\ executed\ paths\ containing\ a\ CVar\ definition}{total\ number\ of\ paths\ containing\ a\ CVar\ definition}$$

Control Flow Coverage Criteria

Control-flow criteria are traditionally considered as program-based and useful for white-box testing [51]. Several criteria and respective coverage measures

have been introduced by Miller(1977) and extended later by other authors. The following coverage criteria are based on the control-flow graphs: statement coverage criterion, branch coverage criterion, condition coverage criterion and path coverage criterion.

Integration testing requires testing interactions between modules and operations. Criteria to test modules which export several operations are defined by Herrmann and Spillner in 1992 [27]:

- INT-all-exports criterion. Every exported operation has to be executed at least once.
- INT-all-imports criterion. Every exported operation has to be called at least once from every module importing this operation.
- INT-all-multiple-imports criterion. Every call statement of an imported operation has to be executed at least once.
- INT-all-import-call-sequences criterion. For every module only a single import of an operation is taken into consideration. Every sequence of calls has to be executed at least once in accordance to the sentence above.
- INT-all-multiple-import-call-sequences criterion. Every call within a system is taken into consideration. Every possible sequence of calls is executed at least once.

The definition of coverage measures is:

$$IC_{ae} = \frac{number\ of executed\ exported\ operations}{number\ of\ all\ exported\ operations}$$

$$IC_{ai} = \frac{number\ of\ executed\ imported\ operations}{number\ of\ all\ imported\ operations}$$

$$IC_{ami} = \frac{number\ of executed\ calls\ of\ imported\ operations}{number\ of\ all\ calls\ of\ imported\ operations}$$

UML-based Coverage Criteria

The UML is a language for specifying, visualizing, constructing and documenting the artifacts of software systems. The UML provides a variety of diagrams that can be used to present different views of an object-oriented system at different stages of the development life cycle [22]. Testing techniques that are based on the UML involve the derivation of the test requirements and coverage criteria from these UML diagrams. McQuillan and Power [33] wrote a survey paper, presenting these techniques with emphasis on the coverage criteria. They introduced and analyzed various criteria based on different UML diagrams.

Author	**Criterion**
Basanieri and Bertoline	All-Paths-Coverage
Binder	Condition/Iteration Coverage
Briand and Labiche	All-Paths-Coverage
Fraikin and Leonhardt	All-Paths-Coverage
Rountev et al.	All-IRCFG-Paths
Rountev et al.	All-RCFG-Paths
Rountev et al.	All-RCFG-Branches
Rountev et al.	All-Unique-Branches

Table 6.1: Coverage criteria based on Sequence Diagrams [33]

A start-end message path in a sequence diagram is a sequence of messages that begins with an externally generated event and ends with the production of a response that satisfies this event. A test requirement based on UML sequence diagrams is that all the start-end message paths are covered by the test execution. This can be referred to as the "all-paths coverage criterion" which can be defined as:

Definition: All-Paths Coverage Criterion A set of message paths P satisfies the all-paths coverage criterion if and only if P contains all start-to-end message paths in a sequence diagram [33].

Binder [9] presents a testing technique that considers a subset of all start-to-end message paths in a sequence diagram. The technique involves converting the sequence diagram to a control flow graph (CFG) and deriving test cases from this graph. The coverage criterion he uses provides branch/iteration coverage and is an extension of the traditional branch coverage criterion which can be defined as:

Definition: Branch Coverage Criterion A set of paths P satisfies the branch coverage criterion if and only if for all edges e in the control flow graph, there is at least one path $p \in P$ such that p contains the edge e [33].

Definition: Iteration Coverage Criterion Given a test set T and sequence diagram SD, for each loop $L \in SD$, T must cause the loop to be either bypassed or taken for the minimum number of iterations, to be taken at lease once and to be taken for the maximum number of iterations [33].

Rountev et al [42] introduced the Inter-procedural Restricted Control-Flow Graph (IRCFG), which depicts the set of message sequences in a sequence

diagram. An IRCFG contains a set of restricted CFGs (RCFGs), together with edges which connect these RCFGs. Each RCFG corresponds to a particular method and shows the sequence of messages that are involved in response to the method call. This IRCFG is used to define a set of coverage criteria for sequence diagrams, as follows:

Definition: All-RCFG-Paths Criterion A set of IRCFG paths P satisfies the all-RCFG-paths coverage criterion if and only if P contains all RCFG paths [33].

Definition: All-RCFG-Branches Criterion A set of IRCFG paths P satisfies the all-RCFG-branches coverage criterion if and only if for all edges e in each RCFG, there is at least one path $p \in P$ such that p contains the edge e [33].

6.3.2 Test Coverage Criteria in Our Approach

Our approach designs an integration testing automation tool for test case execution and test result evaluation. Test data are generated by the tool JAT [17] based on the branch coverage criterion. The Branch Coverage criterion requires a set of paths P if and only if for all edges e in the control flow graph, there is at least one path $p \in P$ such that p contains the edge e [33].

In sequence diagrams, the combined fragment notation elements are used to group sets of messages together to show conditional flows. We can represent different control flows by using different types of combined fragment notation elements. The types of combined fragments include OPT, ALT and LOOP which represent optional (like *if* statement), alternative (like *if-else* statement) and loop (like *while* statement) control flow, respectively. By applying the Branch Coverage criterion to generate test data, each branch is guaranteed to be traversed at least once by running the test data. Thus we cover every possible branch in any control flow in the sequence diagrams. We will introduce the derivation of a control flow graph from a sequence diagram in details in Chapter 7.2.

In this chapter, we gave a very detailed introduction to our test approaches for object-oriented programs at the integration level. The approaches include the TSMC testing the sequence of the methods calls, the TTS testing parameters, and the TOI testing return values along with the CAO and the DOBE. We also presented integration testing coverage criteria based on data-flow, control-flow and UML. In the next chapter, we will introduce how to realize the test cases in the concept of coordination contract and how to execute tests automatically by generating the contracts and the components in CDE.

Chapter 7

Test Design by Contracts

In this chapter, we present the detailed realization of test cases in the concept of coordination contract based on the mechanism of test case generation introduced in Chapter 6. We also give an introduction to the use of the CDE, generating the contracts and the components to fulfil tests execution automation.

7.1 Introduction

As we introduced in the chapter on coordination contract, which defines rules that coordinate the behavior of given objects in a system, and allowing for this rules to be added, or modified, without having to modify the way those objects are implemented. This feature of the contract matches our approach for integration testing.

The integration testing for object-oriented programs is to assure the interactions among the objects are correct with respect to the specifications. The test cases for the integration testing, based on the mechanism of test case generation introduced in chapter 6, manage the interactions among the objects, including what interactions should happen, how they should interact and do they interact as we expect, thus making contracts suitable to implement the test cases.

By using contracts to superpose behaviors on the components, we can detect, modify, and analyze the interactions among the objects for the purpose of the integration testing. Most importantly, we can add and modify the test cases without having to modify the way those objects are implemented. In other words, all the test cases implemented in the concept of the contracts do not interfere with the original program code. We introduce the detailed implementation using the contracts in the following section.

7.2 Contract Design

Used in integration testing, the contracts are based on sequence diagrams. As we introduce in chapter 6, for each sequence diagram, we test three things: messages/calls sequences, parameters and return values. Correspondingly, we have one contract for each test item, totally three contracts for one sequence diagram. Contract one is implemented for testing message calls invoked in the sequence specified by the sequence diagram; contract two is implemented for testing parameters appearing in the first message that are used in the later messages; contract three is implemented for testing the return values by checking the interacted objects behavior as what we expect.

7.2.1 Contract for Testing Sequences of Message Calls

The contract for testing sequences of message calls is designed to ensure the sequence of the method calls is consistent with the sequence specified in the sequence diagram. The contract is implemented based on the approach TSMC introduced in chapter 6. The following is the approach TSMS and corresponding contract implementation.

Approach to Testing Sequence of Method Calls (TSMC)

1. Create an integer variable called **step**. Initialize **step** $step = 0$.
2. Assign each method a unique sequential value, starting from 1, saved in **step**.
3. Assure that each method is called in the right sequential order by checking the current value of **step** is the value corresponding to the method which is expected to execute before the currently executed method.
4. Reset **step** to the value corresponding to the currently executed method.
5. At the end, check the value of **step**; if it is the value we expected (the step value of the last method executed by the sequence diagram), the methods are called in the right sequential order. Otherwise, the test fails.

With respect to each step in the above approach, we implement the contract accordingly as follows.

- In the contract attributes section, define **step** with respect to step 1.

- We define some trigger rules and one state condition rule in the contract. Each trigger rule checks if a certain message is called in the right order. The number of the trigger rules in the contract equals the number of methods called in the sequence diagram. The name of a trigger rule is "CheckStepi"; *i* is the sequence of the related message in the sequence diagram. The state condition rule is used for checking the final result.

- With respect to step 4, in the before section of a certain trigger rule, we reset step to the value related with the method in this rule.

- With respect to step 2, in the trigger rule related with the first message, only the call to the message constitutes the trigger. In other words, when the first message is called, just reset step value in before section as mentioned above.

- With respect to step 3, in the trigger rules other than related with the first message, the call to the message and the condition, which the current value of step is the value of related message which was just called, constitute the trigger. In other words, if the method is called when this condition is false, the contract will not react; as a result, step will not be reset.

- With respect to the final result checking, check the step value. It could be implemented either in the after section of the last trigger rule or in a state condition rule. In most cases, we use trigger rule; only in one special case of conditional message, we have to use a state condition rule to check the value of step. We will introduce this special case in detail below.

Let's look at a very simple sequence diagram, shown in figure 8.4. In this sequence diagram, object1 and object2 are two object instances of Class1 and Class2, respectively. Message_1() is called on object1, followed by the message_2 called on object2. Applying the approach TSMC, we implement the test case in the following contract.

Contract for Testing Sequence of Message Calls in Figure 8.4.

contract example
  participants
    object1:Class1;
    object2:Class2;
  attributes
    boolean result = false;
    int step = 0;

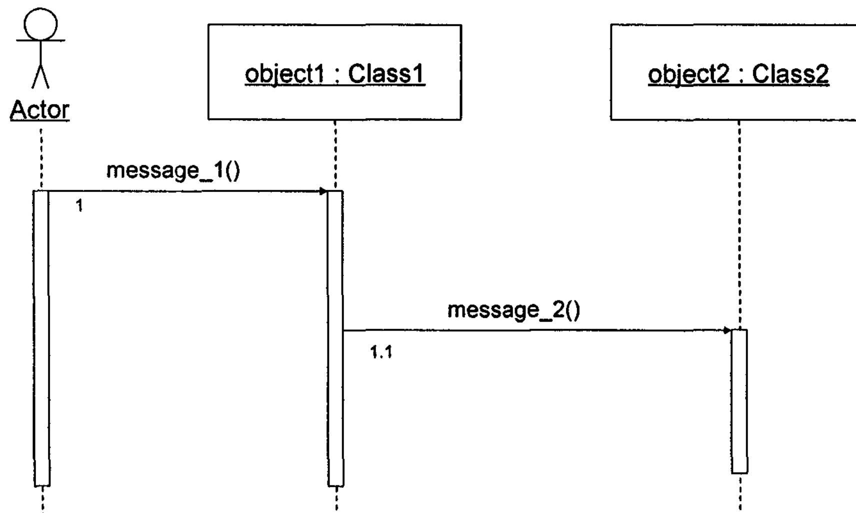


Figure 7.1: A Simple Sequence Diagram Example

```
coordination
  CheckStep1:
    when *->> object1.message_1()
    before {
      step = 1;
    };
  CheckStep2:
    when *->> object2.message_2() && (step == 1)
    before {
      step = 2;
    };
    after {
      if(step == 2) {
        result = true;
        System.out.println("Sequence Test Passes!");
        step = 0;
      }
    };
end contract
```

In the above contract, two objects: object1 and object2, are defined in participants section. Step is defined in attributes section. There are two messages: message_1() and message_2(), in the sequence diagram. So there are two trigger rules in the contract accordingly. The "CheckStep1" rule will be triggered when message_1() is called on object1. And step is reset to 1 before message_1() is executed. The "CheckStep2" rule will be triggered when message_2() is called on object2 and the condition "step==1" is true. And step is reset to 2 before message_2() is executed. If the condition is false, this contract rule will not react. In after section, we check if step equals 2, if so, the sequence test in this sequence diagram passes and reset step to the original value 0; otherwise, it fails.

The above is a simple example, consisting only two messages. Let's suppose there are n messages: $message_1$, $message_2$,...$message_n$, and they are invoked in order. When $message_1$ is invoked, set $step = 1$; when $message_2$ is invoked under the condition $step == 1$, set $step = 2$, and the message calls are in the right sequence so far. When $message_i$ is invoked under the condition $step == i - 1$, set $step = i$, and so on. When $message_n$ is invoked under condition $step == n - 1$, set $step = n$. Check if $step == n$, if so, the sequence of the message calls is correct. We will define n trigger rules in the corresponding contract. With respect to the complicate case, we generate a contract template for testing sequence of message calls as follows.

Contract Template for Testing Sequence of Message Calls

```
contract MCS_template
  participants
    participant1:Component;
    participant2:Component;
    ...
    participantn:Component;
  attributes
    boolean result = false;
    int step = 0;
  coordination
    CheckStep1:
      when *->> participant1.operation(parameter(i))
          && (trigger conditions in Java)
      with (JavaGuardConditions)
      failure {
        //Java guard failure actions
        // throw an exception or return a value
      }
      before {
```

```
        step = 1;
    };
  CheckStep2:
    when *->> participanti.operation(parameter(i)) && (step == 1)
    with (JavaGuardConditions)
    failure {
        //Java guard failure actions
        // throw an exception or return a value
    }
    before {
        step = 2;
    };
  ⋮
  CheckStepn:
    when *->> participanti.operation(parameter(i)) && (step == n-1)
    with (JavaGuardConditions)
    failure {
        //Java guard failure actions
        //throw an exception or return a value
    }
    before {
        step = n;
    };
    after {
        if(step == n) {
            result = true;
            System.out.println("Sequence Test Passed!");
            step = 0;
        }
    };
end contract //MCS_template
```

In chapter on UML, we introduced "combined fragment" notion element in a sequence diagram. A combined fragment is used to group sets of messages together to show conditional flow in a sequence diagram. So we could draw a control-flow graph corresponding to a sequence diagram. We give a definition of a control-flow graph for a sequence diagram as follows.

Definition: Control Flow Graph for Sequence Diagram A control flow graph for a sequence diagram is a representation, using graph notation, of all paths that might be traversed through the sequence diagram during its

execution. Each node in the graph represents either a message or a condition (the condition in the combined fragment). The edge represents the flow of the next message or the condition. According to the type of the combined fragment, there are sequential, optional, conditional, and loop, which corresponds to four different control flow graphs, shown in Figure 7.2: (c) represents sequential, (d) represent optional, (a) and (b) represent conditional, (e) represent loop. Defining the values of variable **step** in contracts is different for different control flow graphs. We will introduce the difference as follows.

- **SEQUENTIAL** For sequential messages/calls, **step** is incremented by one in each contract rule, as in the case of the contract template for testing sequence of message calls. In the last contract rule corresponding to the last message in the sequence diagram, check the value of step, and reset **step** to zero.

Contract Rules for Sequential Messages

```
contract //Figure 7.2.(c)
  participants
    participant1:Component;
    participant2:Component;
    ...
    participantn:Component;
  attributes
    boolean result = false;
    int step = 0;
  coordination
    CheckStep1: ...
      ⋮
    CheckStepi:
      when *->> participanti.m_i() && (step == i-1)
      before {
        step = i;
      };
    CheckStepi+1:
      when *->> participanti.m_{i+1}() && (step == i)
      before {
        step = i + 1;
      };
    CheckStepi+2:
      when *->> participanti.m_{i+2}() && (step == i+1)
      before {
```

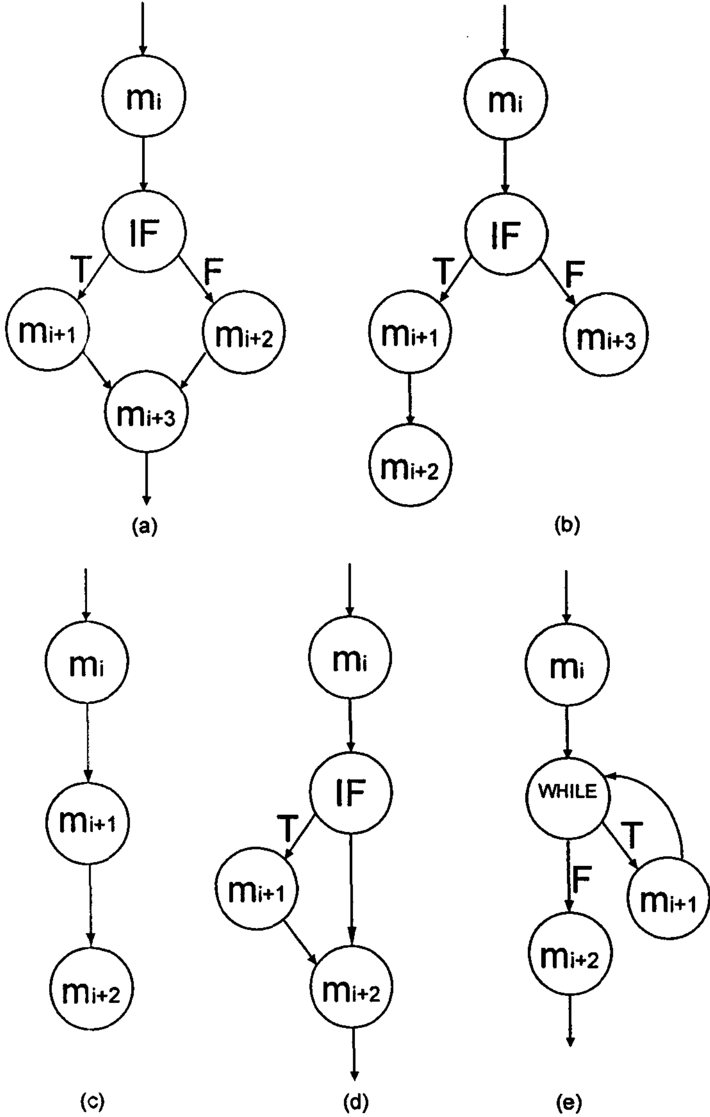


Figure 7.2: Control Flow Graph Examples

```
        step = i + 2;
    after {
        if(step == i+2) {
            result = true;
            System.out.println("Sequence Test Passed!");
            step = 0;
            System.out.println("step is set to 0.");
        }
    };
end contract
```

- **OPTIONAL** such as *if...* without else. Optional messages/calls will be invoked if the optional condition is satisfied. If the condition is satisfied in one scenario, the control flows into the optional compartment, then continues with the rest. Otherwise, the optional compartment will be skipped, like the usual sequential messages. The message right after the optional compartment could be invoked following the messages in the optional compartment or following the message before the optional compartment and skipping the optional compartment. Therefore the contract rule corresponding to the message right after the optional compartment checks the value of step, the value set in the contract rule corresponding to either the last message in the optional compartment, or the message right before the optional compartment.

  Example: Figure 7.2.(d) is part of control flow graph showing optional messages. The following contract segment is designed for Figure 7.2.(d).

  - Suppose $step = i-1$ right before m_i is invoked. When m_i is invoked and at the same time $step == i-1$ is satisfied, set $step = i$ (the contract rule "CheckStepi");
  - Whether m_{i+1} is invoked or m_{i+2} is invoked depends on the value of the *if* condition. If it is true, m_{i+1} is invoked, otherwise m_{i+2} is invoked.
    * When m_{i+1} is invoked and $step == i$ under *if* condition, set $step = i+1$ (the contract rule "CheckStepi+1");
    * When m_{i+2} is invoked and either $step == i$ or $step == i+1$ is satisfied, set $step = i+2$ (the contract rule "CheckStepi+2").

  Please note here the trigger condition (the value of step) in "CheckStepi+2" has two options: either $step == i$ or $step == i+1$, because m_{i+2} may follow m_{i+1} if it goes into the optional compartment or m_i if it skips the optional compartment.

The contract rules design asserts messages/calls invoked are in the right sequence whichever conditional branch is taken.

Contract Rules for Optional Messages

```
contract //Figure 7.2.(d)
  participants
    participant1:Component;
    participant2:Component;
    ...
    participantn:Component;
  attributes
    boolean result = false;
    int step = 0;
  coordination
    CheckStep1: ...
      ⋮
    CheckStepi:
      when *->> participanti.m_i() && (step == i-1)
      before {
        step = i;
      };
    CheckStepi+1:
      when *->> participanti.m_{i+1}() && (step == i)
      with (IfCondition)
      failure {
        //Java guard failure actions
        // throw an exception or return a value
      }
      before {
        step = i + 1;
      };
    CheckStepi+2:
      when *->> participanti.m_{i+2}() && (step == i||step == i + 1)
      before {
        step = i + 2;
      };
    ⋮
end contract
```

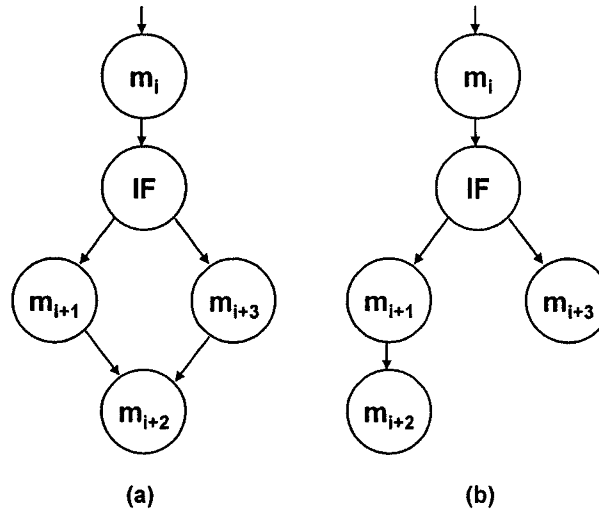


Figure 7.3: Control Flow Graph Examples

- **CONDITIONAL** such as *if then ... else....* For conditional messages/calls, either messages in the *then* compartment or messages in the *else* compartment are executed in one procedure depending on the value of the condition. In order to check that messages are invoked in the correct sequence for one procedure, first of all determine the value of the condition of the *if* statement; if it is **true**, checking if all the methods in the *then* compartment are executed in order and no method in the *else* compartment is executed, vice versa. Other methods outside the *if then...else...* remains the same.

  A special case may occur and need to be dealt with differently: the *if then...else...* compartment is the end of the sequence diagram, which means there is no message after the *else* compartment. In this case, the messages/calls are invoked in the right sequence when the **step** is the value set in the last method of either the *then* compartment or the *else* compartment.

  Example: Figure 7.3.(a) is part of control flow graph showing conditional messages. Figure 7.3.(b) is the special case mentioned above. The following two contract segments are designed for Figure 7.3.(a) and Figure 7.3.(b). Firstly we introduce the contract ConditionalMessage, shown below, for Figure 7.3.(a).

  - Suppose $step = i-1$ right before m_i is invoked. When m_i is invoked and at the same time $step == i - 1$ is satisfied, set $step = i$ (the

contract rule "CheckStepi" in the contract below).

- Whether m_{i+1} is invoked or m_{i+2} is invoked depends on the value of the *if* condition. If it is true, m_{i+1} is invoked, otherwise m_{i+2} is invoked.
  * When m_{i+1} is invoked and at the same time $step == i+1$ is satisfied, set $step = i+1$ (contract rule "CheckStepi+1");
  * When m_{i+2} is invoked and at the same time $step == i+1$ is satisfied, set $step = i+2$ (contract rule "CheckStepi+2").

  Please notice here the trigger condition (the value of **step**) in "CheckStepi+1" and "CheckStepi+2" is the same; the reason is m_{i+1} and m_{i+2} are two possibly consecutive messages after m_i while they are mutually exclusive.
- When m_{i+3} is invoked and either $step == i+1$ or $step == i+2$ is satisfied, set $step == i+3$ (contract rule "CheckStepi+3").

The contract rule design guarantees messages/calls invoked are in the right sequence whichever conditional branch it is taken.

Contract Rules for Conditional Messages

```
contract ConditionalMessage//Figure 7.3.(a)
  participants
    participant1:Component;
    participant2:Component;
    ...
    participantn:Component;
  attributes
    boolean result = false;
    int step = 0;
  coordination
    CheckStep1: ...
      ⋮
    CheckStepi: when *->> participanti.mi() && (step == i-1)
      before {
        step = i;
      };
    CheckStepi+1:
      when *->> participanti.mi+1() && (step == i)
      with (IfCondition)
      before {
```

```
      step = i + 1;
    };
  CheckStepi+2:
    when *->> participanti.m_{i+2}() && (step == i)
    with (!IfCondition)
    before {
      step = i + 2;
    };
  CheckStepi+3:
    when *->> participanti.m_{i+3}() &&
           (step == i+1 || step == i+2)
    before {
      step = i + 3;
    };
  ⋮
end contract
```

Example: Figure 7.3.(b) is part of control flow graph showing conditional messages. But it is a special case mentioned above. The following contract segment is designed for Figure 7.3.(b). The contract rule design is similar to the previous one, but it is different in checking the final result. In this example, there are two messages in the *then* compartment, one message in the *else* compartment.

- Suppose $step = i-1$ right before m_i is invoked. When m_i is invoked and at the same time $step == i-1$ is satisfied, set $step = i$ (rule "CheckStepi" in contract 7.2.1);
- When m_{i+1} is invoked and $step == i$ is satisfied, set $step = i+1$ (contract rule "CheckStepi+1");
- When m_{i+2} is invoked and $step == i+1$, set $step = i+2$ (contract rule "CheckStepi+2");
- When m_{i+3} is invoked and $step == i+1$ is satisfied, set $step = i+3$ (contract rule "CheckStepi+3").
- We have one more contract rule for this case: "StepResultCheck". When $step == i+2$ or $step == i+3$, the whole messages/calss sequence is correct.

Please notice here we use state condition rule for checking the final compared to the usual case; the reason is the final state, in this case, could be $step = i+2$ (m_{i+2} invoked) or $step = i+3$ (m_{i+3} invoked), which is not

easy to do in the last trigger rule. The contract rule design guarantees messages/calls invoked are in the right sequence whichever state it ends in. The following the contract implementation for this special case.

Contract Rules for Conditional Messages - Special Case

```
contract CondMess_Special//Figure 7.3.(b)
  participants
    participant1:Component;
    participant2:Component;
    ...
    participantn:Component;
  attributes
    boolean result = false;
    int step = 0;
  coordination
    CheckStep1: ...
      ⋮
    CheckStepi:
      when *->> participanti.m_i() && (step == i-1)
      before {
        step = i;
      };
    CheckStepi+1:
      when *->> participanti.m_{i+1}() && (step == i)
      with (!IfCondition)
      before {
        step = i + 1;
      };
    CheckStepi+2:
      when *->> participanti.m_{i+2}() && (step == i + 1)
      before {
        step = i + 2;
      };
    CheckStepi+3:
      when *->> participanti.m_{i+3}() && (step == i)
      with (!IfCondition)
      before {
        step = i + 3;
      };
    StepResultCheck: when ? (step == i+2 || step == i+3)
              on participant1, participant2
```

```
        do {
           result = true;
        };
end contract
```

- **LOOP** such as a *while...* loop. For loop messages/calls, the loop body could be executed once, twice,..., n times or none at all. Therefore the messages/calls in the body of the loop could be invoked none, once, twice, even n times in order depending on the values of the loop condition. The contract rule should work for all the cases.

  The coordination rules related with the following messages are the same as those for sequential messages.

  - the messages before the loop body;
  - the messages in the loop body other than the last one;
  - the messages after the loop body other than the first one.

  Only two coordination rules need to be dealt with differently: one is the last message in the loop body, another one is the first message after the loop body.

  1. When it comes to the last message in the body of the loop, we set the value of **step** to the value before it goes into the body of the loop.
  2. When it comes to the first message after the body of the loop, the trigger condition in the coordination rule is whether the **step** equals the value related with the last message before the body of the loop.

  Therefore item 1 guarantees the right order of the messages whenever the loop is executed or not; item 2 guarantees the message invoked before the first message after the loop could be either the last message before the loop (no entry into the loop) or the last message in the loop body (entry into the loop at least once). The message following the last message in the body of the loop consists of the same possibilities as the last message before the loop does.

  The coordination rules guarantee the sequence of the following cases: before the loop, entry into the loop at least once, after the loop and not entry into the loop.

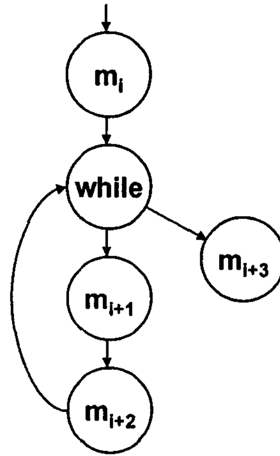


Figure 7.4: Control Flow Graph Examples

Example: Figure 7.4 is part of a control flow graph showing loop messages. The following contract segment is designed for Figure 7.4. In this example, m_i is the last message before the loop; the related **step** value is i. m_{i+1} and m_{i+2} are two sequential messages in the body of the loop. m_{i+3} is the first message after the loop.

- Suppose $step = i - 1$ is invoked just before m_i. When m_i is invoked and at the same time $step == i - 1$ is satisfied, set $step = i$ (coordination rule "CheckStepi").
- When m_{i+1} is invoked and $step == i$ is satisfied with the while loop condition satisfied, set $step = i + 1$ (coordination rule "CheckStepi+1").
- When m_{i+2} is invoked and $step == i + 1$, set $step = i$ (contract rule "CheckStepi+2").
- When m_{i+3} is invoked and $step == i$ is satisfied with the loop condition being false, set $step = i + 3$ (coordination rule "CheckStepi+3").

The coordination rules, such as "CheckStepi+1", "CheckStepi+2", may apply more than once in one procedure depending on the values of the loop condition. The contract rule design guarantees messages/calls invoked are in the right sequence whichever state is the end result.

Contract Rules for Loop Messages

```
contract ContractLoop //Figure 7.4
  participants
    participant1:Component;
    participant2:Component;
    ...
    participantn:Component;
  attributes
    boolean result = false;
    int step = 0;
  coordination
    CheckStep1: ...
      ⋮
    CheckStepi:
      when *->> participanti.m_i() && (step == i-1)
      before {
        step = i;
      };
    CheckStepi+1:
      when *->> participanti.m_{i+1}() && (step == i)
      with (WhileCondition)
      failure {
        //Java guard failure actions
        // throw an exception or return a value
      }
      before {
        step = i + 1;
      };
    CheckStepi+2:
      when *->> participanti.m_{i+2}() && (step == i + 1)
      before {
        step = i;
      };
    CheckStepi+3:
      when *->> participanti.m_{i+3}() && (step == i)
      with (!WhileCondition)
      failure {
        //Java guard failure actions
        // throw an exception or return a value
      }
      before {
        step = i + 3;
      };
```

⋮

end contract

We introduced the contracts for testing the sequences of messages calls in this section above. Basically, each message corresponds to one cooperation trigger rule which checks if the related message is invoked in the right order by the **step** value, that in turn will be reset to a value related with the message being invoked. We also present the difference in the rule between there is a sequential, optional, conditional and loop control-flow in the sequence diagram.

7.2.2 Contract for Testing Parameters

Our approach tests the parameters introduced in the first message of the sequence diagram. We put these parameters into a set, called *ParameterSet*. We only test the parameters belonging to *ParameterSet*. For each parameter in *ParameterSet*, if it is taken in a later message in the sequence diagram, we test if its name, type and value equal to the one in the Parameter Set. We have one cooperation rule related with the first message, and one cooperation rule related with the message whose parameters need to be tested. In our design, we suppose all the parameters are correct, so **result** is initialized to *true*. If any parameter test fails, set **result** to false. The detailed implementation of the contract is as follows.

- In the contract **attributes** section:
  - we define a new variable "expected_parameter(i)" for each parameter(i) in *ParameterSet*, and "expected_parameter(i)" has the same type as parameter(i);
  - we define a boolean variable "precondition" showing whether the first message in the sequence diagram is invoked or not;
  - we define a boolean variable "result" recording the parameter test result for the sequence diagram.
- In the contract **coordination** rules section:
  - In the coordination rule which is related with the first message, in the **before** section, "precondition" is set to true and each "expected_parameter(i)" is initialized to the value of "parameter(i)", parameter(i) $\in$ *ParameterSet* (the coordination rule "Parameter-Precondition").

  - Check each message following the first one in the sequence diagram, if any parameter has the same name as one in *ParameterSet*, create a new coordination rule related with that message, checking whether the parameter is the same as we expected under the trigger condition of the value of "precondition" being true; otherwise, set **result** to *false*.

  In our design, we suppose all the parameters are correct, so **result** is initialized to *true*. If any parameter test fails, set **result** to false. Therefore the value of **result** guarantees the correctness of testing parameters.

The following is the contract template for testing parameters.

Contract Template for Testing Parameters

```
contract TS_template
  participants
    participant1:Component;
    participant2:Component;
  attributes
    boolean precondition = false;
    boolean result = true;
    JavaType expected_parameter(i); // parameter(i) ∈ ParameterSet
  coordination
    ParameterPrecondition:
      when *->> participant1.operation(parameter(i))
          && (trigger conditions in Java)
      with (JavaGuardConditions)
      failure {
        //Java guard failure actions
        //throw an exception or return a value
      }
      before {
        precondition = true;
        expected_parameter(1) = parameter(1);
        ...
        expected_parameter(i) = parameter(i);
      };
    ParameterTest_1:
      when *->> participanti.operation(parameter(i)) && (precondition)
      before {
        if(parameter(i) == expected_parameter(i)) {
          System.out.println("Parameter parameter(i) test is passed.");
```

```
        }
        else {
            result = false;
            System.out.println("Parameter parameter(i) test failed.");
        }
    }; :
  ParameterTest_k:
    when *->> participanti.operation(parameter(i)) && (precondition)
        //k ∈ N
    before {
        if(parameter(i) == expected_parameter(i)) {
            System.out.println("Parameter parameter(i) test is passed.");
        }
        else {
            result = false;
            System.out.println("Parameter parameter(i) test failed.");
        }
    };
end contract //TS_template
```

7.2.3 Contract for Testing Returned Value

In chapter 6, we introduce the approach TOI, a general approach to test objects interactions by checking the post-conditions of each participant after the interaction. It consists of the other two approaches: CAO and DOBE. The details of CAO and DOBE are presented in chapter 6. In this section, we will introduce how to realize the test cases by the mechanism of the approach TOI in terms of coordination contract. The following is the approach TOI and the corresponding contract implementation.

Approach to Test Object Interactions (TOI) Suppose we have a sequence diagram, depicting the interactions among a group of classes: $class_1$, $class_2$,..., $class_n$. Let $object_1$, $object_2$,..., $object_n$ denote the objects of each class, respectively. Given test data, the following operations will be executed sequentially by the sequence diagram specification:

$$object_1.operation_1(parameter_1);$$
$$object_2.operation_2(parameter_2);$$
$$\vdots$$
$$object_n.operation_n(parameter_n).$$

1. Clone Objects. Before the first operation is invoked on the $object_1$ in the sequence diagram, clone each of $object_1$, $object_2$,..., $object_n$ and rename new objects as pre_object_1, pre_object_2,..., pre_object_n using the Approach to Clone an Object (**CAO**), see below.

2. Execute Operations on the Cloned Objects as We Expect. Run

$$pre_object_i.operation_i(parameter_i), i = 1, 2, ..., n$$

   sequentially.

3. Execute Operations on the Objects in Actual Program. Continue to invoke the first operation on the $object_1$, it will trigger all the message invocations in the sequence diagram.

4. Use the Approach to Determine Objects Behavioral Equivalent (**DOBE**) to examine whether the program is executed as we expected by determining if $object_i$ is equivalent to $pre_object_i, i = 1, 2, ..., n$.

Now we give the corresponding contract implementation as follows.

- The contract **participants** section defines n participants by listing pairs of $object_i$ and its class $class_i$, $i = 1, 2, ..., n$.
- In the contract **attributes** section:
  - we define a copy of each participant object, named "pre_participanti" $(0 \leq i \leq n)$;
  - we define n boolean variable "isEqual_Objectk" $(0 \leq k \leq n)$, each of which records the result of the comparison between the participant and its object copy.
- In the contract **cooperation** rule section, only one coordination rule is defined. The trigger is that the first message in the sequence diagram is invoked.
  - In **before** section, we make a clone of each participant object using approach **CAO** with respect to step 1 in the approach TOI;
  - with respect to step 3 in the approach TOI, the original operation is executed if **do** section is omitted;
  - in **after** section:
    1. firstly, we call messages on the cloned objects as we expect, with respect to step 2 in the approach TOI;

2. secondly, with respect to step 4 in the approach TOI, we compare the cloned object instance "pre_participanti" with the original object instance "participanti" using approach **DOBE** and record the result by means of "isEqual_Objecti";
3. lastly, if all "isEqual_Objecti" for $i = 1, 2, ..., n$ are *true*, then we set **result** *true*;

The above contract guarantees the correctness of the objects interactions. The following is a contract template for testing returned value using the implementation mentioned above.

Contract Template for Testing Returned Value

```
contract RV_template
  participants
    participant1:Component;
    participant2:Component;
    ...
    participantn:Component;
  attributes
    //define the same type objects with participanti
    Component pre_participant1;
    Component pre_participant2;
    ...
    Component pre_participantn;
    boolean result = false;
    boolean isEqual_Object1 = false;
    boolean isEqual_Object2 = false;
    ...
    boolean isEqual_Objectn = false;
  coordination
    ReturnValueTest:
      when *->> participant1.operation(arguments)
          && (trigger conditions in Java)
      before {
      //initialize objects
      pre_participant1 = new Component;
      pre_participant2 = new Component;
      ...
      pre_participantn = new Component;
      //make copies of participanti
      //attribute(i) in participant1
      pre_participant1.attribute(i) = participant1.attribute(i);
```

```
    //attribute(j) in participant2
    pre_participant2.attribute(j) = participant2.attribute(j);
    ...
    //attribute(k) in participantk
    pre_participantn.attribute(k) = participantn.attribute(k);
  }
  after {
    pre_participanti.message();
    ...
    //compare two objects
    if((pre_participant1.attribute(1) == participant1.attribute(1)) &&
      ... &&
      (pre_participant1.attribute(i) = participant1.attribute(i)))
      {isEqual_Object1 = true;}
    ...
    if((pre_participantn.attribute(1) == participantn.attribute(1)) &&
      ... &&
      (pre_participantn.attribute(i) = participantn.attribute(i)))
      {isEqual_Objectn = true;}
    if(isEqual_Objectn && ... && isEqual_Object2 && isEqual_Object1)
      {result = true;}
  };
end contract //contract RVTest_template
```

Example of Contract for Testing Returned Value Suppose we have *bank* application, where there are two classes: Class *SavingAccount* and Class *CheckingAccount*. Both classes have two attributes: accountNumber and balance; and two operations: *deposit(int amount)* which subtracts amount money from the current balance and *withdraw(int amount)* which add amount money to the current balance. Class *SavingAccount* has one more operation *transferTo(CheckingAccount chkAccount, int amount)* which transfers amount money from current saving account to a checking account chkAccount, an instance of class CheckingAccount. This method has an instance of another class as one of its parameters. A sequence diagram depicting transfer money from saving account to checking account is shown in Figure 9.1.

Using the implementation introduced above, we generate a contract based on the approach TOI for testing the interactions between two objects. The contract is as follows.

an example of contract for testing returned value

```
contract transferTo_RVTest
```

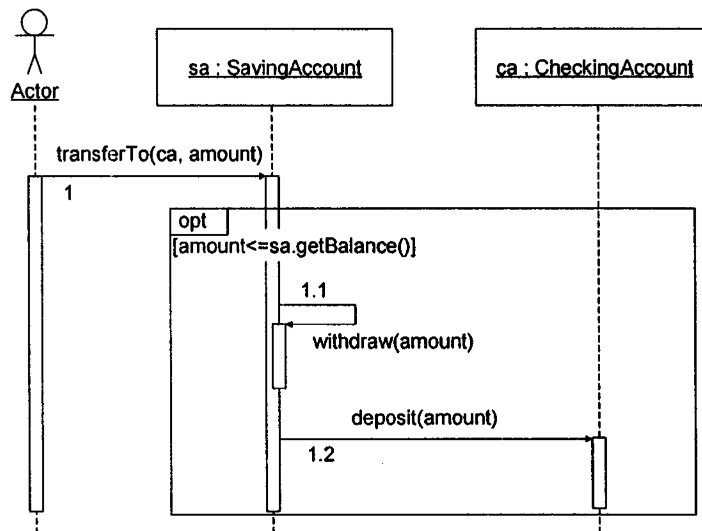


Figure 7.5: Sequence Diagram of "transferTo"

participants
  savingaccount:SavingAccount;
  checkingaccount:CheckingAccount;
attributes
  SavingAccount pre_savingaccount;
  CheckingAccount pre_checkingaccount;
  boolean result = false;
  boolean isEqual_Object1 = false;
  boolean isEqual_Object2 = false;
coordination
  ReturnValueTest:
    when *−>> savingaccount.transferTo(ca,amount) **&&**
      (checkingaccount == ca)
    before{
      pre_savingaccount = new SavingAccount();
      pre_savingaccount.balance = savingaccount.balance;
      pre_savingaccount.accountNumber =
        savingaccount.accountNumber;
      pre_checkingaccount = new CheckingAccount();
      pre_checkingaccount.balance = checkingaccount.balance;
      pre_checkingaccount.accountNumber =
        checkingaccount.accountNumber;

```
    }
    after{
      if(amount <= savingaccount.balance) {
        pre_savingaccount.withdraw(amount);
        pre_checkingaccount.deposit(amount);
      }
      //compare two objects
      if((pre_savingaccount.balance == savingaccount.balance) &&
        (pre_savingaccount.accountNumber ==
          savingaccount.accountNumber))
          {isEqual_Object1 = true;}
      if((pre_checkingaccount.balance == checkingaccount.balance) &&
        (pre_checkingaccount.accountNumber ==
          checkingaccount.accountNumber))
          {isEqual_Object2 = true;}
      if(isEqual_Object2 && isEqual_Object1)
          {result = true;}
      };
end contract
```

7.3 Test Case Execution

Test cases are generated using the concept of coordination contract. The details of the realization of the test cases in the concept of the contracts have been presented in this chapter. From the implementation of the contracts, one test case includes what to test, how to test and how to get the test result. CDE, introduced in chapter 5, is a tool to help develop Java applications using coordination contracts. The CDE translates the contracts into Java classes, which can then be compiled with the other classes of the program under test that form a test framework. We also developed a test driver, which is shown in Figure 8.2 in chapter 6, to run the test framework by taking the test data. Running the test driver with the test data, we get the test result for the program under test at the integration level. The process is depicted by Figure 7.6. Hence, we implement the test execution automation, which is hardly implemented by other integration testing approaches.

In this chapter, we gave an detailed introduction to the implementations of contracts for testing the sequence of the messages calls, testing parameters and testing returned value of the interactions among objects, based on the testing approaches introduced in Chapter 6. In testing the sequence of the messages calls, we defined a control-flow graph for a sequence diagram; and we introduced how to implement the contracts for sequential, optional, conditional

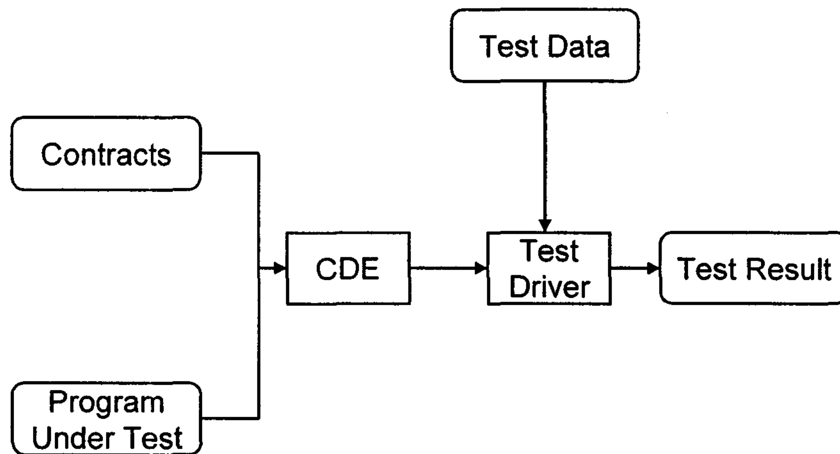


Figure 7.6: Test Execution Process

and loop, respectively. We also gave some examples to help understand the contract implementation.

Having a sequence diagram, we need to generate three contracts for testing the sequence of the message calls, testing parameters and testing returned value respectively. While generating contracts manually could be error-prone and increase test costs. We have developed a tool which takes a sequence diagram and class diagram, and generates the contracts automatically. Next chapter will give the main algorithms using in the tool and justifications of some algorithms.

Chapter 8

Prototype

The contracts for testing object-oriented programs integration testing based on the approaches we introduced in chapter 6 can be generated automatically from the sequence diagrams by a tool we developed. In this chapter, we present the main algorithms used in the tool and the justifications of the algorithms.

8.1 Introduction

A prototype tool has been developed to support the methodologies, known as TSMC, TP, TOI, CAO and DOBE, and enables test case generation and execution automation. The tool was developed using Java SDK 5.0. It consists of three modules: package *DataStructure*, class *XMLParser* and class *DS2Contract*.

- The module *DataStructure* defines the main data structure by the following classes: Object, ObjectCollection, Message, MessageCollection, Class, ClassCollection, Parameter, ParameterCollection, Frame, FrameCollection and CFSEnum. Class *Object* is used to describe the object instances participating in the sequence diagram. Class *ObjectCollection* is a set of objects of Class *Object*. Class *Message* is used for representing the messages in the sequence diagram. Class *MessageCollection* is a set of objects of Class *Message*. Class *Class* is used to describe each class in the class diagram. Class *ClassCollection* is a set of objects of Class *Class*. Class *Parameter* is used to represent the parameter taken in the message in the sequence diagram. Class *ParameterCollection* is a set of objects of Class *Parameter*. Class *Frame* is used to describe the combined fragment notion element in a sequence diagram. Class *FrameCollection* is a set of objects of Class *Frame*. Class *CFSEnum* is a type of enumeration, describing the type (*alt*, *opt* and *loop*) of the combined fragment notion element in sequence diagram. The relationship between the classes is shown in Figure 8.1.

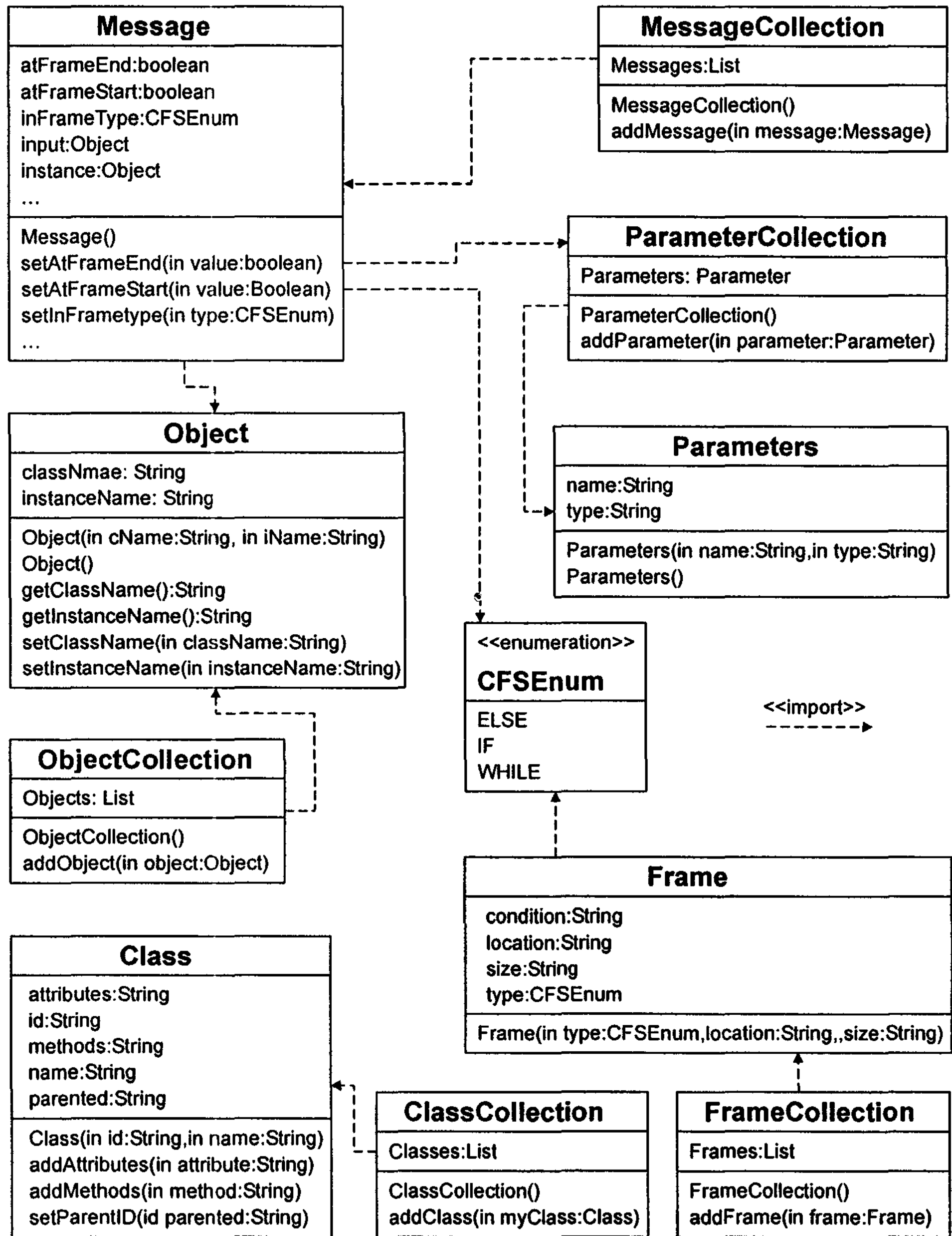


Figure 8.1: Class Diagram of the Main Data Structures

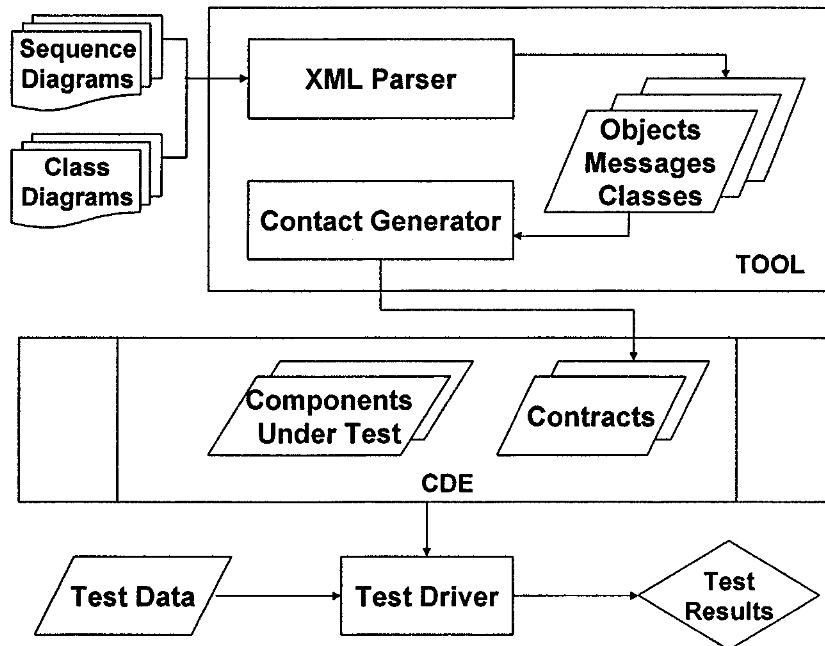


Figure 8.2: Architecture of Integration Testing Approach.

- The module *XMLParser.java* consists of a XML parser based on DOM, which reads sequence diagrams and class diagrams and saves the information about the diagrams in terms of the data structures defined in package *DataStructure*.
- The module *DS2Contract.java* formulates the three contracts corresponding to the given sequence diagrams and class diagrams and generates the three contracts automatically.

The logical structure view of the tool can be still shown in Figure 8.2.

In using the tool, the **INPUT** is sequence diagram and class diagram drawn by Omondo EclipseUML, the **OUTPUT** are coordination contracts designed by the methodologies introduced above. Import the contracts and the components under test into CDE, generate and compile them together. Having successfully compiled all the files, test suites are bundled together with the program under test to form a test framework. Given the test data, we use the test driver to run the test framework. Test results are then generated.

8.2 Algorithms

In this section, the main algorithms in the tool are described and justified. In summary,

- **Algorithm** 1, processSDWithDOM(XMLFile), parses an XML file which represents a sequence diagram, gets all information about the sequence diagram and saves it into *objectsInSD* and *messagesInSD*, which are objects of Class *ObjectCollection* and *MessageCollection*, respectively.
- **Algorithm** 2, CreateContractOne(objectsInSD, messagesInSD), given *objectsInSD* and *messagesInSD* which are generated from algorithm 1, designs contract one for testing the sequence of messages/calls and generates the contract automatically.
- **Algorithm** 3, processCDWithDOM(XMLFile), parses an XML file which represents a class diagram, gets all the information about the class diagram and saves it into *classesInCD*, which is an object instance of Class *ClassCollection*.
- **Algorithm** 4, CreateContractThree(objectsInSD, messagesInSD, classesInCD), given *objectsInSD*, *messagesInSD* and *classesInCD* which are generated from algorithm 1 and algorithm 3, designs contract three for testing the result of the interactions among those objects and generates a contract automatically.
- **Algorithm** 5, getAttributeList(classesInCD, currentClass, attributeList), retrieves all the attributes of the current class including ones from its parent's classes and saves them into *attributeList*.
- **Algorithm** 6, CreateContractTwo(objectsInSD, messagesInSD, classesInCD), given *objectsInSD*, *messagesInSD* and *classesInCD* which are generated from algorithm 1 and algorithm 3, formulates contract three for testing the parameters taken by the messages in the sequence diagram and generates contract automatically.
- **Algorithm** 7, addParameters2Message(classes, currentMessage), attaches the currentMessage's attributes to the message object.

ALGORITHM 1: processSDWithDOM(xmlfile) This call reads an XML file and saves its graphical information into *objectsInSD* and *messagesInSD*.

1. Parse an XML file, get elements by tag name "children", save to NodeList *instanceList*.

<table>
<tr><td></td><td colspan="2">isPreviousMessageInFrame</td><td>¬isPreviousMessageInFrame</td></tr>
<tr><td rowspan="5">MessageInFrame</td><td colspan="3">MessageFrameType = FrameType</td></tr>
<tr><td colspan="3">if(last message), then Message InFrameEnd</td></tr>
<tr><td>two messages in same frame</td><td>not in the same frame</td><td rowspan="2">Message inFrameStart</td></tr>
<tr><td>previousMessage inFrameEnd currentMessage inFrameStart</td><td>¬previousMessage inFrameEnd</td></tr>
<tr></tr>
<tr><td>¬MessageInFrame</td><td colspan="2">previousMessage inFrameEnd</td><td></td></tr>
</table>

Table 8.1: Tabular to Define the Location of the Messages to Frames

2. for each element in instanceList: if the value of attribute "xsi:type" equals "editmodel:InstanceEditModel", initialize an object of class *Object* for which the initial value of *className* and *instanceName* is the value of attribute "id" and "itemName", respectively; add the object to ArrayList *objectsInSD*.

3. for each element in *instanceList*, if the value of attribute "xsi:type" equals to "editmodel:FrameEditModel", initialize an object of class *Frame* which the initial value of type, location, size and condition is the value of attribute "type", "location", "size" and "condition" respectively, add the object to ArrayList *frames*.

4. Parse the XML file again, get elements by tag name "sourceConnections", save to NodeList *messageList*.

5. for each element in *messageList*: if the value of attribute "associatedMethod" is not empty, initialize an object of class *Message* which the initial value of index and messageName is the value of attributes for "associatedSequenceNumber" and "associatedMethod", respectively, the initial value of relatedObject is the return value of method *getRelatedObject(int instanceTargetCode)*, the argument instanceTargetCode is the value of attribute "target"; then add the object to ArrayList *messages*.

6. check the relationship between the *messages* and the *frames*, assign value to the attributes: inFrameType, atFrameStart and atFrameEnd of each *message* by the conditions presented in Table 8.1.

LEMMA: All the messages in a sequence diagram is translated into a $messagesInSD$ of type List.

PROOF: The sequence diagram drawn in Omondo EclipseUML is saved with a *.usd* postfix, and it is a standard XML file, converting it to a tree structure, shown in Figure 8.3. The nodes in the tree represent *actor*, *object*, *message* and *frame* respectively in a sequence diagram:

- The red node (the first node from the left on the second level from the top) ⟨children xsi:type = editmodel : ActorEditMode⟩ corresponds to the *actor* in the sequence diagram;
- the blue node (the second node from the left on the second level from the top) ⟨children xsi:type = editmodel : InstanceEditModel⟩ corresponds to the *object* in a sequence diagram;
- the green node (all the nodes at the bottom level) ⟨SourceConnections xsi:type = editmodel : SequenceMessageEditModels ⟩ corresponds to *message* in a sequence diagram;
- the yellow node (the first node from the right on the second level from the top) ⟨children xsi:type = editmodel:FrameEditModel⟩ corresponds to *frame* in a sequence diagram.

The relationship among these elements is determined by the attributes of the nodes: "location" and "size".

In order to parse the XML file, we could use either Simple API for XML (SAX) or Document Object Model (DOM) API. The tool uses DOM API to parse XML file. A DOM parser reads an entire document and then makes the tree for the entire document available to program code for reading and updating. The sequence diagram information is in ⟨*children*⟩ elements and the value of attribute "xsi:type" determines which role the node is.

For example, element ⟨ SourceConnections xsi:type = editmodel : SequenceMessageEditModel ⟩ records message information. These elements are found by calling getElementsByTagName("SourceConnections"). The *getElementsByTagName* method returns a NodeList; this is a simple collection of Nodes. Each Node is then cast to an Element in order to use the convenience method *getAttribute()*. The getAttribute method gets the message's name. All the messages information are in elements ⟨*SourceConnections*⟩. The *processWithDOM(xmlfile)* in XMLParser class navigates every ⟨*SourceConnections*⟩ element and save message information into *messages*. So all the messages are stored in List *messages* one by one.

ALGORITHM 2: CreateContractOne(objectsInSD, messagesInSD)
The call $CreateContractOne(objectsInSD, messagesInSD)$ generates contract One by manipulating data saved in two objects: $objectsInSD$ and $messagesInSD$.

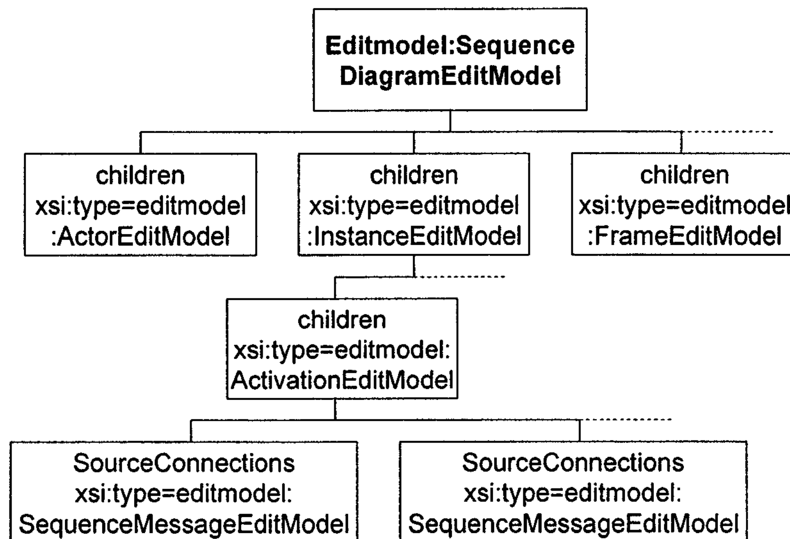


Figure 8.3: Sequence Diagram XML Node Tree

1. take arguments $objectsInSD$ and $messagesInSD$, create iterators: $messageIter$ and $objectIter$.

2. write **contract name** part in contract.

3. write **attributes** part in contract.

4. write **participants** part in contract: for each object in *objectIter*, the values of attributes "instanceName" and "className" are the "participants" and "components" in contract, respectively, separated by a comma.

5. write **coordination rules** part in contract: create a trigger rule for each message object in *messageIter*:

   (a) The **participant** is the value of attribute "className and the **operation** is the the value of attribute "instanceName".

   (b) In the case that the message is not the first message in the sequence diagram: define **trigger condition** by a condition statement $step == counter - 1$, but if the message is the first one in the **alt** frame **else** part, the condition statement is $step == ifConditionStartPoint - 1$.

(c) In the case that the message is the first one in a frame representing a combined fragment in the sequence diagram, add the frame's condition to the **guard condition for the trigger** of the rule, following keyword **with**. Assign the value of *counter* to *ifConditionStartPoint* (*elseConditionStartPoint* or *loopConditionStartPoint*) if the the type of the frame is **if** (**else** or **loop**) by checking the value of the attribute "inFrameType" of the current message.

(d) In the case that the message is the last one in a **loop** frame, set $counter = loopConditionStartPoint - 1$.

(e) In the case that the message is in **else** frame and the last one in *messageIter* as well, write a **state condition rule**; the **condition** is $(step == elseConditionStartPoint - 1)||(step == counter)$, the body of **do** is *result is true.* And set *hasMoreRules* to false.

(f) If *hasMoreRules* is true, write **after** part to the rule. The body of after is *if step equals counter, set result true.*

6. write **end contract**.

7. concatenate each part of the contract, output the contract.

LEMMA: The algorithm 2 preserves the relationship of the message and the receiving object that the message being sent to between the contracts and the sequence diagram.

PROOF: The information about objects in a sequence diagram is in those $\langle children \rangle$ elements with an attribute of $xsi : type = editmodel : Instance-EditModel$, the blue node (the second node from the left at the second level from the top) in figure 8.3. These elements are found by calling the function getElementsByTagName("children") under condition $getAttribute(``xsi : type") == ``editmodel : InstanceEditModel"$.

The method getElementsByTagName() is a Document Object Method; it returns a NodeList of all elements with a specified name. The method getAttribute() is an XML Document Object Method; it gets an attribute value by name. For example, the getAttribute() method gets the object's instance name and class name.

The call "processWithSDDOM(xmlfile)" in class "XMLParser" finds those elements, creates instances of the class "Object" and saves them into a List "objects". So all the objects are stored in a List "objects" one by one sequentially. For one message, its related instance location information is saved in attribute "target". We get its target instance location by calling the method getAttribute("target"). The argument *instanceTargetCode* in the following procedure is the index of the related object in List "objects" (index starts from

1). The value of *instanceTargetCode* is at least 1 because *Actor* corresponds to number 0 while *Actor* can't be the target of any message. Then we get the target instance object by the following procedure.

procedure *getRelatedObject(int instanceTargetCode)*

1: *Iterator iter = objects.Objects.iterator();*
2: *DataStructure.Object relatedObject = new DataStructure.Object();*
3: **while** $(instanceTargetCode - -) \neq 0$ **do**
4: *relatedObject = (DataStructure.Object)iter.next();*
5: **end while**
6: *return relatedObject;*

The object related to a given message is saved in the attribute "instance" of class *Message*; when initialize an instance object of class *Message*, we get the value of the related object by calling the above procedure and assign it to the attribute "instance". Hence for each object "message" in the List *messages*, we can get the related object by getting the attribute "instance" of the object "message". When writing contract, the name of the related object is found by getting the name of the attribute "instance"; the relevant Java code is:

$$currentMessage.instance.getInstanceName()$$

LEMMA: The relationship between the message and its frame in contracts preserves that in the sequence diagram.

PROOF: There are three types of frames in a sequence diagram: **OPT**, **ALT** and **LOOP**. For the ALT frame, we can select to use a frame for an "ELSE" statement or not. So we classify them to four frames: OPT, IF, ELSE and LOOP. Let us assume no nesting among the frames. The relationship between message and frame may be one of the following three:

1. is the message in the frame?
2. is the message the first one in the frame?
3. is the message the last one in the frame?

The answer to question one is the return value of the following method:

isMessageInFrame(fLocation, fSize, mLoc_Y)

The message is in the frame if and only if the message location Y coordinate is between frame location Y coordinate and frame location Y coordinate + size Y coordinate. If the message location Y coordinate is between the frame location Y coordinate and the frame location Y coordinate + size Y coordinate, a call to method $isMessageInFrame(fLocation, fSize, mLoc_Y)$ returns true.

Answers to questions two and three are determined by the value given in Table 8.1.

The relationship between a message and a frame is represented by the values of the attributes defined in class *Message*. These attributes are: *relatedCondition, inFrameType, atFrameStart, atFrameEnd.* The attribute *relatedCondition*, a type of String, denotes the frame's condition. The **empty** value of *inFrameType* means the message is not in any frame. The **true** value of *atFrameStart* means the message is the first one in the frame. Once the location of one message is determined by the answers to the above questions, we save these data to the relevant attributes of message object. Therefore, we can get the relationship between messages and frames in the sequence diagram by getting the values of the attributes of all the instances "message".

LEMMA: The test result given by running the compiled contract one asserts the correctness of the sequence of the messages calls with respect to the specification.

PROOF: In contract one, which is designed for testing the sequence of the messages calls, we define trigger rules, each of which corresponds to one message. We set *counter* = 0 at the beginning. Every time we read a new message, *counter* is incremented by 1, so *counter* represents the sequence number of the current message. *step* is an integer variable, defined in the **attribute** part of the contract. Each message has a unique value of *step*, which is set in the corresponding rule. The variable *step* is used in **trigger conditions** to check the sequence of the message calls. When the trigger message is called, set *step* to 1 in **before** part of contract. When the next expected message is called and *step* equals the value set in previous message's corresponding rule, the sequence of message calls is correct and we set *step* = *counter* in **before** body of contract. Finally, if *step* is the value set in the expected last message, the sequence of message calls in the program preserves that defined in the sequence diagram. How do we get the value of *step* corresponding to the previous message? We answer this question by considering different control flows.

- **No Frame**: shown in Figure 8.4. When *message_1* is called, *step* = 1; when *message_2* is called and the condition *step* = 1 is satisfied as well, set *step* = 2; finally, if *step* = 2, the test result is true.
- **OPTION**: shown in Figure 8.5. There are two cases: there are no more message calls after the optional message or there are more message calls after the optional message.
    1. When *message_1* is called, *step* = 1; when *message_2* is called under condition C and the condition *step* == 1 is satisfied as well,

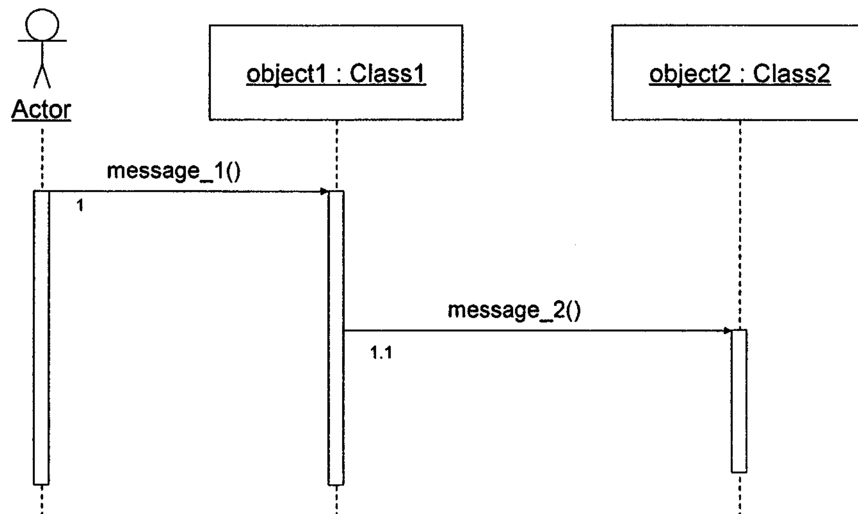


Figure 8.4: Sequence Diagram Example - No Frame

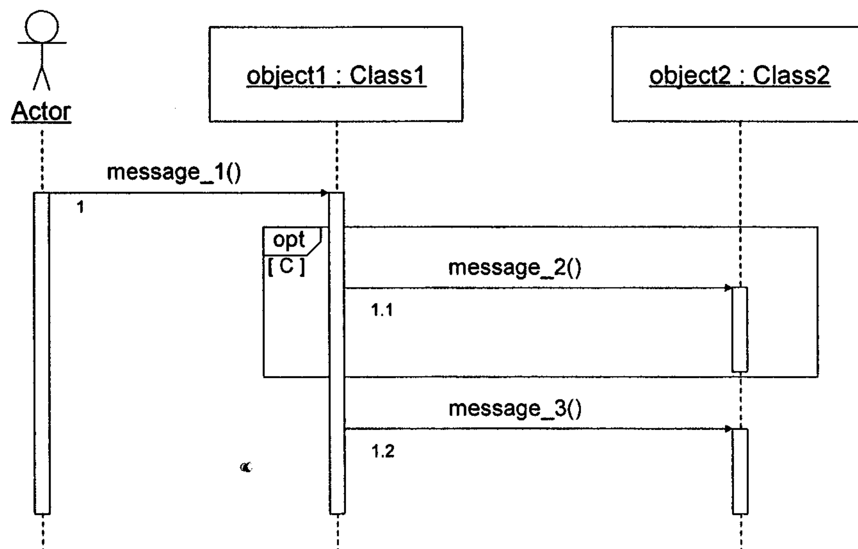


Figure 8.5: Example of Sequence Diagram with Option Frame

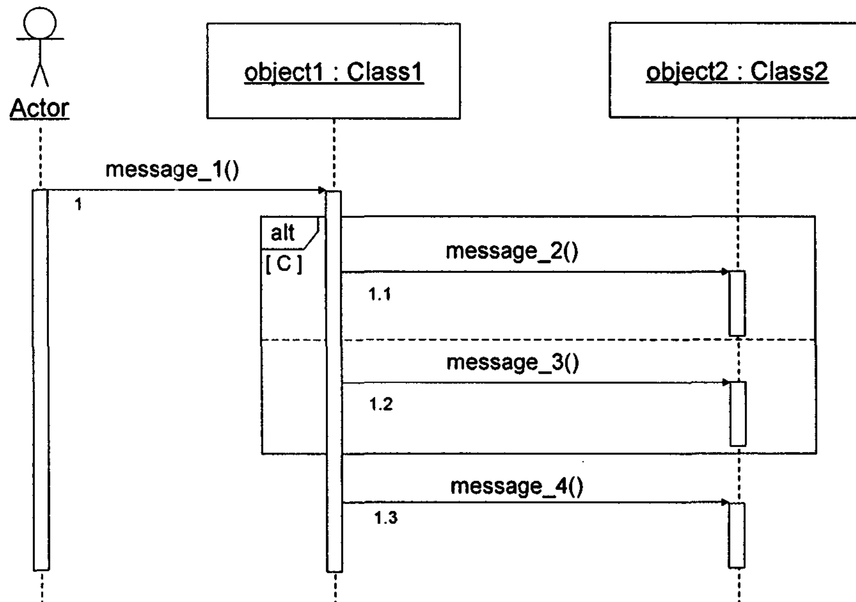


Figure 8.6: Example of Sequence Diagram with Alt Frame

set *step* = 2; finally, if *step* == 1 when condition C is not satisfied, the result is true. If *step* == 2 when condition C is satisfied, the result is true. Whether the optional message is called or not, the *step* value for checking the result is always the value of *step* in the last rule.

2. This case is a continuation of the above case. When *message_3* is called, and *step* == 1(*message_2* is not called) or *step* == 2 (*message_2* is called) is satisfied, set *step* = 3. Finally, if *step* == 3, the result is true.

- **ALT**: shown in Figure 8.6. There are two cases: there is no more message calls after the *else* frame or there are more message calls after the *if...else* frame.

  1. When *message_1* is called, *step* = 1; when *message_2* is called under the IF condition C and *step* == 1 is satisfied as well, set *step* = 2; when *message_3* is called under the ELSE condition $\neg C$ and *step* == 1 is satisfied as well, set *step* = 3. We know *step* = 1 from the value of the *step* defined in the rule corresponding to the message last called before the **if** statement. Finally, if *step* == 2 or *step* == 3, the result is true. This is a special case because the final

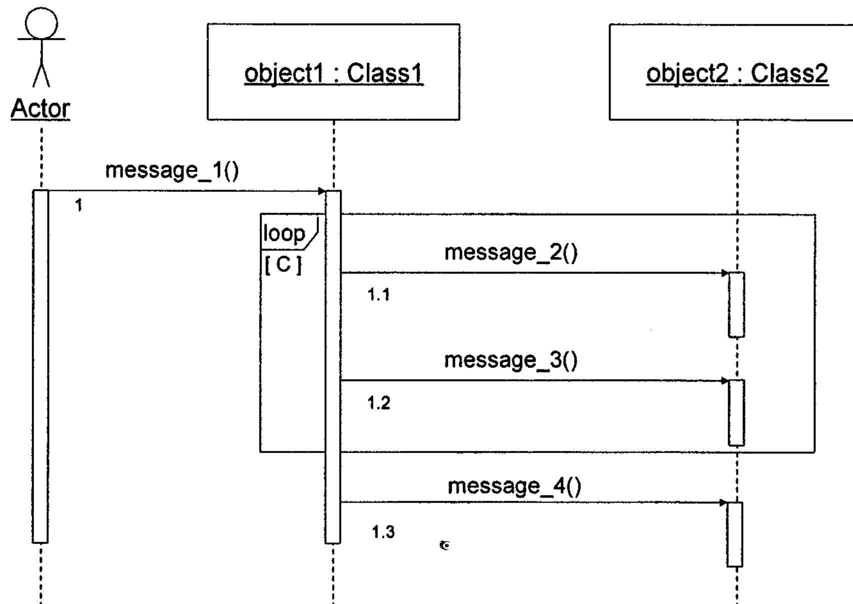


Figure 8.7: Example of Sequence Diagram with Loop Frame

value of **step** has two possibilities. We define a **state condition rule** to determine the test result.

2. When *message_1* is called, $step = 1$; when *message_2* is called under the condition C and $step == 1$ is satisfied as well, set $step = 2$; when *message_3* is called under the condition $\neg C$ and $step == 1$ is satisfied as well, set $step = 3$; when *message_4* is called and either $step == 2$ or $step == 3$ is satisfied, set $step = 4$. Finally, if $step == 4$, the result is true.

If the ALT frame does not include an ELSE statement, we will reuse the argument for OPTION.

- **LOOP**: shown in Figure 8.7. When *message_1* is called, set $step = 1$; when *message_2* is called under the LOOP condition C and $step == 1$ is satisfied as well, set $step = 2$; when *message_3* is called and $step == 2$ is satisfied as well, set $step = counter$. In this case, $counter = 1$ because *counter* has been reset to the value of *counter* corresponding to the last message before the LOOP statement if the current message is the last one in the LOOP frame. This is another special case because we reset *counter* to the value before the LOOP statement begins. If the LOOP condition C is still satisfied, repeat the above steps. When the LOOP

condition C is not satisfied, the program either terminates or has more message calls outside the loop. If the program terminates and $step = 1$ is satisfied, the result is true. If $message_4$ is called under condition $step == 1$, set $step = 4$. Finally, if $step == 4$, the result is true.

ALGORITHM 3: processCDWithDOM(xmlfile) Given a class diagram drawn in Omondo EclipseUML, which is saved in a standard XML file, we can obtain all the necessary class information via a call to method $processCDWithDOM(xmlfile)$.

Class Diagrams and the corresponding XML files: $classesInCD$ is a class collection, collecting all the classes in turn from the class diagram. Each class is an instance of Class *class* with five attributes: id, name, methods, attributes, parentID. Attributes methods and attributes are of **List** type, an ordered collection of all the methods and attributes, respectively. The others are of **String** type. Attribute id is the class name including which package it belongs to. Attribute parentID is the identifier of the parent class.

In the corresponding XML file, each ⟨*children*⟩ element just under the root represents a class. The value of the attribute runTimeClassModel is a string, saving all the methods and attributes in turn. The ⟨*sourceConnections*⟩ element under the first level ⟨*chilren*⟩ element saves the class Inheritance relationship. The parent class name is in the value of attribute id. The XML node tree is shown in Figure 8.8.

The call $processCDWithDOM$ reads an xml file and saves graphical information into $classesInCD$.

1. Parse XML file, get elements by tag name "children", save to NodeList classList. For each element in classList:

   (a) If attribute "xsi:type"'s value equals "editmodel:ClassEditModel", initialize an instance of **Class** *class* which the initial values of classID and className are the values of attributes "id" and "name', respectively.

   (b) All the attributes and methods for the new initialized instance of *class* are the value of attribute $runTimeClassModel$ of the current element. Get the value by calling the method:

   getAttribute("runTimeClassModel")

   Separate *attributes* and *methods* into Lists *attributes* and *methods*, respectively, which are another two attributes of the new initialized instance of *class*, by calling the following method:

   getClassAttributeAndMethod(currentClass, runTimeClass)

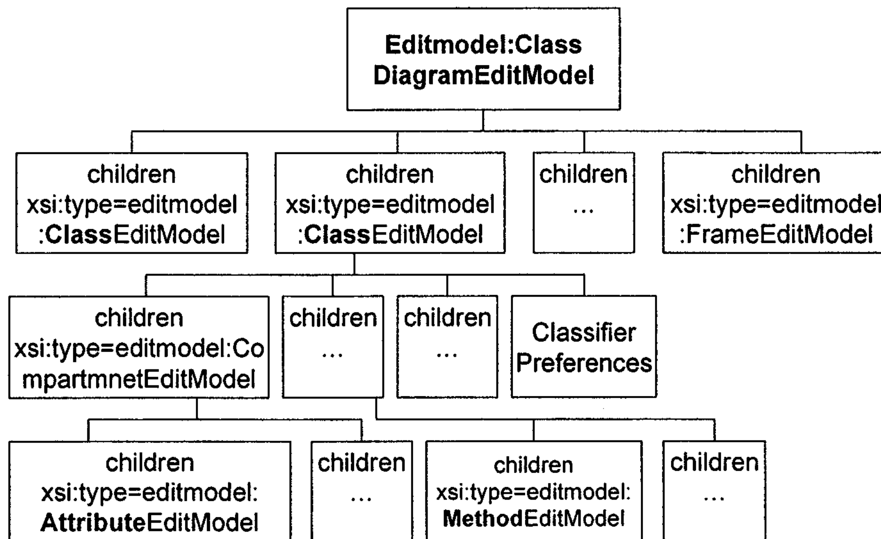


Figure 8.8: Class Diagram XML Node Tree

The idea of the method can be described in the following three steps:

- step 1: read the string *runTimeClass* until char ",";
- step 2: if the substring includes char "(", add it to *methods* of currentClass, otherwise add it to *attributes* of currentClass.
- repeat step 1 and step 2 until there is no more char ","; then repeat step 2 for the rest of the substring.

(c) If the fourth sibling of the first child of the current element is named "sourceConnections", the currentClass has a parent class, whose ID is in the value of the attribute "id". Get the value of the attribute "id" by calling method *getAttribute("id")*. The ID of the parent class is the substring before char '<'. Set the currentClass's parent class ID to the substring.

(d) Add the currentClass to the class collection *classesInCD*.

LEMMA: A class diagram is correctly represented by a relevant object *ClassesInCD*.

PROOF: Class diagram information consists of classes and relationships among them. Information about each class consists of class name, attributes list and methods list. *ClassesInCD* is an instance of class "ClassCollection", which is a list of objects of class "Class". Class "Class" has the following attributes: class name, class ID, attributes, methods and parent id, referring to all the information about classes and relationships among them. All the classes in the class diagram are saved in an instance of class "ClassCollection" via the above algorithm because each class entity, saved in the class diagram as a ⟨*children*⟩ node with attribute "xsi:type=editmodel:ClassEditModel", is visited and a new object of Class "Class" is initialized with the class id and class name, and each object in turn is added to an object *classInCD* of Class "ClassCollection".

The attributes of each object of Class "Class" are assigned according to the class diagram. Class attributes and methods are saved in the attribute "runTimeClassModel" of the ⟨*children*⟩ node as a long string. Parsing the string, we get a lot of separated substrings by commas, each of which represents one attribute (if the string does not end with parentheses) or one method (if the string does end with parentheses).

The relationship among the classes can be found from the attribute "parent id" in Class "Class". If a class has a parent class in the class diagram, then it has a child node "sourceConnections"; the parent class id is saved in the attribute "id" of the ⟨*sourceConnections*⟩ node. Get the parent id and store it in the attribute "parent id" of the current class. Therefore the object *ClassesInCD* can represent the class diagram correctly.

ALGORITHM 4: CreateContractThree(objects, messages, classes)
The call *CreateContractOne*(*objects*, *messages*, *classesInCD*) generates contract one by taking *objects*, *messages* and *classesInCD* as the arguments.

1. take arguments *objects* and *messages*, get the iterator: *messageIter*, *objectIter* and *objectIter2*.

2. write **contract name** part in contract. Contract name is concatenation of two substrings: one is the first message in the sequence diagram, the other is "_RVTest", meaning Return(R) Value(V) test.

3. write **attributes** part in contact. There are two parts in attributes for contract:

    (a) for each object in *objectIter*, define a variable of the same class type and name it as "pre_< *objectName* >".

    (b) define local boolean variables, indicating the result of comparing two objects, assign the initial values "false".

   (c) define a local boolean variable indicating the testing result, name it as "result" and assign the initial value "false".

4. write **participants** part in contract: for each object in objectIter, write attribute value instanceName and className into the "participants", separated by comma.

5. write **coordination rules** part in contract. There is only one coordination rule in this contract: when the first message in the sequence diagram is called under the condition:

   (a) write the **before** part in the rule: make a copy of each object in objectIter.

   (b) write the **after** part in the rule: call methods in turn according to the sequence diagram by the copy of the object instead of the object itself; compare the object and its copy in turn; determine the test result.

6. write **end contract**.

7. combine each part of the contract in the whole, and output the contract.

ALGORITHM 5: getAttributeList(classesInCD, currentClass, attributeList) This is a recursive algorithm. It takes three arguments: DataStructure.ClassCollection myClasses, DataStructure.Class currentClass, and List attributeList, returns List attributeList. The purpose of the algorithm is to get all the attributes of the current class, including all its parent classes and save them into the List attributeList.

procedure *getAttributeList(classesInCD, currentClass, attributeList)*

```
1: if currentClass.parentID.length()> 0 then
2:    parentClass = getClassByID(myClasses, currentClass.parentID);
3:    attributeList = getAttributeList(myClasses, parentClass, attributeList);
4: end if
5: Iterator classAttriIter = currentClass.attributes.iterator();
6: while classAttriIter.hasNext() do
7:    String currentAttr = (String)classAttriIter.next();
8:    attributeList.add(currentAttr);
9: end while
10: return attributeList;
```

ALGORITHM 6: CreateContractTwo(objects, messages, classes) This call generates contract two by taking *objects*, *messages* and *classes* as arguments. Contract two is designed to test parameters appearing in other

message calls in the sequence diagram consistent with the input parameters to the sequence.

1. take arguments *objects* and *messages*, get the iterator: *messageIter*, *objectIter* and *objectIter3*.

2. write **contract name** part in contract. Contract name is concatenation of two substrings: one is the first message in the sequence diagram, the other is "_TSTest", meaning Type(T) Signature(S) test.

3. write **attributes** part in contact. There are two parts in attributes for contract:

   (a) define two boolean variables: "precondition" and "result" and assign the initial values "false".

   (b) for each parameter in current message, obtained by calling *addParameters2Message(myClasses, currentMessage)*, define a variable with the same type and name with "expected_" as prefix.

4. write **participants** part in contract: for each object in objectIter, write attribute value instanceName and className into the "participants", separated by comma.

5. write **coordination rules** part in contract.

   (a) write the first coordination rule, "ParameterPrecondition": when the first message in the sequence diagram is called and the trigger condition is satisfied, in this rule's **before** section: set "precondition" true, and set all the parameter variables defined in the attributes part equal to the parameters taken by the message.

   (b) the remaining coordination rules except the last one correspond to the testing of parameters taken in other messages. For each message except the first one, get the parameter list by calling method getParameterList(currentMessage). For each parameter, if it is introduced in the first message, write a rule to check if the parameter is equal to the one in the first message, including value and type, and add a boolean variable in attributes to record the test result; also add the boolean variable value in a list "ResultList" to track the overall result.

   (c) the last coordination rule, "ParameterResultCheck", checks the overall result. If all the boolean variables in "ResultList" are true, set $result = true$.

6. write **end contract**.

7. combine each part of the contract in the whole; output the contract.

LEMMA: Contract two tests the message parameters correctly.

PROOF: Our approach only tests the parameters introduced in the first message of the sequence diagram, which are all the elements in $ParameterSet$. We have the following relationship:

$$parameter(i) \in ParameterSet \Longleftrightarrow parameter(i) \text{ in first message of SD}$$

The message may be invoked or not according to the control flow graph. If one message is invoked, each parameter is compared with the elements in $ParameterSet$. If there is a matching one in the $ParameterSet$, a coordination rule is created in the contract, testing if the parameter equals the expected one in the $ParameterSet$. Boolean variable "result" is initialized as "true". If the type or value of the parameter is changed, "result" is assigned to "false" and a system message is sent about which parameter testing failed. Finally, the value of variable "result" gives the parameter testing result: if it is "false", some parameter is not correct; if it is "true", no incorrect parameter is found among invoked messages.

ALGORITHM 7: addParameters2Message(classes, currentMessage)
This call attaches the current message's parameters to the message. The class **message** has an attribute "parameters" whose type is class **ParameterCollection**. Class **ParameterCollection** has an attribute "parameters" whose type is **ArrayList** and each element type in the list is class **parameter** which has two attributes "name" and "type". An object *message* is initialized when parsing the sequence diagram, from which we can only get the parameter name for *message*. While parameter type is saved in object *classes* when parsing the corresponding class diagram. This algorithm matches each parameter name to its type and saves them to the current message; refer to the Figure 8.9.

procedure $addParameters2Message(classes, currentMessage)$

1: Initiate an object of class **ParameterCollection**, named "parameterColl"
2: Retrieve object **myClass** by class name in currentMessage.
3: Search by currentMessage's message name in **myClass** messages list, assign the search result to string "methodIdCD".
4: **if** $methodInCD.length > 0$ **then**
5:     intercept parameters names and assign the value to string "paramName"
6:     intercept parameters types and assign the value to string "paramType"
7:     **while** $paramName.length() > 0$ **do**
8:         **if** $paramName.contains(``,")$ **then**
9:             assign the front string "paramName" before "," to string "name"
10:             assign the corresponding part of string "paramType" to string "type"
11:             **if** $paramType.startWith(``L")$ **then**

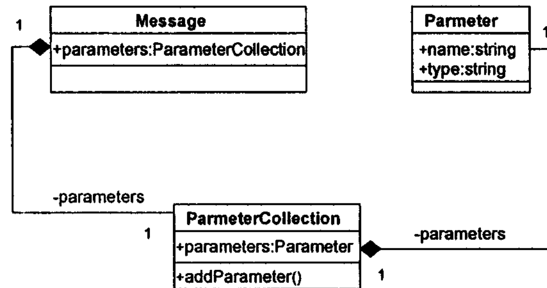


Figure 8.9: Relationship between three classes.

12: assign the part of string "paramType" before ";" to string "type".
13: reset "paramType" to the rest of string "paramType" after ";"
14: **else**
15: assign string "type" value as int, boolean, char, double, float, long, short if the first char of "paramType" is I, Z, C, D, F, J, S respectively.
16: reset string "paramType" to *paramType.substring*(1).
17: **end if**
18: Initiate a new object "myPara" of class **Parameter** with the initial value of "name" and "type";
19: add "myPara" to "parameterColl".
20: **else**
21: **if** *paramType.startWith*("*L*") **then**
22: assign the string "paramType" to string "type".
23: **else**
24: assign string "type" value as int, boolean, char, double, float, long, short if the first char of "paramType" is I, Z, C, D, F, J, S respectively.
25: **end if**
26: Initiate a new object "myPara" of class **Parameter** with the initial value of "paramName" and "type";
27: add "myPara" to "parameterColl".
28: reset "paramName" to empty string
29: **end if**
30: **end while**

SYMBOL	I	Z	C	D	F	J	S	Ljava.lang.String;	L⟨*class*⟩;
TYPE	int	boolean	char	double	float	long	short	String	⟨*class*⟩

Table 8.2: Relationship between symbols and types

31: **end if**
32: assign "parameterColl" to current message's attribute "parameters".

LEMMA: Both name and type of parameters added to the message are correct, and no parameter is missing.

PROOF: Parameters in the message from the sequence diagram are only the names of parameters. Parameters in the corresponding method from the class diagram are only the types of parameters. And the names match the types correspondingly, meaning each name corresponds to one type. Names are separated by commas in the message, Types are different. If the type is a type of "String" or other classes defined by the user, it begins with capital letter "L" and ends with a semicolon; otherwise, the type is only one capital letter 'I', 'Z', 'C', 'D', 'F', 'J', 'S', representing int, boolean, char, double, float, long, short, respectively. The relationship described above is shown in Table 8.2.

Get one name by reading the string "names" until reaching a comma, get the corresponding type by reading the string "types" until reaching a semicolon; the real type is the substring if it starts with capital 'L' or there is only one capital letter in the substring; otherwise we need to convert the symbol to the real type by Table 8.2. Repeat the same operation until there is no more comma in names. Then repeat the operation once for the last name and type. Therefore name and type are matched correctly and none is missing.

Chapter 6 introduces the approaches to generate test cases from UML sequence diagrams and class diagrams for object-oriented programs integration testing. Chapter 7 presents how to realize the test cases in the concept of coordination contract and execute tests automatically using CDE. We have developed a tool to generate the contracts from sequence diagrams and class diagrams automatically. In this chapter, we introduced seven main algorithms used in the tool, developed in Java, and justified each algorithm. These three chapters are the main part in our research work and also our contributions to object-oriented programs integration testing. In the next chapter, we will give a case study showing how to apply our approaches and how to use the tool and the CDE to help testing.

Chapter 9

Case Study

In this chapter, we describe how to test a subsystem using our approach. We have a component *bank*, including three classes: *Account*, *CheckingAccount* and *SavingAccount*. Class *Account* has two attributes: *accountNumber* and *balance*, both are of type the integer. Class *Account* has two methods: *deposit(int amount)* which subtracts the amount money from the current balance and *withdraw(int amount)* which adds amount money to the current balance. Class *CheckingAccount* and Class *SavingAccount* are two sub-classes of Class *Account*, inheriting attributes and methods in class *Account*. Class *SavingAccount* has its own method *transferTo(CheckingAccount chkAccount, int amount)* which transfers the amount money from the current savings account to a checking account chkAccount, an instance of class CheckingAccount. This method has an instance of another class as one of its parameters. Any faults in this method cannot be detected by unit testing because it involved two classes.

9.1 Test Process

The detailed steps are presented as follows.

- **Analyze UML Specifications and Identify Components under Test.** We have been given UML sequence diagram and class diagram specifications, shown in Figure 9.1 and Figure 9.2. We identify the components involved in the interactions presented by the sequence diagrams; and find the corresponding class diagram.

- **Generate Contracts Automatically.** We model the test suites by required collaborations using contracts. Given the input of the sequence diagram and class diagram, we run our tool to generate three contracts automatically:

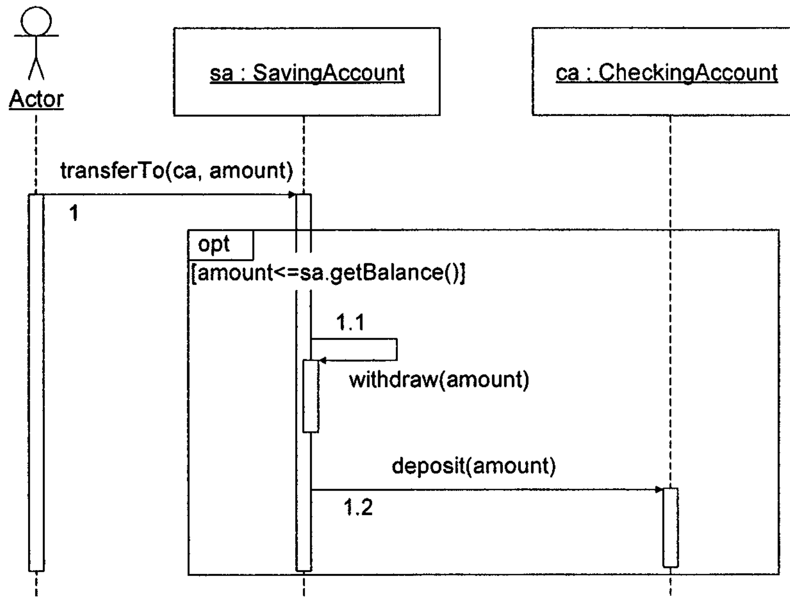


Figure 9.1: Sequence Diagram of "transferTo"

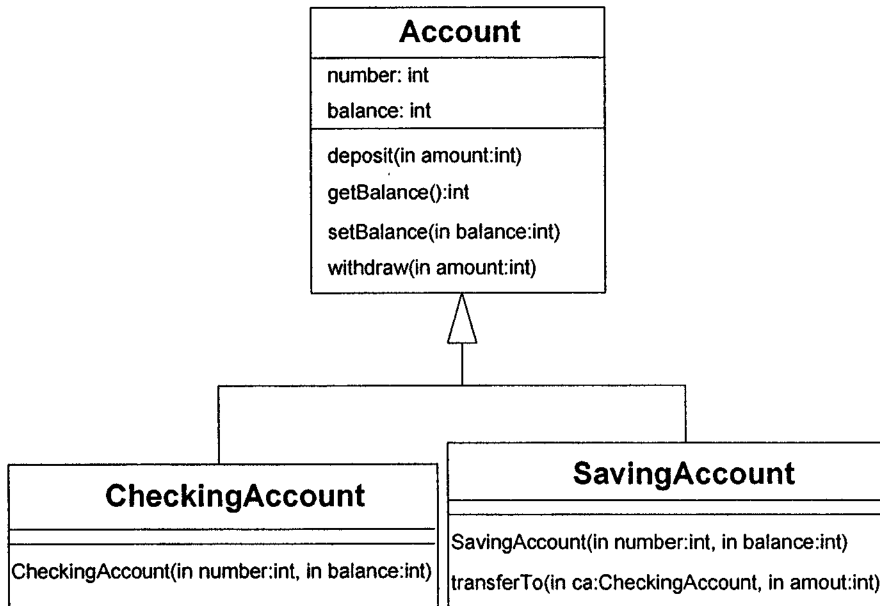


Figure 9.2: Class Diagram of Bank Accounts

  - contract transferTo_MCSTest.ctr for testing the sequence of the message calls;
  - contract transferTo_TSTest.ctr for testing parameters;
  - contract transferTo_RVTest.ctr for testing the interactions among objects.

  The code of the three contracts are shown in Appendix A. Currently we have the components under test and corresponding contracts for test ready and they are put in the subdirectory *src*.

- **Add Components into CDE.** In CDE, we build a new project for this test task by clicking File->New Project and we name it *bank.cdp*. The *component* subdirectory is also created. We add the three components under test into CDE and CDE saves them in the *component* subdirectory. The original *Account.java* file in *src* subdirectory has been renamed as *Account.java.original*, and similarly for the *SavingAccount* class and *CheckingAccount* class.

  The reason for this is that the CDE has to adapt the component classes in order for them to work with contracts. For that purpose, the CDE will change the original Java files and put the result in the generation directory, together with the compiled contracts. The files in the generation and source directories make up the application, and therefore there cannot be two files with the same name in those directories: hence, all files in the source directory that implement classes under coordination must be renamed [4].

- **Add Contracts into CDE.** We add the three contracts generated by our tool into CDE and the contracts are compiled automatically by CDE. The CDE contract compiler checks whether a specific contract specification is consistent with the CDE specification for contracts and, also, if its Java sections are correct in terms of Java syntax. We use the CDE to develop the Java version of our contracts on top of the components under test and generate the code that implements the adapted components and the contracts.

  The differences between the contracts at the modeling level and the CDE-Java specific ones are that the former are abstract and the latter are superposed on top of existing specific Java components. All the contracts classes are saved in package *cde.contracts* under the generation subdirectory.

- **Generate Test Framework.** Using CDE to generate the contracts and the components is to produce the Java code that implements the micro-architecture that we have introduced in chapter 5.4 for allowing

	accountNumber	balance	amount transferred
CheckingAccount	1	100	26
SavingAccount	2	200	

Table 9.1: Test Data One for Bank Account Integration Testing

	accountNumber	balance	amount transferred
CheckingAccount	1	100	250
SavingAccount	2	200	

Table 9.2: Test Data Two for Bank Account Integration Testing

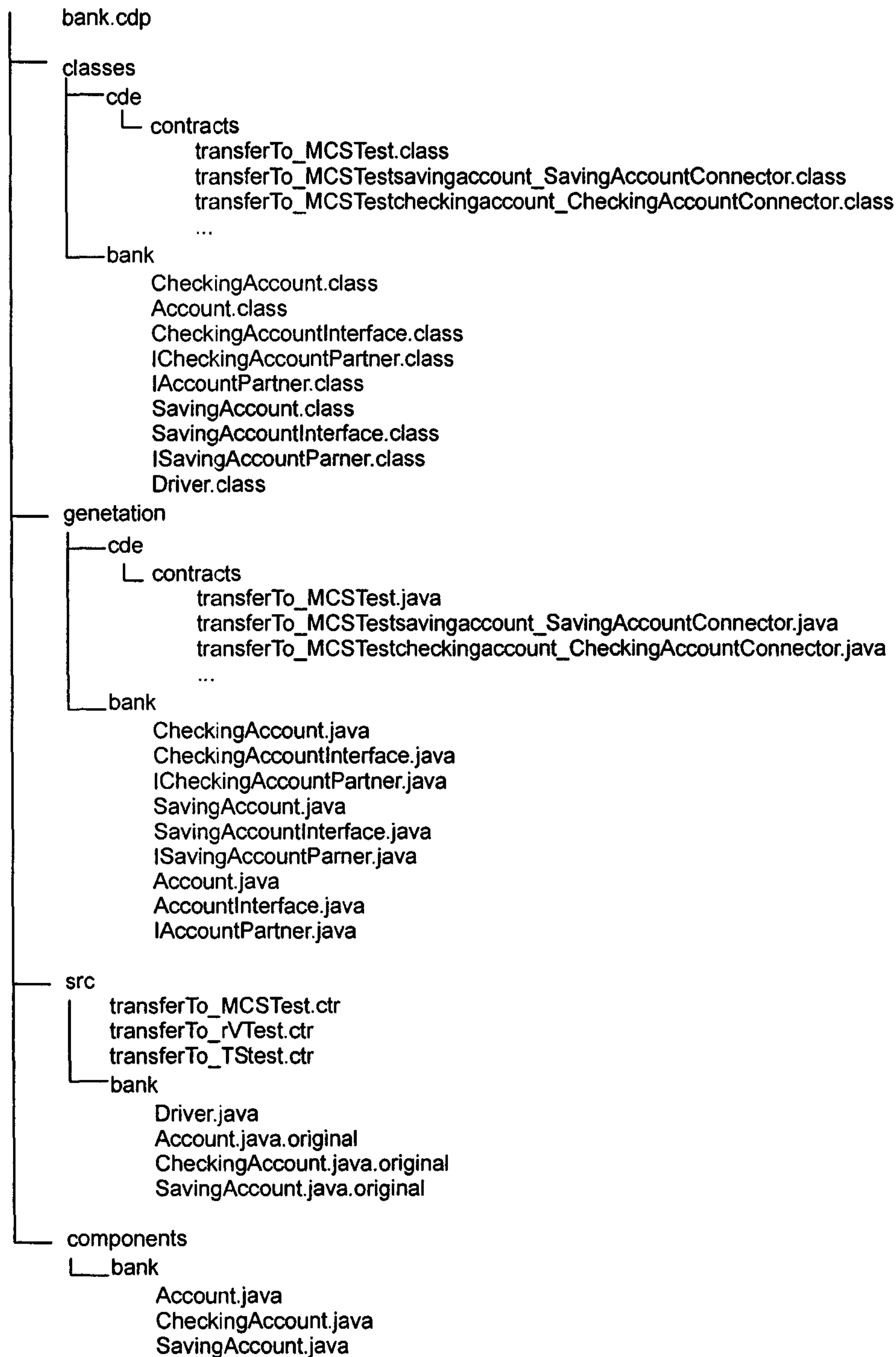
the coordination contracts to be superposed on the components without the latter being aware of the contracts existence.

Now we have built a test framework that uses the coordination contracts API to dynamically test the interactions between the objects in the components under test. The whole directory structure and files in each subdirectory are shown in Figure 9.3.

- **Create Test Driver and Compile it.** We now develop a Java final application by applying instances of the contracts to the instances of the components, also known as participants objects. By doing so, the selection of the test aspects of our integration is feasible by simply reconfiguring the contracts and components relationships (add, delete, substitute, change contracts). The application is in file *Driver.java* in the source directory. To compile and run the application, we open file *Driver.java* in the src/bank by directory by selecting File,Open File...; select Project, Compile, Java Compile.

- **Generate Test Data.** We generate the test data using tool JAT[17] based on the Branch Coverage criterion. In this example, the following test data are required: the initial values of *number* and *balance* for CheckingAccount and SavingAccount, respectively, and the value of *amount* which is going to be transferred from the SavingAccount to the CheckingAccount. Two sets of the test data are shown in Table 9.1 and Table 9.2, respectively.

- **Execute Tests and Generate Test Results.** In Windows, open a Command prompt, go to the application directory and execute the command *java -cp "classes;CDE" bank.Driver*, where classes is the classes subdirectory under this project and CDE is the CDE runtime library that we have indicated in the project options. After running the test


bank.cdp
classes
cde
contracts
transferTo_MCSTest.class
transferTo_MCSTestsavingaccount_SavingAccountConnector.class
transferTo_MCSTestcheckingaccount_CheckingAccountConnector.class
...
bank
CheckingAccount.class
Account.class
CheckingAccountInterface.class
ICheckingAccountPartner.class
IAccountPartner.class
SavingAccount.class
SavingAccountInterface.class
ISavingAccountParner.class
Driver.class
genetation
cde
contracts
transferTo_MCSTest.java
transferTo_MCSTestsavingaccount_SavingAccountConnector.java
transferTo_MCSTestcheckingaccount_CheckingAccountConnector.java
...
bank
CheckingAccount.java
CheckingAccountInterface.java
ICheckingAccountPartner.java
SavingAccount.java
SavingAccountInterface.java
ISavingAccountParner.java
Account.java
AccountInterface.java
IAccountPartner.java
src
transferTo_MCSTest.ctr
transferTo_rVTest.ctr
transferTo_TStest.ctr
bank
Driver.java
Account.java.original
CheckingAccount.java.original
SavingAccount.java.original
components
bank
Account.java
CheckingAccount.java
SavingAccount.java

driver with the given test data, tests are executed and test results are displayed, including sequence of message calls test result, parameter test result and objects interactions test result.

9.2 Test Coverage

The faults related with the interactions can be found efficiently using our approach. An experimental data shows the sequence of message calls, the message parameters and the return values can be tested sufficiently. The faults related with the three parts mentioned above can be found using our approach. Let us introduce some faults in the code under test on purpose:

- We exchange method withdraw(amount) and method deposit(amount).
- We change the parameter *amount* in method withdraw to *amount*1.
- We add another method setBalance(number) to reset the balance of the checking account.
- We take off the condition in the code. In other words, methods withdraw(amount) and deposit(amount) are executed anyway.

The above faults in the code under test can be detected using our approach. It even can detect more faults which are not listed above.

9.3 Another Case

Let us look at another more complicated example. We have four classes: *Bank*, *Check*, *AccountLedger* and *CheckingAccount*. The classes and their relationship are shown in Figure 9.4. Class *CheckingAccount* inherits from class *Account*. Class *AccountLedger* has an attribute *accounts* of type **List**. *AccountLedger* stores all the bank accounts in the subsystem. A new account will be added into List accounts by calling method addAccount(account). We can also get an account by calling method retrieveAccount(number). Class Bank has an attribute ledger of type **AccountLedger**. Method cashCheck(check) in class *Bank* gets the amount of money and the account number by calling getAmount() and getNumber() on object check, then get the account by calling method retrieveAccount(number) on object ledger, and get the balance by calling method getBalance() on object account. If the condition "balance >= account" is true, send message addDebitTransaction(amount) and storePhotoOfCheck(check) to object account; otherwise, send message insufficientFundFee() to object account and message returnCheck(check) to object myBank. This scenario is shown by the sequence diagram in Figure 9.5.

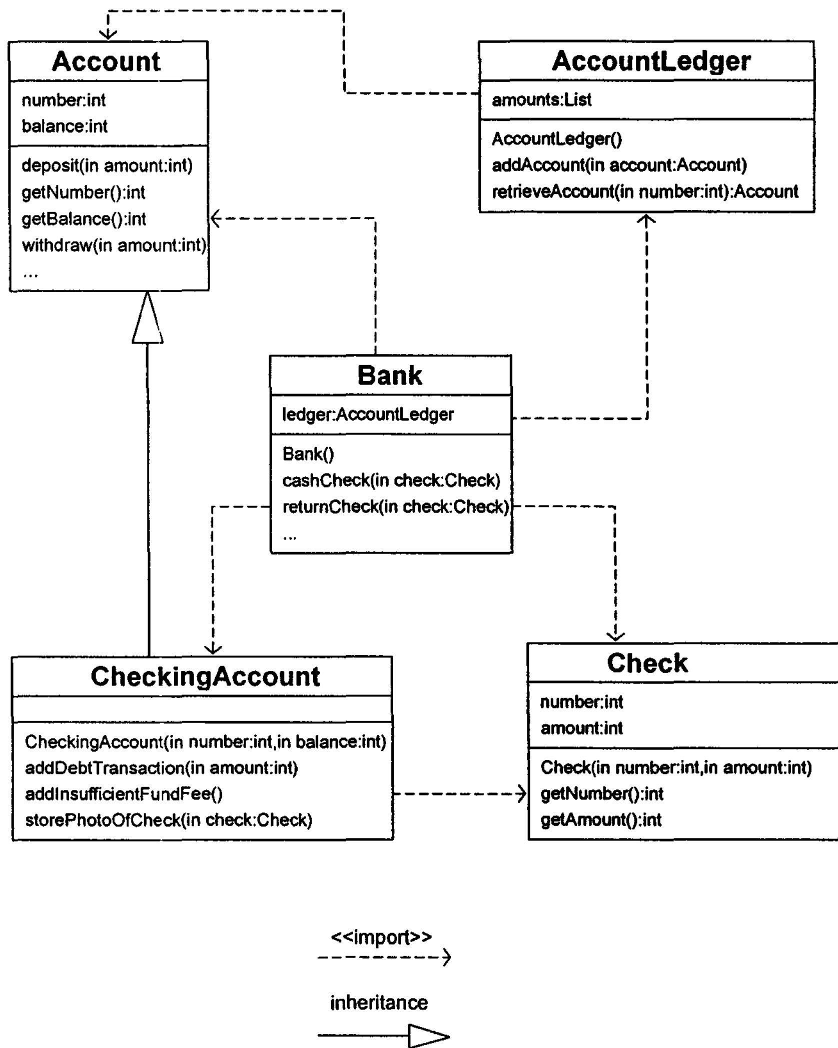


Figure 9.4: Class Diagram of Bank Subsystem

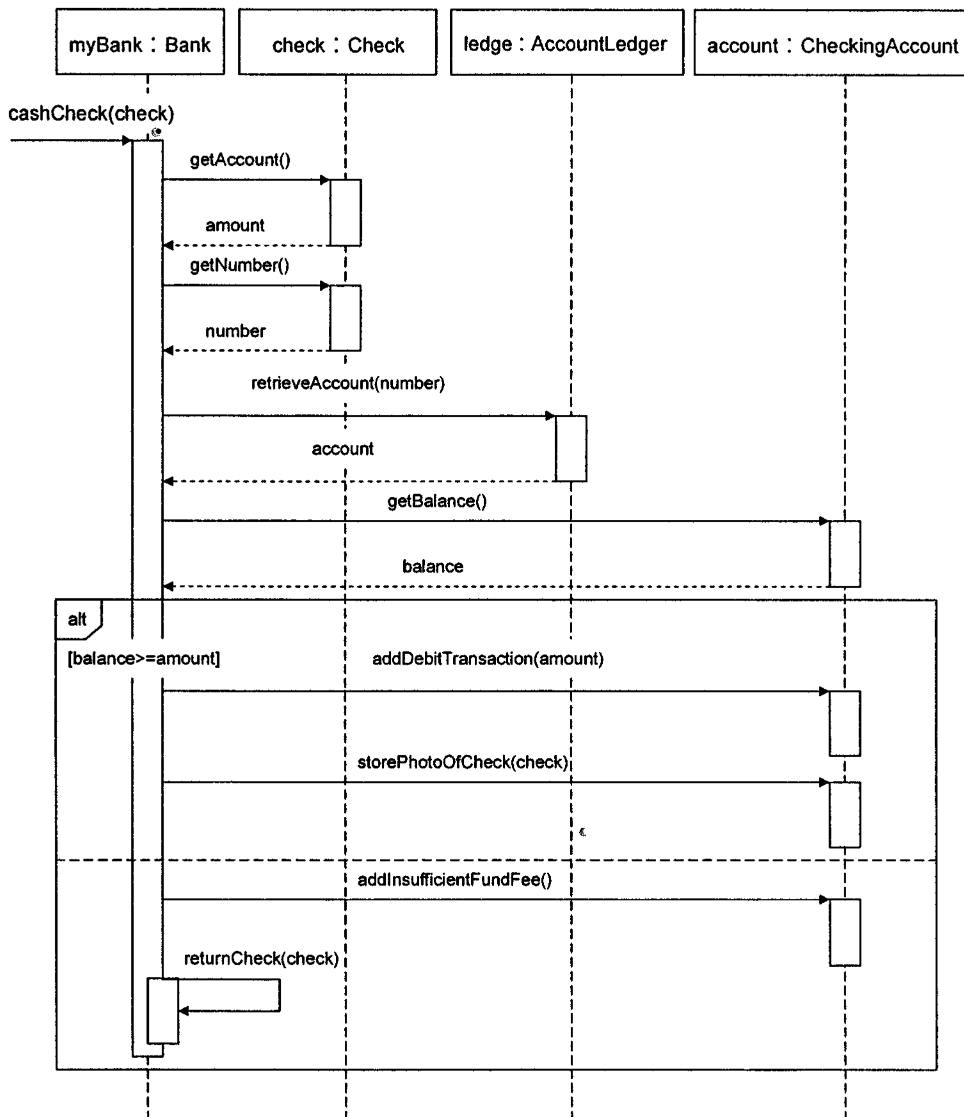


Figure 9.5: Sequence Diagram of “cashCheck(check)”

The purpose of the testing is to find faults related with the interactions among the objects participating in this scenario. The details of the testing steps are similar to the previous one. The contracts are displayed in the Appendix B.

In this chapter, we presented two cases about how to test a subsystem using our approach and the tools in details. One is transferring the amount of money from a saving account to a checking account; another is to cash a check from a checking account. We also gave the experimental test result if there are some faults in the code under test.

Chapter 10

Conclusion and Future Work

10.1 Conclusion

Object-oriented programming is very popular in software development because of its unique features, which facilitate software reuse and component-based development. Integration testing plays a very important role in improving object-oriented software quality. Our research work is about integration testing for object-oriented programs. We have proposed a systematic approach to test object-oriented programs at the integration level. In our approach, we generate test cases from UML sequence diagrams and class diagrams and realize the test cases using the concept of coordination contract. We have developed a tool to generate the contracts for a sequence diagram automatically, based on the mechanism of test case generation we designed.

Our research work presents a new approach for software integration testing. It makes some progress in both test case generation and test case execution for object-oriented program integration testing. The most distinct features of our approach is as follows.

- In recent years, more and more software engineers would like to use UML diagrams to specify software design. UML is a standardized specification language for object modeling using thirteen modeling diagrams. We generate test cases for the integration testing from UML sequence diagrams and class diagrams directly. A sequence diagram depicts an interaction by focusing on the sequence of messages that are exchanged. Our approach is completely based on UML models. Test cases generated from UML sequence diagrams give us good coverage for the interactions among objects. Therefore, it is capable of revealing the faults related with the interactions among components.

- Test cases are implemented by the concept of coordination contracts. The coordination contract is related with the idea of the association class

in UML. It superposes behaviors on the components without interfering their implementation. One of the most important advantages of using the contract is that test cases can be added, deleted, and modified in terms of contracts without interfering with the implementation of the program under test and other established test cases. This strengthens the flexibility of test case generation greatly. As a result, it supports the cooperation of team work very well.

- Executing test cases and checking test results manually is a very tedious task for the software testers in industry and is very error-prone. In our approach, test case execution can be implemented automatically by using the CDE to generate the components and the related contracts into an executable application. Test case execution automation has been considered along with test case design in the concept of contract. The CDE translates the contracts into Java classes, which then can be compiled with the other classes that form a test framework. We can get the test result by running the test framework by a test driver with generated test data.

Even though UML is widely employed in industry and research, only a little part of the reported literature has addressed its use in the integration testing phase so far, and most of them addressed only test case generation instead of test case execution. Our approach presents how to generate test cases and how to automate test case execution. It could be applied in the industry.

Object-oriented program has some unique features, like inheritance and polymorphism. Our test approach does not address a specific feature of object-oriented programs. We proposed an integration testing approach addressing the automation of test case generation and test execution. It would be better to work with other testing techniques [41, 46] addressing these object-oriented features.

10.2 Future Work

- The tool we developed for the testing reads each message in the sequence diagram from top to bottom and generate relevant contracts automatically. The combined fragment notion elements in the sequence diagram, however, break up the sequence of messages by employing decision making, looping, and branching, enabling you to conditionally execute a particular path. The decision-making statements (if-then, if-then-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Java programming

language. While our TOOL has only implemented *if-then*, *if-then-else* for decision making statements, *while* for looping statement. One of our near future work is to make our tool recognize every statement supported by Java programming language.

- We divide elements in UML Sequence Diagrams into basic and advanced notion elements. The basic notion elements, including lifelines, messages and combined fragments, depict most interactions taking place in a common system. Our approach to generate test cases at the integration level has considered all cases of the sequence diagram consisting of the basic notion elements. One of the future work could be extending our approach to a more complex sequence diagram consisting of the advanced notion elements.

- The current version of the CDE does not support coordinating classes of objects for which the source code is not available, for instance .class files or Java library classes. We can only coordinate components for which the source code is available. Therefore, for instance, we may not define contracts that superpose behavior on operations that belong to a Java class library. Without this feature of the CDE, the integration testing is limited. It is promising but uncertain that the next version of CDE will support the coordination of components for which the source code is not available such as Java .class files by ATX.

Bibliography

[1] Aynur Abdurazik and Jeff Offutt. Using UML collaboration diagrams for static checking and test generation. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939, pages 383–395. Springer, 2000.

[2] Luís F. Andrade, José L. Fiadeiro, Joao Gouveia, and Georgios Koutsoukos. Separating computation, coordination and configuration. *Journal of Software Maintenance*, 14(5):353–369, 2002.

[3] SA ATX Software. http://www.atxsoftware.com/.

[4] SA ATX Software. *CDE Documents.* http://www.atxsoftware.com/CDE.

[5] Ralph-Johan Back, Luigia Petre, and Ivan Porres. Analyzing UML Use Cases as Contracts. In *Proc. of UML'99 - Second International Conference on the Unified Modeling Language: Beyond the Standard*, number 1723 in Lecture Notes in Computer Science, pages 518–533, York, UK, 1999. Springer-Verlag.

[6] F. Basanieri and Antonia Bertolino. A Practical Approach to UML-based Derivation of Integration Tests. In *Proc. of the 4th International Quality Week Europe QWE2000*, pages –, 2000.

[7] Boris Beizer. *Software System Testing and Quality Assurance.* Van Nostrand Reinhold (March 1984), 1984.

[8] Boris Beizer. *Software Testing Techniques.* International Thomson Computer Press; 2nd edition (June 1990), 1990.

[9] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns and Tools.* Addison-Wesley Professional; 1st edition (October 28, 1999), 1999.

[10] A. S. Boujarwah, K. Saleh, and J. Al-Dallal. Testing java programs using dynamic data flow analysis. In *SAC '00: Proceedings of the 2000 ACM*

symposium on Applied computing, pages 725–727, New York, NY, USA, 2000. ACM Press.

[11] David E. Brumbaugh. *Object-oriented development: building CASE tools with C++*. John Wiley & Sons, Inc., 1994.

[12] Timothy Budd. *An Introduction to Object-Oriented Programming.* Addison Wesley; 3 edition (October 12, 2001), 2001.

[13] Kai H. Chang, Shih-Sung Liao, Stephen B. Seidman, and Richard Chapman. Testing object-oriented programs: from formal specification to test scenario generation. *Journal of Systems and Software*, 42(2):141–151, 1998.

[14] Huo Y. Chen. An Approach for Object-Oriented Cluster-Level Tests Based on UML. In *IEEE International Conference on Systems, Man and Cybernetics, 2003*, pages 1064–1068, 2003.

[15] Huo Y. Chen, T.H. TSE, and T.Y. Chen. Taccle: a methodology for object-oriented software testing at the class and cluster levels. *ACM Trans. Softw. Eng. Methodol.*, 10(1):56–109, 2001.

[16] James W. Cooper. *The Design Patterns Java Companion.* Addison-Wesley Desing Patterns Series, 1998.

[17] Juan P. Galeotti and Marcelo Frias. Dynalloy as a formal method for the analysis of java programs. In K. Sacha, editor, *Software Engineering Techniques: Design for Quality*, volume 227, pages 249–260. IFIP International Federation for Information Processing, Springer, 2006.

[18] Leonard Gallagher and Jeff Offutt. Integration Testing of Object-oriented Components Using FSMS: Theory and Experimental Details. Technical report.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series).* Addison-Wesley Professional; 1st edition (January 15, 1995), 1995.

[20] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510, New York, NY, USA, 1975. ACM Press.

[21] The Object Management Group. *Introduction to OMG's Unified Modeling Language.* http://www.omg.org/gettingstarted/what_is_uml.htm.

[22] The Object Management Group. *UML 2.0 draft specification, 2003.*

[23] The Object Management Group. *UML 2.0 Superstructure Final Adopted Specification.* http://www.omg.org/cgi-bin/doc?ptc/2003-08-02.

[24] Mary J. Harrold. Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, NY, USA, 2000. ACM Press.

[25] Jean Hartmann, Claudio Imoberdorf, and Michael Meisinger. Uml-based integration testing. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 60–70, New York, NY, USA, 2000. ACM Press.

[26] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 169–180, New York, NY, USA, 1990. ACM Press.

[27] J. Herrmann and Andreas Spillner. Kriterien für den Intergrationstest modularer Softwaresysteme. In *In Informatik zwischen Wissenschaft und Gesellschaft = Zur Erinnerung an reinhold Franck, H.-J. Kreowski(ed)*, pages 21–26, York, UK, 1999. Springer-Verlag, Heidelberg.

[28] Paul C. Jorgensen. MM-Path: A white-box approach to software integration testing, 1984.

[29] Hareton K.N. Leung and Lee White. A study of integration testing and software regression at the integration level. In *Proceedings of the Conference on Software Maintenance 1990*, pages 290–301, San diego, CA, 1990.

[30] Hareton K.N. Leung and Lee White. Insights into testing and regression testing global variables, 1990.

[31] Ursula Linnenkugel and Monika Müllerburg. Test data selection criteria for (software) integration testing. In *ISCI '90: Proceedings of the first international conference on systems integration on Systems integration '90*, pages 709–717, Piscataway, NJ, USA, 1990. IEEE Press.

[32] Wayne Liu and Paul Dasiewicz. The Event-Flow Technique for Selecting Test Cases for Object-Oriented Programs, 1997.

[33] Jacqueline A. McQuillan and James F. Power. A Survey of UML-Based Coverage Criteria for Software Testing. Technical report.

[34] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall 1988, 1988.

[35] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[36] Glenford J. Myers. *The art of software testing.* John Wiley & Sons, Inc., 1946.

[37] Glenford J. Myers. *Software Reliability: Principles and Practices.* Wiley; 1 edition (September 22, 1976), 1976.

[38] Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. *Art of Software Testing.* Wiley; 2 Rev Upd edition (June 21, 2004), 2004.

[39] Inc. No Magic. *MagicDraw UML.* http://www.magicdraw.com/.

[40] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723, pages 416–429. Springer, 1999.

[41] Alessandro Orso and Mauro Pezzè. Integration testing of procedural object-oriented languages with polymorphism. In *Proceedings of the 16th International Conference on Testing Computer Software: Future Trends in Testing (TCS 1999)*, Washington, D.C., USA, june 1999.

[42] Atanas Rountev, Scott Kagan, and Jason Sawin. *Coverage Criteria for Testing of Object Interactions in Sequence Diagrams.* Springer Berlin / Heidelberg, 2005.

[43] Graeme Smith and John Derrick. Specification, refinement and verification of concurrent systems-an integration of object-z and csp. *Form. Methods Syst. Des.*, 18(3):249–284, 2001.

[44] IBM® Rational®Software. *IBM Rational Software Modeler.* http://www-306.ibm.com/software/awdtools/modeler/swmodeler/.

[45] Andreas Spillner. Control flow and data flow oriented integration testing methods. *Software Testing, Verification, and Reliability*, (2):83–98, 1992.

[46] Siros Supavita and Taratip Suwannasart. Testing Polymorphic Interactions in UML Sequence Diagrams. In *Proceedings of the the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II - Volume 02*, pages 449–454, Washington, DC, 2005.

[47] Alessandro Orso Vincenzo Martena and Mauro Pezzè. Interclass testing of object oriented software. In *ICECCS '02: Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*, page 135, Washington, DC, USA, 2002. IEEE Computer Society.

[48] Linzhang Wang, Jiesong Yuan, Xiaofeng Yu, Jun Hu, Xuandong Li, and Guoliang Zheng. Generating Test Cases from UML Activity Diagram based on Gray-Box Method. In *Proceedings of the the 11th Asia-Pacific Software Engineering Conference (APSEC'04) - Volume 00*, pages 284–291, Washington, DC, 2004.

[49] The Free Encyclopedia Wikipedia. http://en.wikipedia.org/wiki/Formal_verification.

[50] Hoijin Yoon, Byoungju Choi, and Jin-Ok Jeon. Mutation-based Interclass Testing. In *Proceedings of the Software Enginnering Conference, 1998 Asia Pacific*, pages 174–181, Taipei, Taiwan, 1998.

[51] Hong Zhu, Patrick A.V. Hall, and John H.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

Appendix A

Contracts for Testing "transfterTo(ca, amount)"

transferTo_MCSTest.ctr

```
contract transferTo_MCSTest
participants
  savingaccount:SavingAccount;
  checkingaccount:CheckingAccount;
attributes
  boolean result = false;
  int step = 0;
coordination
  CheckStep1:
    when *->> savingaccount.transferTo(chkAccount,amount) &&
    (checkingaccount == chkAccount)
    before {
      step = 1;
    };
  CheckStep2:
    when *->> savingaccount.withdraw(amount) && (step == 1)
    with(amount <= savingaccount.balance)
    before {
      step = 2;
    };
  CheckStep3:
    when *->> checkingaccount.deposit(amount) && (step == 2)
    before {
      step = 3;
    };
    after {
```

```
        if(step == 3) {
            result = true;
            System.out.println("the sequence of the method calls is correct!");
            step = 0;
            System.out.println("step is set to 0");
        }
    };
end contract //contract transferTo_MCSTest
```

transferTo_TSTest.ctr

```
contract transferTo_TSTest
participants
    savingaccount:SavingAccount;
    checkingaccount:CheckingAccount;
attributes
    boolean precondition = false;
    boolean result = true;
    CheckingAccount expected_chkAccount;
    int expected_amount;
coordination
    ParameterPrecondition:
        when *->> savingaccount.transferTo(chkAccount,amount) &&
            (checkingaccount == chkAccount)
        before{
            precondition = true;
            expected_chkAccount = chkAccount;
            expected_amount = amount;
        };
    ParameterTest_1:
        when *->> savingaccount.withdraw(amount) && (precondition)
        before{
            if(amount == expected_amount) {
                System.out.println("parameter amount test is passed.");
            }
            else {
                result = false;
                System.out.println("parameter amount test failed.");
            }
        };
    ParameterTest_2:
        when *->> checkingaccount.deposit(amount) && (precondition)
```

```
    before{
      if(amount == expected_amount) {
        System.out.println("parameter amount test is passed.");
      }
      else {
        result = false;
        System.out.println("parameter amount test failed.");
      }
    };
end contract //contract transferTo_TSTest
```

transferTo_RVTest.ctr

```
contract transferTo_RVTest
participants
  savingaccount:SavingAccount;
  checkingaccount:CheckingAccount;
attributes
  SavingAccount pre_savingaccount;
  CheckingAccount pre_checkingaccount;
  boolean result = false;
  boolean isEqual_Object1 = false;
  boolean isEqual_Object2 = false;
coordination
  ReturnValueTest:
    when *->> savingaccount.transferTo(chkAccount,amount) &&
      (checkingaccount == chkAccount)
    before{
      pre_savingaccount = new SavingAccount();
      pre_savingaccount.balance = savingaccount.balance;
      pre_savingaccount.accountNumber = savingaccount.accountNumber;
      pre_checkingaccount = new CheckingAccount();
      pre_checkingaccount.balance = checkingaccount.balance;
      pre_checkingaccount.accountNumber =
          checkingaccount.accountNumber;
    }
    after{
      if(amount ¡= savingaccount.balance) {
        pre_savingaccount.withdraw(amount);
        pre_checkingaccount.deposit(amount);
      }
      //compare two objects
```

```
if((pre_savingaccount.balance == savingaccount.balance) &&
   (pre_savingaccount.accountNumber ==
      savingaccount.accountNumber))
      {isEqual_Object1 = true;}
   if((pre_checkingaccount.balance == checkingaccount.balance) &&
      (pre_checkingaccount.accountNumber ==
      checkingaccount.accountNumber))
   {isEqual_Object2 = true;}
   if(isEqual_Object2 && isEqual_Object1)
      {result = true;}
   };
end contract //contract transferTo_RVTest
```

Appendix B

Contract One for Testing "cashCheck(check)"

cashCheck_MSCTest.ctr

```
contract cashCheck_MCSTest
participants
   bank:Bank;
   checkingaccount:CheckingAccount;
   accountledger:AccountLedger;
   check:Check;
attributes
   boolean result = false;
   int step = 0;
coordination
   CheckStep1:
      when *−>> bank.cashCheck(Check)
      before {
         step = 1;
      };
   CheckStep2:
      when *−>> check.getAmount() && (step == 1)
      before {
         step = 2;
      };
   CheckStep3:
      when *−>> check.getAccountNumber() && (step == 2)
      before {
         step = 3;
      };
   CheckStep4:
```

```
    when *->> accountledger.retrieveAccount(int) && (step == 3)
    before {
      step = 4;
    };
  CheckStep5:
    when *->> checkingaccount.getBalance() && (step == 4)
    before {
      step = 5;
    };
  CheckStep6:
    when *->> checkingaccount.addDebitTransaction(amount) &&
      (step == 5)
    with(balance >= amount)
    before {
      step = 6;
    };
  CheckStep7:
    when *->> checkingaccount.storePhotoOfCheck(Check) &&
      (step == 6)
    before {
      step = 7;
    };
  CheckStep8:
    when *->> checkingaccount.addInsufficientFundFee() &&
      (step == 5)
    with(!balance >= amount)
    before {
      step = 8;
    };
  CheckStep9:
    when *->> bank.returnCheck(Check) && (step == 8)
    before {
      step = 9;
    };
  StepResultCheck:
    when ? (step == 7 || step == 9) on bank, checkingaccount
    do {
      result = true;
      System.out.println("the sequence of the method calls is correct!");
    };
end contract //contract cashCheck_MCSTest
```