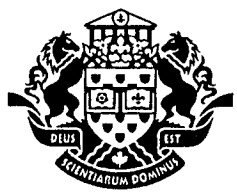

Université d'Ottawa · University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Pulei XIONG

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

M. Sc. (Systems Science)

GRADE - DEGREE

Systems Science

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

Life-Cycle E-Commerce Testing with Object-Oriented TTCN-3

R. Probert

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

L. Peyton

S. Some

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Life-Cycle E-Commerce Testing with Object-Oriented TTCN-3

Pulei Xiong

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of

Master of Science

Under the auspices of the System Science Program

University of Ottawa
Ottawa, Ontario, Canada
August 2004

Your file Votre référence
ISBN: 0-494-01649-3
Our file Notre référence
ISBN: 0-494-01649-3

Abstract

Web applications are becoming more and more popular in the past few years. Ensuring the quality of web applications is a critical issue in the development process. Testing web application is different from testing traditional software for the characteristics of web applications. While some traditional testing methods are still applicable, a few novel methods are proposed by researchers to deal with specific issues of web application testing. In this thesis, we propose *a life-cycle testing approach with Object-Oriented TTCN-3* for web applications.

First of all, the life-cycle testing approach is a systematic, model-based engineering process. In this process, testing activities take place from the analysis phase of software development process, to the implementation phase. That is, the testing process parallels the development process. The testing is based on *multiple models* at the analysis and design levels, as well as on implementations, by applying *multiple testing methods*.

Secondly, this life-cycle process is integrated by specifying test cases at an abstract level in OO-TTCN-3, an object-oriented extension to TTCN-3 which is a standard test language. The integration brings the following benefits:

i.) Facilitate the reuse of Abstract Test Suites (ATS) from one testing stage to another.

ii.) Provide a standard ATS interface which is language-independent and platform-independent. This helps to utilize *multiple testing tools* in software projects. Furthermore, a standard interface is expected to get more support from IT industry.

Finally, in this life-cycle process, several well-proven, commonly used cost-effective testing strategies and techniques are used to reduce the cost of life-cycle testing.

As a case study, we apply the approach to testing *EX system*, a typical e-commerce system. We build multiple models for the system, derive test cases from the models and source code, and specify ATS in OO-TTCN-3. It is concluded that the approach is feasible and efficient.

Acknowledgments

I would like to express my heartiest appreciation and gratitude to my supervisor Professor Robert L. Probert for his support, encouragement and financial help throughout my thesis research. I would like to thank Dr. Liam Peyton and Dr. Stephane Some for their valuable comments. I would also like to thank Dr. Os Monkewich and Mr. Bernard Stepien for their help during this research. I especially wish to thank Dr. Gustavo Crespo for his kindly help and encouragement.

My deepest love and respect to my parents, my brother and sisters for their encouragement to help me get through this.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Web technologies are evolving rapidly and so are the applications that are developed by using these technologies. Web applications which are based on web techniques have become ubiquitous. Most enterprises and government departments have adopted web applications as the basic information system architecture. Our daily life also increasingly relies on the software products running on the web. These make the quality of web applications a critical issue. However, the nature and complexity of web applications make writing bug-free code extremely difficult, even for highly experienced programmers. Software testing is imperative for building high quality web applications. It is an essential part of software development process.

In practical software projects, however, the effort on software testing is always constrained by limited resources such as time, budget and qualified testers. Complete testing is infeasible. The target of efficient and effective software testing is to implement testing in a cost-effective way to achieve high quality software products.

1.1 Cost-effective Software Testing

There are two well-proven efficient and effective testing strategies: life-cycle testing process [Kit95] [Bashir+99] [McGregor+01] and high-yield, risk-directed test case design strategy [Probert+00].

The idea of life-cycle testing is based on the facts that most errors in software can be traced back to incomplete, inaccurate or misunderstood requirement and design models, and finding/fixing the errors as early as possible is cost-effective [Kit95]. Therefore, the testing process should parallel the development process. That is, testing activities run from the analysis phase of software development process to the implementation phase.

Life-cycle testing is a systematic, model-based engineering process [Binder00]. The general life-cycle testing process is discussed in detail in section 2.3.4.1. A specific life-cycle testing process for object-oriented software is discussed in Chapter 3. The life-cycle testing process for web applications is similar to that of object-oriented systems since web applications are generally kinds of object-oriented

systems. However, there are some differences. For example, the levels of unit and integration testing correspond to the n-tiered architecture of web applications, functional testing may be based on navigation structure of web applications, and performance testing and security testing become more important for system level testing. The life-cycle testing process adapted to web applications is discussed in Chapter 6.

Life-cycle testing has been supported by several tools from IT industry. For example, IBM *Rational Software* provides a complete set of tools to support multiple development activities such as analysis, design, quality assurance, configuration management, process and project management [IBMRational]: RequisitePro, Data Modeler and XDE Modeler for requirements and analysis; XDE Developer, Technical Developer, Rapid Developer and Ada Developer for design and construction activities; Functional Tester, Robot and Performance Tester for automated system testing; PurifyPlus, XDE Developer Plus and Test RealTime for developer testing; ClearCase and ClearQuest for software configuration management; Rational Unified Process (RUP) and Team Unifying Platform for process and project management.

A problem in life-cycle testing is that the development and maintenance of test cases may be quite costly. A high-yield and risk-directed test case design strategy [Probert+00] which is discussed in detail in section 2.3.4.2, is quite effective to reduce the number of test cases but with same confidence. In addition, some commonly used testing techniques such as Equivalence-Classes Partitioning and Boundary-Value Analysis which are discussed in detail in section 2.3.5.2, are also quite effective to reduce the number of test cases.

In this thesis, we intend to apply these cost-effective testing strategies and techniques to web application testing.

1.2 Motivation and Statement of Research Problem

1.2.1 Definition of Terms

Electronic Commerce system and *Web Application* are two terms that are used frequently in this thesis. However, as far as we know, there are no formal definitions of these two terms. In this thesis, we do not intend to pursue precise description for

them. Instead, we only present informal definitions which are clear enough that readers will not feel confused about the content of the thesis.

An *Electronic Commerce (E-Commerce) system* is any software system which supports business that is conducted over the Internet. The business activities may use any of the applications that rely on the Internet, such as e-mail, instant messaging, shopping carts, Web services, UDDI, FTP, and EDI, among others [Webopedia04].

Web application, a term similar to e-commerce system, is also used in this thesis. Informally, any applications that use web technologies, including web browsers, web servers, and Internet protocols, to implement business logic can be classified as *web applications*. A definition given by Microsoft is as follows: A software program that uses HTTP for its core communication protocol and delivers web-based information to the user in the HTML language [Microsoft04].

In our opinion, we consider that "e-commerce system" is a term with a broader scope than "web application": an e-commerce system may consist of several relatively independent web applications. For example, an online bookstore system, which is considered as an e-commerce system, may include several web applications, e.g. order system, payment agent, and shipping center etc. The term of "e-commerce system" is rather loosely defined. In this thesis, we focus on the web application which is usually at the heart of an e-commerce system. We may use these two terms alternatively since we believe the discussion is applicable to both of them.

1.2.2 Web Application Testing: Related Work

Black-box and white-box testing approaches are commonly used in traditional software testing. In web application testing, they are still applicable. In addition, a grey-box testing approach is integral to the effective testing of web applications since web applications comprise numerous software and hardware components [Nguyen+03]. Grey-box testing incorporates elements of both black-box and white-box testing. It factors in design, environment, and interoperability conditions to reveal problems that are not as easily considered by black-box or white-box analysis, especially problems of end-to-end information flow and distributed hardware/software system configuration and compatibility [Nguyen+03].

Many research works have been done on web application testing.

In [Yang+99], the authors proposed an object-oriented architecture to construct web application testing environments. The architecture includes six subsystems: source document analysis subsystem, test management subsystem, test development subsystem, test execution subsystem, test failure analysis subsystem, and test measurement subsystem. The testing process can be achieved with the cooperation of these subsystems.

In [Kung+00A], the authors presented a web application testing methodology based on the Web Test Model (WTM). The WTM captures both structural and behavioral test related artifacts of web applications from three perspectives: from the object perspective, entities of a web application are represented using Object Relation Diagram (ORD) in terms of objects and inter-dependent relationships; from the behavior perspective, a Page Navigation Diagram (PND) is used to depict the navigation behavior of a web application, and a set of Object State Diagrams (OSDs) are employed to describe the state behavior of interacting objects; from the structural perspective, Block Branch Diagrams (BBDs) and Function Cluster Diagrams (FCDs) are used to capture the control flow and data flow information of scripts and functions in a web application. The structural testing of web applications was discussed in more detail in [Kung+00B]. In addition, special testing concerns can be expressed in terms of textural test constrains which can be imposed on the corresponding objects.

In [Ricca+01], the authors built a high-level model with extended UML for web applications. The model is the basis to the static verification and dynamic validation of web applications. The paper also defined white-box testing criteria for web applications, including page testing, hyperlink testing, definition-use testing, all-uses testing, and all-path testing.

In [Lucca+02], the authors proposed a testable object-oriented model for web applications, and discussed the level of unit testing for web application. Based on the model, the method of unit testing and integration testing is proposed.

In [Wen01], the author proposed a URL-Driven Automated Testing (URL-DAT) methodology for navigation testing of web applications, and demonstrated that the methodology can be a cost-effective, manageable and user-friendly.

In [Elbaum+03], the authors presented an innovative approach for web application testing. The approach differs from commonly used approaches in that it uses captured user behavior to generate test cases. The approach leads to a reduction in the amount

of required tester intervention while the effectiveness is comparable and likely complementary to more formal white-box testing approaches.

The testing approaches discussed above focus on testing web applications after they have been built. In [Probert+03], the authors presented a formal method which integrates testing work into the development process as early as at the system analysis and design phases. The paper demonstrated how to use the Specification and Description Language (SDL), Message Sequence Charts (MSCs), the Tree and Tabular Combined Notation (TTCN), and industrial-strength system design tools such as Telelogic TAU to develop and test a CORBA-based e-commerce system. This approach provides an effective means of improving e-commerce system quality.

1.2.3 Motivations

The testing approaches discussed above contributed to improve web application quality. However, they only apply to part of the development life-cycle: either at the analysis and design phases or the implementation phase. In addition, the test cases developed in a testing approach are written in a specific script language and depend on proprietary or specific industrial tools. These test scripts can not be reused by other testing approaches. In this thesis, we intend to propose a life-cycle testing approach which uses multiple testing methods in a life-cycle testing process. Furthermore, we integrate all the testing phases by specifying test cases on the abstract level with OO-TTCN-3, an object-oriented extension to TTCN-3 which is an international standard test specification and implementation language. These abstract test cases can be easily reused in the whole testing process, and are independent of specific testing tools.

1.2.4 Statement of Research Problem

To propose a life-cycle, integrated testing process for web applications based on a high-yield and risk-directed test case design strategy. Furthermore, to improve platform independence by specifying test cases at an abstract level in an extension of a standard test language, OO-TTCN-3, designed by us.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows:

Chapter 2: Describes life-cycle process of software development and basic concepts on software quality engineering. Then we discuss two well-proven cost-effective testing strategies: life-cycle testing process and high-yield, risk-directed test case design strategy. Finally we introduce the basic testing methods and techniques which are commonly used in a life-cycle testing process.

Chapter 3: Discusses in detail a life-cycle testing process for object-oriented software and the inheritance relationship between test cases for object-oriented software. This is the basis for the further discussion of the life-cycle testing approach for web applications, which are generally a kind of object-oriented systems.

Chapter 4: Introduces the web application architecture and techniques, and how to build models for web applications on multiple levels. The knowledge on web application architecture and techniques is essential to grey-box testing. The models facilitate the testing for web applications.

Chapter 5: Introduces the formal testing method and standard testing language TTCN-3. An object-oriented extension to TTCN-3 is proposed to make it capable of specifying object-oriented test cases. This extension makes TTCN-3 applicable to every phase of life-cycle testing for web applications.

Chapter 6: Proposes a life-cycle testing approach for web applications. The level of unit testing corresponds to the n-tier web application architecture. The whole process is integrated by specifying test cases at an abstract level in TTCN-3.

Chapter 7: Presents a case study to illustrate how to apply the life-cycle testing approach with TTCN-3 to a typical e-commerce system. The test case design is based on high-yield and risk-directed strategy.

Chapter 8: Gives conclusions and suggestions for future work.

Chapter 2 Background: Software Engineering, Software Quality Engineering and Software Testing

Life-cycle testing is a model-based engineering process which parallels the software development process. In this process, cost-effective test case design strategy and techniques may be used. In this chapter, we present the necessary background knowledge on software engineering and software quality engineering, and basic concepts of life-cycle testing process, test methods and test techniques.

2.1 Overview of Software Engineering

Software engineering is the process of solving customers' problems by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints [Lethbridge+01].

2.1.1 Software Development Lifecycle Model

The process used to create a software product from its initial conception to its public release is known as the *software development lifecycle model* [Patton01]. It helps developers to decide what work should be done and in what sequence to perform the work. There are many well-known models that can be used for developing software, e.g. build-and-fix model, waterfall model, rapid prototyping model, spiral model, and iterative and incremental model. These models should be seen as aids to thinking, not rigid prescriptions of the way to do things. No model is necessarily the best for a particular project. Among them, however, the iterative and incremental model is particularly well suited to object-oriented software development [McGregor+01].

2.1.1.1 "Pragmatic" IDP (Iterative and Incremental Model)

Under an iterative and incremental development process, a system is developed as a sequence of increments. An *increment* is a deliverable, including models, documentation, and code, which provides some of the functionality required for the

system [McGregor+01]. The products in one increment iteratively feed into the development of the next increment.

The *"Pragmatic" IDP* (Incremental Development Process), as shown in Figure 1, is an instance of the iterative and incremental model. In this process, software is developed incrementally, that is, from an initial prototype, to the final product. A product is released in four stages [Probert03] [Jacobson+99]:

- Release α: this release is for the purpose of collecting ideas and proof-of-concept, in other words, trying to understand what customers need.
- Release 1.0: the deliverable of this release is a prototype which will be installed on corporate hosting site (servers) for display
- Release 1.1: the deliverable is the product with minor fixes of Release 1.0 plus "must-have" functions that are not included in Release 1.0
- Release 2.0: the deliverable is the product to sell to a specific customer with intention to attract others

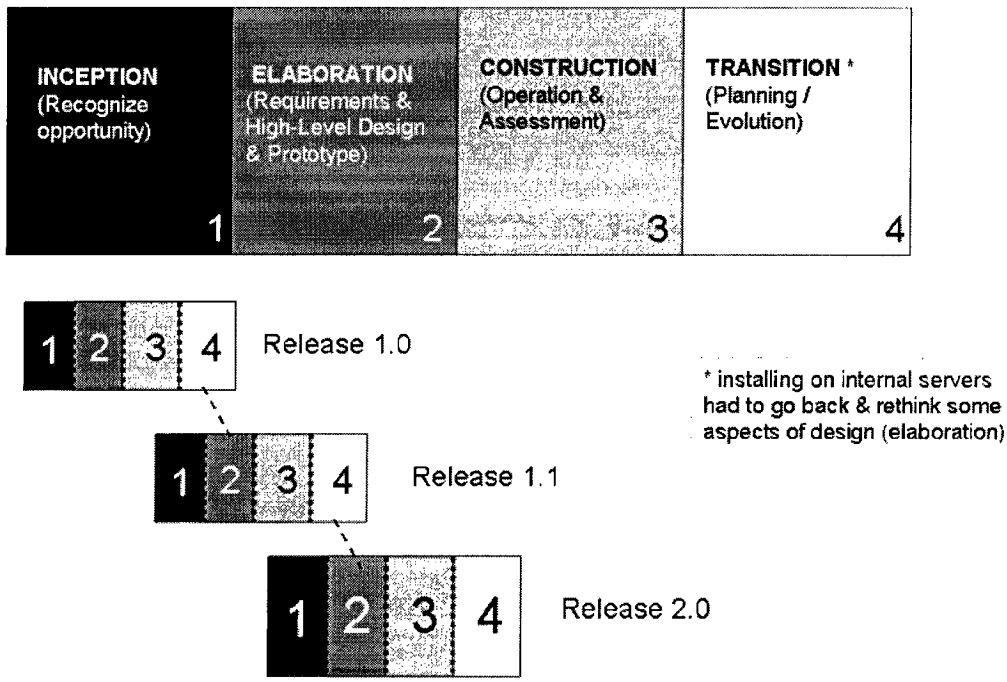

Figure 1 "Pragmatic" IDP

The " Pragmatic" IDP is considered as one of key engineering principles [Probert03] [Jacobson+99], which act as guidelines to produce software with improved time to market as well as high quality.

2.1.2 Software Development Process

The *software development process* is the way to incorporate the software life-cycle model, the tools, and most important of all, the individuals building the software [Schach02]. Generally, software development follows six phases: requirements, specification, design, implementation, integration, and maintenance. Testing is not a separate phase but an activity that takes place all the way through software development. There also is no separate documentation phase. It is important that each phase be fully documented before the next phase begins.

2.1.2.1 Requirements Phase

The real objective of the requirements phase is to determine what software customers need, but not what they want. The problem is that many customers do not know what they need. Furthermore, they may have difficulty in accurately conveying their ideas to developers.

One way of solving this communication-based problem is to build a rapid prototype. The role of the software quality assurance (SQA) group during the rapid prototyping phase is to verify with the customer that the final version of the rapid prototype is totally satisfactory.

2.1.2.2 Specification Phase

Specifications explicitly describes the functionality of software --- that is, precisely what the product is supposed to do --- and lists any constraints that the product must satisfy [Schach02]. Specification document must be informal enough for customers while formal enough for developers. In many development projects, the specification document is written in natural languages. This is the informal specification method. The specification document can also been developed by using structured system analysis methods, such as Entity-Relationship Diagrams (ERD) and Unified Modeling Language (UML). They are the semiformal method. Finite State Machines (FSM) and Petri Nets are the examples of formal methods.

In the specification phase, the SQA group must check the specifications carefully, looking for any omissions, contradictions, and ambiguities. In addition, SQA also must ensure the specifications are feasible and traceable. Specification inspection is an excellent way of checking the specification document.

2.1.2.3 Design Phase

The aim of the design phase is to determine how the product is to realize its requirements [Schach02]. The design phase is almost always broken down into architecture design and detailed design. During the architecture design, the designers decompose the product into modules. Once the architecture design has been done, the detailed design is performed. During the detailed design, algorithms and data structures are chosen for each module.

In the design phase, SQA group checks whether the design agrees with the specification document and whether every statement of the specification document is reflected in some part of the design. The types of faults to look for include logic faults, interface faults, lack of exception handling and most important, and nonconformance to the specifications.

2.1.2.4 Implementation Phase

Implementation is the process of translating the detailed design into code [Schach02]. The issues relating to the implementation phase include choice of programming language, good programming practice, practical coding standards, reuse of modules, and module testing etc.

A module should be tested while it is being implemented. After preliminary testing by programmers, the modules are handed over to SQA group. There are two basic types of methodical testing, *execution-based* (dynamic) testing, in which the module is run against test cases, and *non-execution-based* (static) testing, in which the module is reviewed by a team. Test cases for testing a module can be constructed systematically in two ways: *test to specification,* which is used to derive test cases only based on specifications; and *test to code,* which is used to derive test cases only based on code.

2.1.2.5 Integration Phase

During the integration phase, the modules are put together and tested as a whole. Although the integration phase can be treated as a separate, independent phase with the implementation phase, these two phases are usually carried out in a parallel way, called implementation and integration phase [Schach02]. Basically, there are three approaches of integration: top-down, bottom-up, and sandwich integration.

The purpose of integration testing is to check that the modules are combined together correctly to achieve a product that satisfies its specifications.

2.1.2.6 Maintenance Phase

Once a customer has accepted a product, any changes to the source code, documentation, manuals, or any other component of the product are performed in the maintenance phase. There are three types of maintenances: corrective maintenance to fix faults, perfective maintenance to extend the functionality of a product, and adaptive maintenance to react to changes in the environment in which a product operates.

It is unlikely that members of a maintenance team have been involved in the original development. Therefore, the maintainer tends to see the product as a set of loosely related modules. *Regression testing*, that is, testing the changed product against previous test cases to ensure that it still works correctly, is performed during the maintenance phase.

2.2 Software Quality Engineering

2.2.1 Introduction

Software quality engineering is a relatively new field of research and application. In the context of software quality engineering, software quality is measured and improved via metrics and models. There are four types of software quality models: reliability and projection models, quality management models, complexity metrics and models, and customer-oriented metrics, measurements, and models [Kan02]. The essence of software quality engineering is to investigate the relationships among these models and metrics, and, based on the findings, to engineer improvements in both end-product quality and process quality.

From popular views, the term *quality* is vague: it is some type of thing that can be discussed, felt, and judged, but can not be weighted and measured. The popular views do not help the quality improvement effort in industries. Quality must be defined in an operational way if improvement is to be achieved. Therefore, the definition of quality from professional views is constructed from two perspectives: "conformance to requirements" and "fitness for use". Based on the two perspectives, the de facto definition of quality consists of two levels: the intrinsic product quality (small q),

which is often limited to the product's defect rate and reliability, and a broader definition that includes product quality, process quality, and customer satisfaction (big Q) [Kan02].

Total Quality Management (TQM) or Total Quality Control (TQC) is a quality approach that aims at long-term success by linking quality and customer satisfaction. The TQM philosophy has been adopted by many companies and organizations in the form of product quality management and control strategies, such as Malcolm Baldrige National Quality Award (MBNQA), ISO 9000, Hewlett-Packard's TQC, Motorola's Six Sigma Strategy, and IBM's Market Driven Quality. A TQM system contains four key elements [Kan02]:

- Customer focus: to achieve total customer satisfaction by studying customers' wants and needs, gathering customers' requirements, and measuring and managing customers' satisfaction.
- Process improvement: to reduce process variations and to achieve continuous process improvement.
- Human side of quality: to create a companywide quality culture.
- Measurement and analysis: to drive continuous improvement in all quality attributes based on models and metrics.

The professional definition of quality fits perfectly in the TQM context, which correlates closely with the first two of the TQM elements. In the following two sections, we will discuss them in more detail.

2.2.2 Software Product Quality

Software product quality is commonly recognized as lack of "bugs" in the product. This definition is usually expressed in two ways: defect rate, e.g., number of defects per million lines of source code, per function point, or other unit; and reliability, e.g., number of failures per *n* hours of operation, mean time to failure, or the probability of failure-free operation in a specified time [Kan02]. Product quality can also be measured by customer satisfaction. In fact, customer satisfaction is the ultimate validation of quality. Besides overall customer satisfaction with software products, satisfaction toward specific attributes is also gauged. For example, IBM uses CUPRIMDSO, which stands for the quality attributes of capability (functionality),

usability, performance, reliability, installability, maintainability, documentation/information, service, and overall, to measure the degree of customer satisfaction with the products; while Hewlett-Packard focuses on FURPS which stands for functionality, usability, reliability, performance, and serviceability.

2.2.3 Software Process Quality

Software process quality is based on defect prevention process, process maturity framework and quality standards [Probert03].

The defect prevention process is aimed at development process. When integrated with the development process, it facilitates process maturity since it enables the process to fine-tune itself through the closed-loop learning process. The defect prevention process includes three simple steps [Kan02]:

- Analyze root causes of existing defects
- Suggest preventive actions to eliminate the defect root causes
- Implement defect prevention strategy

Several software development life-cycle models have been discussed in section [2.1.1]. Whereas certain models are better for certain types of projects under certain environments, the success of a project depends heavily on the implementation maturity, regardless of which model is used. In fact, in addition to development models, the frameworks to assess the process maturity of an organization or a project are important to the outcome of software projects [Kan02]. These include Software Engineering Institute's (SEI) *Capability Maturity Model* (CMM), Software Productivity Research's (SPR) assessment approach, the Malcolm Baldrige assessment process, and the ISO 9000 registration process.

2.3 Software Testing Basics

2.3.1 Introduction

Software development is an error-prone process. Quality software can't be created with an ad-hoc, part-time, bug hunt. It requires a methodical and disciplined approach to prevent, find, and report bugs. Software testing, which is an important quality assurance activity, confirms that a software system performs its intended functions

correctly. It includes a set of disciplined activities performed to measure software quality, such as functionality, usability, performance, and reliability.

Software testing is not equal to software Quality Assurance (QA). Software QA is responsible for verifying that project activities and products conform to the project's designated processes, procedures, standards, and requirements [Kan02]. Software QA has a much larger scope and responsibility than software testing, while software testing, in contrast to software QA, performs in-depth analysis, testing, and overall evaluation of a software product [Kit95].

Testing can be separated into two basic forms: *verification* and *validation*, which act complementarily. *Verification*, as defined in [IEEE/ANSI90], is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Verification answers the question of "are we building the product right?" Most syntax-directed quality assurance activities are verification activities, e.g., inspections of requirement specifications, design specifications, and code. *Validation*, as defined in [IEEE/ANSI90], is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies requirements. Validation answers the question of "are we building the right product?" Most functional testing activities are validation activities. Usually, verification is a static analysis process, while validation involves executing actual software or a simulated mock-up [Kit95].

2.3.2 A Systems Engineering View of Testing

Software testing can be considered as a problem in systems engineering, as shown in Figure 2 [Binder00]. It is a specific kind of software system that is designed and implemented for the validation and verification of another software system. Test design and implementation are best carried out in parallel with the analysis, design, and coding of the *System under Test* (SUT).

Test design involves several steps [Binder00]:

- Identify, model, and analyze the responsibilities of the SUT.
- Design test cases based on this external perspective.
- Add test cases based on code analysis, suspicions, and heuristics.

- Develop expected results for each test case or choose an approach to evaluate the pass/no pass status of each test case.

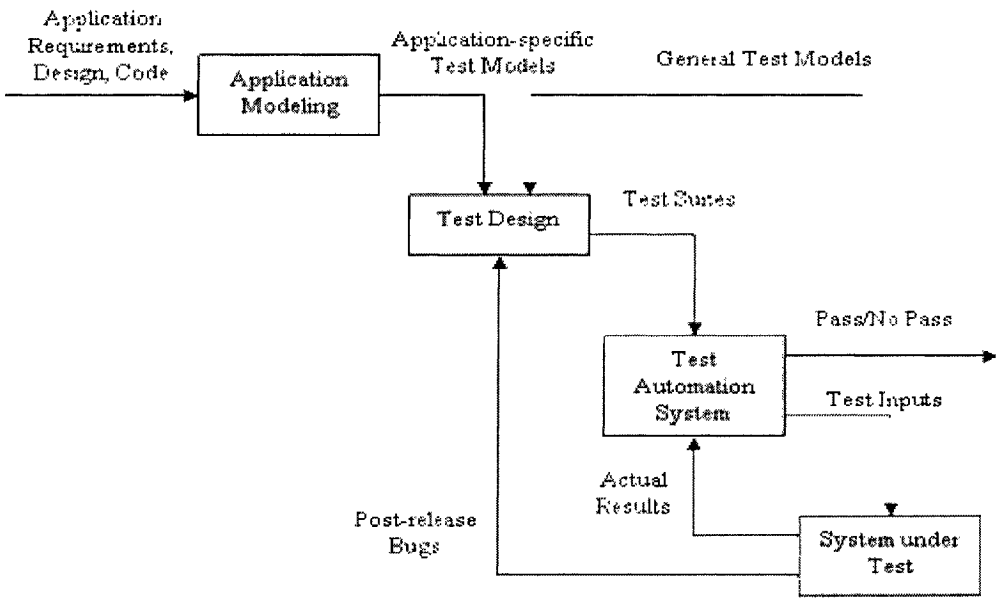

Figure 2 A Systems Engineering View of Testing

After test design is completed, the test suites are applied to the SUT. Although manual testing still plays a role, testing is mainly about the development of an automated system to implement an application-specific test design. It may be implemented with general-purpose test tools (scripting, coverage analyzers etc.), by coding application-specific test drivers and stubs, or by adding test code to the application. Test execution typically follows the following steps, although not all of them are always necessary or feasible [Binder00]:

- Exercise the interfaces between the parts of the Implementation under Test (IUT) to make sure the IUT is minimally operational.
- Execute the test suite; the result of each test is evaluated as pass or no pass.
- Use a coverage tool to instrument the IUT. Rerun the test suite and evaluate the reported coverage.
- If necessary, develop additional tests to exercise uncovered code.
- Stop testing when the coverage goal is met and all tests pass.

2.3.3 Model-based Testing

Effective testing should be a systematic, focused, and automated process [Binder00]: it must be systematic to ensure that every targeted combination is tried; it must be focused to take advantage of available information about where bugs are likely to be found; and it must be automated to produce and run the greatest number of consistent and repeatable tests.

Model-based testing can achieve all the three objectives using abstraction to conquer the astronomical complexity of typical software systems [Binder00]. A model abstracts the system under study by capturing only essential relationships. A model is easier to develop or analyze than the system under study.

Test design must be based on both general and specific models [Binder00]. General models, such as combinational logic models and state machines, offer a systematic and repeatable means to generate test cases. Application-specific test models, e.g. a testable OOA/D model, can be used to represent the required behavior of SUT. If there is no such model available, application-specific test models must be developed.

- Combinational logic models

  Many applications must evaluate output actions by selecting combinations of input conditions [Binder00]. Such requirements can appear at method, class, cluster, subsystem, and use case scope. Combinational logic models, such as decision table, decision table tree (activity diagram), logic function, and cause-effect graph, provide effective ways for representing these kinds of condition-action relationships.

  Test cases can be developed from a verified combinational model by using different strategies, such as All-Explicit variants, All-Variants, All-True, All-False, and All-Primes etc.

- State machines

  A state machine is a system whose output is determined by both current and past inputs [Binder00]. The effect of previous inputs is represented by a state which abstracts the information concerning past inputs that is needed to determine the behavior of the system on subsequent inputs. A state machine has four main

elements: state, transition, event, and action. It can be represented graphically, e.g. using state transition diagrams. It can also be represented in tabular format, e.g. state transition tables.

Object-oriented software is well suited to state-based testing, which may be applied at any scope. The Flattened Regular Expression (FREE) state model proposed in [Binder00] provides a testable model of class behavior. In the FREE model, the class under test is flattened to represent the behavior of inherited features to support the development of effective test cases. The FREE model can be expressed using *Unified Modeling Language* (UML), with some extensions.

- Application-specific models using the UML

  The UML is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as non-software systems [Booch+99]. It has become a de facto standard for object-oriented modeling. UML models, such as use cases, class diagrams, sequence diagrams, and statechart diagrams, are important sources of information for test design.

2.3.4 Cost-Effective Testing Strategies

Each product must be tested to ensure that the product is reliable enough, safe enough, and meets customers' entire requirements. However, it is always infeasible, if is not impossible, to test everything completely under the tight schedule of catching market. From a financial point of view, testing must be implemented in a cost-effective way. Complete life-cycle coverage [Kit95] [Bashir+99] based on a high-yield and risk-directed testing strategy [Probert03] is fundamental to deliver quality software on time within budget:

- Complete life-cycle coverage: testing should be a continuous activity throughout development cycle. It parallels the life-cycle of development process. It focuses on detecting errors as early as possible in order to prevent the costly migration of the errors downstream. In this life-cycle process, multiple test methods are applied to multiple models, and multiple testing tools maybe utilized.
- High-yield and risk-directed strategy: ensure to develop and perform adequate testing on the scenarios that have high possibility of detecting a defect (high-yield) and that will cause great loss (high-risk) if they fail.

2.3.4.1 Complete Life-cycle Coverage

There is strong evidence that errors are concentrated in the earlier stages of development process. As shown in Figure 3, more than half the errors are usually introduced in the requirement phase [Kit95]. Correction of these errors is less costly in the earlier stages of software development than in later stages. If the errors are detected only at a later stage of development, they entail costly rework. Therefore, detecting these errors in the same phase as they are introduced, and conducting an effective test process that prevents the migration of errors from a development phase to any subsequent phases, is deemed to minimize the cost of the errors.

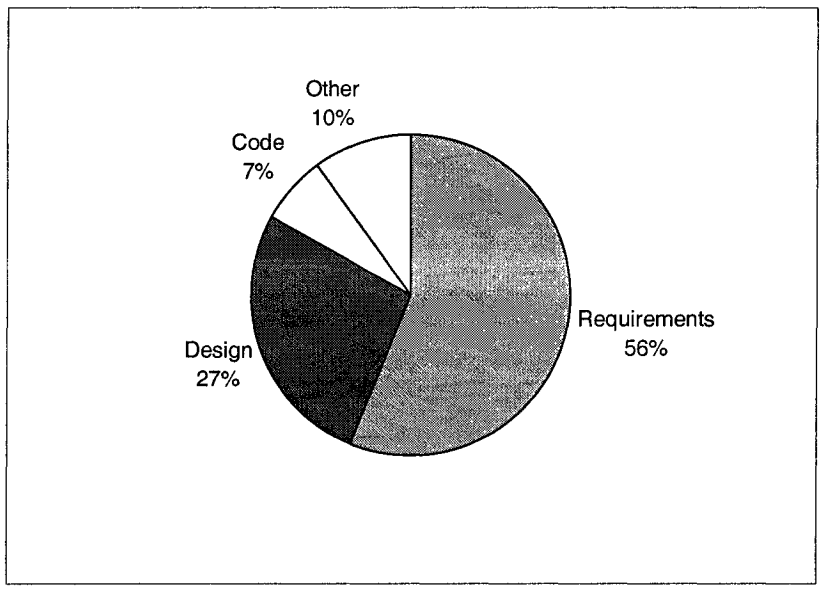

Figure 3 Defect Distribution

This means that only testing the end product is not a cost-effective way. Software testing has a life-cycle of its own. It is more than just a phase near the end of the development cycle. Testing processes should, as far as possible, be integrated at the most effective points in the development life cycle [Kit95]. In fact, early testing is a key to the success of the testing effort, and there are useful and constructive testing activities to be done throughout the entire life cycle of development. The testing process begins with the product requirements phase and from there parallels the entire development process. In other words, for each phase of the development process there is an important testing activity. In this life-cycle testing process, the term software means much more than just the code. The definition of software should include the

related documentation such as specifications, design documents, and user manuals [Kit95].

The steps for achieving complete life-cycle coverage are listed as follows [Probert03]:

- Set quality objectives (quality factors)
- Define quality metrics, including product metrics and process metrics
- Identify process certification points
- Identify products at each point
- Apply metrics at each point
- Make go/no go decision
- Assess process with quality objectives

2.3.4.2 High-yield and Risk-directed Strategy

In the real world of software projects, exhaustive testing is infeasible, and the testing of any program will be necessarily incomplete. The negative effects of this incompleteness can be minimized by identifying the subset of all possible test cases based on high-yield and risk-directed strategy [Probert03].

The term *yield* refers to the number of defects detected. Usually, normal scenarios are low-yield, and exceptional scenarios are high-yield [Probert03]. The term *risk* refers to the cost associated with a failure. Sometimes, the term of *combined risk* is used, which is the product of cost of a failure and the probability of the failure. Obviously, it is a cost-effective way to implement testing with test cases that has the highest probability of detecting the most errors and that have the most serious consequences if they fail. In this way the greatest possible number of errors can be found, and it is confident that all high-risk scenarios have been tested, with a finite number of tests.

2.3.5 Test Case Design Methods

2.3.5.1 Introduction

A *Test case* is a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement [OULU04]. The execution of a given test case will either achieve requirements coverage which aims to verify and validate

certain requirement, or logic coverage which aims to verify certain parts of a program's internal logic.

Accordingly, there are two basic approaches to develop test cases: *requirements-based approach* and *code-based approach.* A requirements-based approach is used to derive test cases only based on system requirements. It is also referred to black-box, functional, or input/output driven testing. A code-based approach is used to derive test cases only based on source code. Other names for this approach are white-box, glass-box, structural, logic-driven, and path-oriented testing.

Requirements-based approaches and code-based approaches are both important in software testing. A recommended strategy for integrating the two approaches to improve test efficiency is as follows [Kit95]:

- Methodically identify, design, and develop functional requirements-based test cases for function testing, and non functional requirements-based test cases for system testing, such as usability, performance, and security testing.
- Execute the test cases and measure their collective internal logic coverage.
- Using code-based methods identify and develop supplementary test cases as necessary to improve internal logic coverage.

Two other terms used to describe how software is tested are *static testing* and *dynamic testing*. Static testing refers to testing software without running it, but just examining and reviewing it. The examples of static testing include specification verification, code inspections, symbolic execution, and data flow anomaly analysis. Dynamic testing, by contraries, is the process of evaluating a system or component based upon its behavior during execution. In the following sections, we focus on dynamic testing.

2.3.5.2 Black-Box Testing

Black-box testing is a customer-directed testing method. In using this method, a tester views a program as a "black box". That is, the tester is completely unconcerned about the internal behavior and structure of the program. Rather, the tester is only interested in finding out in which circumstances the program does not behave according to its specifications [Myers79].

Black-box testing focuses on software's external attributes and behavior. It is used to derive test cases solely from requirement specification (both functional requirements and non-functional requirements), without taking advantage of the knowledge of the internal structure of a program. When test cases are derived from functional requirements, test design focuses on parameter limits, the availability of particular types of customer services, and "what-if" scenarios derived from normal customer interactions [Probert03].

Black-box testing is usually applied to functional testing and system level testing, such as performance testing, acceptance testing, and usability testing.

There are well-proven black-box test design methods that provide an intelligent and effective means of identifying high-yield test cases, such as equivalence-class partitioning, boundary-value analysis, state-based testing, cause-effect graphing (CEG), and error guessing. Each of these methods has strengths and weaknesses, that is, it is likely to detect some specific types of errors while it will fail to detect other errors. Among them, equivalence-class partitioning and boundary-value analysis are most commonly used.

2.3.5.2.1 Equivalence-Class Partitioning

Equivalence-class partitioning is a systematic process that identifies a set of interesting classes of input conditions to be tested, where each class is representative of (or covers) a large set of other possible tests [Kit95]. It can significantly reduce the number of test cases required to cover the input conditions, but it does not test their combinations.

There are two distinct steps to develop test cases by using the equivalence-class partitioning [Probert03]:

1. Identify equivalence classes
   - Identify input (external import) conditions which may occur.
   - For each condition, identifies normal behaviors (valid equivalence class) by associated parameter ranges, and derives exceptional behaviors (invalid equivalence class) from normal behaviors.
   - List all valid and invalid equivalence classes. Assign a unique number to each equivalence class.
2. Identify test cases

Identify test cases to cover the input equivalence classes so that normal cases (valid) are broadly covered and that exceptional cases (invalid) are covered in depth:

- Each new test case covers as many currently uncovered valid equivalence classes as possible.
- Each additional test case covers no more than one invalid equivalence class.

2.3.5.2.2 Boundary-Value Analysis

Boundaries are always a good place to look for defects. Boundary-value analysis is a variant and refinement of equivalence-class partitioning. It enhances yield over equivalence-class partitioning by two major differences [Probert03]:

- Boundary-value analysis focuses on "boundaries" of equivalence classes. Elements are selected just on or just beyond the borders of each equivalence class.
- Boundary-value analysis considers both input and output domains of use. In addition to the input space, test cases can be derived by considering output equivalence classes as well.

2.3.5.3 White-Box Testing

White-box testing permits one to examine the internal structure of a program to derive test cases. It requires knowledge of a program's internal structure, such as internal data structures, physical logic flow, and architecture at the source code level. Sometimes, specification is still needed in white-box testing to know expected output.

White-box testing is an important approach to improve test case coverage for two reasons [Kit95]:

- It indirectly improves function coverage.
- It is necessary for the testing of logic paths that are not discernible from the external functionality. For example, a math function may use completely different algorithms depending on the value of the input arguments. It can not be tested adequately from an external view.

However, white-box testing can not detect missing functions that are described in functional design specification but not implemented. It is usually applied to low level testing, such as unit testing.

Depending on the testing objective, there are different white-box testing methods. These methods ensure the internal parts of a program are being adequately tested with sufficient logic coverage, including code coverage and data coverage. The methods aiming to code coverage include statement coverage, branch coverage, condition coverage, and path coverage. The methods aiming to data coverage include All-Defs, All-Uses, All-p-Uses/Some-c-Uses, All-c-Uses/Some-p-Uses, and All-du-paths coverage.

2.3.5.4 Grey-Box Testing

Both black-box and with-box testing are critically important components of a complete testing effort. However, they have some limitations. For example, black-box testing is less effective at uncovering certain error types, such as data-flow errors or boundary condition errors at the source code level, while white-box testing does not readily highlight macro-level quality risks in operating environment, compatibility, time-related errors, and usability [Nguyen+03].

Grey-box testing, which incorporates elements of both black-box and white-box testing, is a system-directed approach that utilizes the knowledge of system design logic to improve the probability of finding bugs [Nguyen+03]. Grey-box testing design is based on the knowledge of algorithms, internal states, architectures, and other high-level descriptions of system behaviors. Outcome on the user end, system-specific technical knowledge, and operating environment are all taken into account in the grey-box test design.

Grey-box testing focuses on the interoperability of system components. It is usually applied to unit and integration testing, and is well suited for web application testing since web applications comprise numerous software and hardware components [Nguyen+03].

2.4 Summary

Testing is imperative for building high quality software products. In this chapter, we provided background information on software engineering and software quality engineering, as well as basic concepts, methods and techniques of software testing. We also introduced well-proven cost-effective testing strategies, life-cycle testing process and a high-yield and risk-directed test case design method.

Chapter 3 Testing Object-Oriented Software

Today, object-oriented technology has been dominant in IT industry for its unique mechanisms of inheritance, polymorphism and encapsulation. These object-oriented mechanisms are good for dealing with large scale, complex systems. However, they also make the testing of object-oriented software quite different from that of traditional procedure-oriented software.

In this chapter, we present a brief overview of object orientation paradigm; discuss the impact of object-oriented technology on software testing; and discuss the life-cycle testing process for object-oriented software. In addition, we also discuss the inheritance relationship between test cases derived from class inheritance hierarchy.

3.1 Overview of Object Orientation Paradigm

3.1.1 Object-Oriented Paradigm versus Procedural Paradigm

In the earliest days of programming, software was organized around procedures (also called functions or routines). These provide *procedural abstraction* which helps to simplify programmers' view of a system. *Data abstractions*, e.g. records and structures which group together the pieces of data that describe some entity as a unit, is also utilized to help reduce some of a system's complexity. The so-called *procedural paradigm* works very well when software systems are in small or middle scale.

The *object-oriented paradigm*, which is an approach to the solution of problems in which all computations are performed in the context of objects [Lethbridge+01], utilizes both procedural abstraction and data abstractions. By 1990s, the object-oriented paradigm had become accepted as the best way to organize most systems.

In the procedural paradigm, a software system is organized into a set of procedures; whereas in the object-oriented paradigm, a running software system consists of a collection of objects which collaborate to perform a given task.

3.1.2 Basic Concepts of Object-Oriented Paradigm

The following are the essential concepts that distinguish an object-oriented language or system from one that is not object-oriented.

- Classes, Objects, and Relationships

An *object* is a chunk of structured data in a running software system [Lethbridge+01]. Usually, an object consists of two parts: properties and behavior. *Properties* describe the current state of an object, and *behavior* is the way an object acts or reacts.

Class is a unit of abstraction in an object-oriented system. A *class* is a software module that represents and defines a set of similar objects with same properties and behavior [Lethbridge+01]. It contains all of the code that relates to its objects, including code describing how the objects of the class are structured and code of the methods that implement the behavior of the objects.

An object might be naturally related to other objects. There are two basic types of object relationships: association and aggregation. An *association* is used to show how two classes related to each other [Lethbridge+01]. Symbols of *multiplicity*, which are shown at each end of an association, are used to indicate how many instances of a class can be linked to an instance of another class. *Aggregations* are special associations that represent "part-whole" relationships [ibid]. A *composition* is a strong kind of aggregation in which if the aggregation is destroyed, then the parts are destroyed as well [ibid].

- Instance Variables

A variable is a place used by a software system to store data. *Instance variables* are those defined inside a class corresponding to data present in each instance, which are either used to implement attributes or used to implement associations [Lethbridge+01]. An *attribute* is a simple piece of data used to represent the properties of an object [ibid].

- Methods, Operations, and Message

An *operation* is a higher-level procedural abstraction that specifies a type of behavior, independently of any code implementing that behavior [Lethbridge+01]. A *method*, on the contrary, is a procedural abstraction used to implement the behavior of a class [ibid]. Several different classes can have methods with the same name that implement the same abstract operation in ways suitable to each class.

A *message* is a request that an operation be performed by some objects [McGregor+01]. Running of an object-oriented system is based on the collaboration between each object. This collaboration is achieved by sending messages to one another.

- Polymorphism

Polymorphism is a property of object-oriented software by which an abstract operation may be performed in different ways in different classes [Lethbridge+01]. Every time an operation is called, a running program decides which of several identically named methods to invoke by identifying the class of the object in a particular variable. Polymorphism is one of the fundamental features of the object-oriented paradigm.

There are two types of polymorphism: inclusion polymorphism and parametric polymorphism [McGregor+01]. *Inclusion polymorphism* is the occurrence of different forms in the same class [ibid], which is also referred as *dynamic binding*. *Parametric polymorphism* is the capability to define a type (*generic class*) in terms of one or more parameters [ibid].

- Inheritance

Inheritance is the implicit possession by all subclasses of features, which include variables and methods, defined in their superclasses [Lethbridge+01]. Inheritance is essential in the object-oriented paradigm. It supports efficient reusability and extensibility. An *Inheritance hierarchy* formed from one superclass with several immediate subclasses is commonly called a *generalization*, which shows the *isa* relationships among the superclass and the subclasses. In some case, instead of creating a superclass, it might be better to create an *interface*, which is an aggregation of behavioral declarations. Organizing classes into inheritance hierarchies is a key step in object-oriented design and programming.

Much of the power of the object-oriented paradigm comes from polymorphism and inheritance working together [Lethbridge+01]. Concepts, such as abstract class (abstract method), overriding, immutable object, and dynamic binding etc, are the products of the combination of polymorphism and inheritance.

- Encapsulation

Encapsulation means that several items are packaged together into one unit. A class is such a unit that acts as a container to hold its features (variables and methods)

and defines an interface that allows only some of them visible from outside [Lethbridge+01]. This gives rise to *information hiding* which simplifies the overall system since designers and programmers need to know far fewer than they otherwise might.

3.1.3 Benefits of Object-Oriented Approach

The object-oriented approach addresses the software development problems with quality, productivity, and maintainability.

First, the object-oriented approach helps to build quality software. Object-oriented analysis and design is an incremental and iterative process. A core set of models are utilized throughout analysis and design [Benett+02]. In the UML, the fundamental analysis models are the use case and the class diagram, which map straightforwardly to design models, in turn, map to code. Other design models are derived from them by adding more detail at each stage. This way of smooth model transitions avoids missing or incorrect specifications. Additionally, object-oriented software is constructed based on objects. Each object in a system is relatively small, self-contained, and manageable [Satzinger+01]. This reduces the overall complexity of the software. Less complexity means fewer logical and syntactical errors.

Second, one of the biggest benefits from objects is the reuse of program code and analysis results [Brown02]. This is due to the highly modular nature of object-oriented software. In an object-oriented system, a class provides only its interface to other classes. Its internal structure and logic do not interfere with the design and implementation of other classes. This follows that each part of an object-oriented system can be constructed largely independent of other parts, which is what modularity means. Encapsulation, inheritance, and polymorphism are the key mechanisms leading to this smooth and efficient way of reuse. Reuse can greatly increase productivity, while it also results in improved quality because the reused objects are proven products.

Finally, the object-oriented approach helps to improve system maintainability [Satzinger+01]. Because of the self-contained nature of objects, methods and attributes can be added to existing classes without disrupting the rest of the system. New classes can also be added easily by exploiting the inheritance feature with little programming effort.

Therefore, the object-oriented approach makes it possible to build quality software faster in a way that facilitates enhancements and change.

3.2 Impact of Object Orientation on Testing

While the object-oriented paradigm features improve the quality and management of software development, they obviously impact some aspects of testing. The conventional software testing methods are not adequate for object-oriented systems. Object-oriented software testing has to deal with new problems introduced by the powerful new features of the object-oriented paradigm, such as encapsulation, inheritance, and polymorphism etc. Changes in object-oriented software development process and changes in the focus of analysis and design also affect testing. Additionally, object-oriented software metrics are needed for the purpose of measuring and improving software quality and project management.

3.2.1 Side Effects of the Paradigm

3.2.1.1 Encapsulation

Encapsulation helps to prevent problems with global data access which is common to procedural program. Encapsulation does not directly incur the occurrence of bugs, but it presents an obstacle to testing [Binder00]. Encapsulation supports information hiding which makes part of an object inaccessible to the outside, therefore makes it difficult to directly set or get the concrete state of an object. Although *accessor methods*, which provide information about an object, and *modifier methods*, which change the state of an object by setting one or more attributes to have new values, can be created additionally and used to set and get the state of an object, these methods themselves need to be validated first.

3.2.1.2 Inheritance

Inheritance weakens encapsulation, creating problems similar to global data in procedural languages. Besides, inheritance may be abused in many ways, and this brings bug hazards [Binder00].

- A deeply inheritance hierarchy makes it difficult to understand source code.
- Inheritance is used to implement generalization/specialization relationships. However, in many cases, inheritance may easily be used as a programming

convenience, independent of any relationship between superclass and subclass. This kind of irregular type hierarchies can lead to incorrect polymorphic message bindings.

- Multiple inheritance mechanism allows a class inherits from two or more parent classes. In the case that parent classes contain features with the same names, incorrect binding and unanticipated interaction among inherited features may occur. In addition, repeated inheritance occurs when a common superclass appears in a multiple inheritance hierarchy.
- Abstract and generic classes provide important support for reuse. However, testing abstract and generic classes may be complicated because an instantiation of them must be constructed for the testing purpose. In addition, generic class, which accepts a type parameter(s) for substitution, is an unbound type declaration which easily leads to bugs.

3.2.1.3 Polymorphism

Polymorphism can greatly simplify client interface to polymorphic servers and eases maintenance and reuse. However, polymorphism and dynamic binding dramatically increase the number of program execution paths. Therefore, static analysis of source code to identify paths is of little use. Problems caused by polymorphism include [Binder00]:

- Polymorphic, dynamically bound messages can result in hard-to-understand, error-prone code.
- Changes can be made to a polymorphic server without regard to clients. Even a little change may cause a client to fail.
- The actual behavior of polymorphic messages are quite complex. It is determined by many variables not visible in the client's source code. The yo-yo problem is a good example to illustrate this problem.

3.2.1.4 Message Sequence and State-related Bugs

A *state* of an object is defined as the combination of the attribute values of the object [Binder00]. In object-oriented programming, objects preserve state. But state control, which is implemented by message sequences, is typically distributed over an entire system. State control errors are likely, and this makes object state testing an important aspect of object-oriented software testing. Object state testing is different

from the conventional control flow testing and data flow testing [Kung+98]. It focuses on testing the state-dependent behaviors of objects, while in control flow testing the focus is testing program according to the control structures, that is, sequencing, branching, and iteration of the program being tested; and in data flow testing the focus is testing the correctness of individual data define-and-use [ibid].

3.2.1.5 Testing Axioms

Adequate testing of object-oriented software is quite important. According to [Binder00], there are three axioms for object-oriented software testing:

- Similar functions, for example, an overridden method in a subclass and the method in a superclass, may require different tests to achieve coverage because a test suite that is adequate for one implementation of a specification is not necessarily adequate for a different implementation of the same specification (antiextensionality axiom).
- Adequate testing of a client does not necessarily result in adequate testing of its server because all of the server's methods maybe can't be called by clients (antidecomposition axiom).
- Different contexts of usage, for example, a method in a subclass and this method in a superclass, may require different tests to achieve coverage because test suites that are individually adequate for subroutines are not necessarily collectively adequate for a system that calls these subroutines (anticomposition axiom).

3.2.2 Object-Oriented Software Testing Process

The object-oriented software development process is different from its procedural counterpart. It promotes an iterative and incremental development life cycle. Similar to the development life cycle, the object-oriented software testing is also an iterative and incremental process, called *parallel testing life cycle* (see Figure 4) [Bashir+99]. The parallel testing process includes six stages: requirements testing, design testing, unit testing, integration testing, integrated system testing, and acceptance testing.

The most significant impact on the testing of object-oriented software is the shift in focus to unit and integration testing strategies [Bashir+99]. For object-oriented software, the unit of testing is not a subroutine or a function any more. Instead, a class, as the basic building block for constructing object-oriented software, is the most

natural unit of testing. Unit testing of object-oriented software focuses on the methods associated with an object and their behaviors depends on the state of the object. Inheritance relationship is another concern in unit testing. The foremost task in subclass testing is the identification of methods that mandate retesting as well as those that can safely be used without retesting.

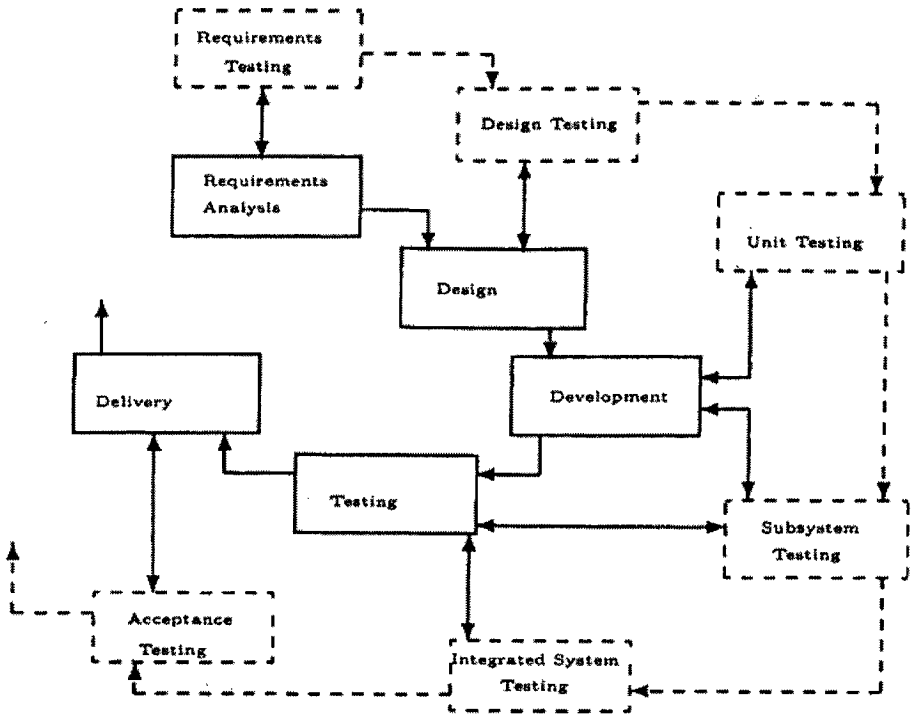

Figure 4 Parallel Testing Life Cycle for Object-Oriented Software

The change in unit testing impacts integration testing. In integration testing, the unit of integration is a class, instead of a sub-program in procedural programming. Classes can be combined into components, or they can be aggregated to form composite classes. The primary objective of integration testing is to catch any errors when two or more individually developed components are combined to execute their functionalities [Beizer90]. According to [Bashir+99], the set of possible integration levels includes:

- Integration of members into a single class
- Integration of two or more classes through inheritance
- Integration of two or more classes through containment
- Integration of two or more classes to form a component
- Integration of components into a single application

Integration testing strategies for each level will be addressed in the following sections, including class level integration testing in [3.3.2], component level in [3.3.3], and system level in [3.3.4].

3.3 Life-Cycle Testing for Object-Oriented Software

Object-oriented software testing is an iterative and incremental process. In every iteration, the testing process includes six stages [3.2.2]. In each phase in the development process, testing is the last activity to verify the work produced in the phase possesses the desired quality. The work is in the form of either a UML model or code in programming languages. In the second and succeeding iterations, new test cases which cover added features, together with failed test cases in the last iteration and those covering areas most likely to fail, are executed.

3.3.1 Testing Analysis and Design Models

Guided inspection is an enhanced inspection technique for verifying the models as they are created and for validating the completed models against project requirements [McGregor+01]. The primary models in the system analysis and design phases include use case diagrams, class diagrams and class specification, state diagrams, sequence diagrams, and activity diagrams. Guided inspection provides a means of objectively and systematically applying testing perspective to these models by using explicit test cases. This helps to find and fix faults in the system early in the development process. Therefore, guided inspection can result in big savings.

The object-oriented analysis and design is an incremental process where each development phase moves the product a step toward the final system. The model produced in one development phase is more specific (lower abstraction) and more complete (wider scope of the content) than the model from the previous phase. Accordingly, the inspection of each succeeding model can also become more specific and narrowly focused while the information in the model becomes more specific. This sequence of models which are transformed incrementally provides an opportunity to establish a "chain of quality", in which each model is verified before moving to the next phase [McGregor+01].

3.3.1.1 The Basics of Guided Inspection

3.3.1.1.1 Guided Inspection Steps

According to [McGregor+01], the basic guided inspection steps are as follows:

- Specifying the scope and depth of the inspection. The scope of an inspection is defined by specifying a set of use cases, a set of packages, or abstract classes/interfaces. The depth of the inspection is defined by describing the level of detail, e.g. layers in aggregation hierarchies, to be covered.
- Identify the basis from which the Model under Test (MUT) was created. The basis for all but the initial model is the set of models from the previous development phase, while the initial model is from the knowledge of domain experts.
- Develop test cases for each of the evaluation criteria to be applied by using the contents of the basis model, e.g. use case model, as input. Test cases for a guided inspection are scenarios that should be represented in the MUT. Several techniques, such as equivalence classes, combinational logic, orthogonal defect classification, user profile, and risk analysis, can be used to select test cases to ensure that specific types of coverage are achieved, or to find specific types of defects.
- Establish criteria for measuring test coverage, which will be discussed in detail in section [3.3.1.1.2].
- Perform the static analysis using the appropriate checklist. The purpose of this step is not concerned with the content but only the form of the model. For example, examine the models for certain syntactic information.
- "Execute" the test cases. This step of the guided inspection is performed in one of two ways, depending upon whether the model has been automated or not. If a prototype or other working model has been created, the test cases will be implemented, usually in some scripting language, and executed using the simulation facilities of the prototype of the model. If the model has not been prototyped, a symbolic execution of the test cases will simulate the system behaviors that will occur when actual code is available.
- Evaluate the effectiveness of the tests using the coverage measurement. Expand the test suite and apply additional tests if the coverage is insufficient, otherwise terminate the testing.

3.3.1.1.2 Coverage in Models

In the UML models, the usual object-oriented concepts: classes, relationships, objects, and messages are the model elements that are used to measure test coverage. A test case "covers" one of these elements if it uses that element as part of a test case [McGregor+01].

A particular element does not need to exhaust all possible values of the attributes of that element, depending on the level of MUT. In the object-oriented incremental development process, as the detail of models increases, the detail at which coverage matters increases as well. For a domain analysis model, a test case containing a single object from a class can be considered sufficient to cover the class. Coverage for this level of model can be stated as a percentage of classes and relationships covered [McGregor+01]. At the design level, a class can be said to be covered by the test cases that use every method in the interface of the class. Coverage for this level is more likely to be stated by counting all of the methods in the model [ibid].

Defects in an abstract class will propagate to all of the concrete classes that eventually are derived from the abstract class. For this reason, the more abstract the classes, the higher the level of coverage that should be required [McGregor+01]. Likewise, higher level of coverage is required for the models that are in higher level.

3.3.1.1.3 Evaluation Criteria

Guided inspection is performed with respect to a set of evaluation criteria: correctness, completeness, consistency, and additional qualities. The concrete content of criteria for each model will be discussed in the next section: [3.3.1.2].

- Correctness is a measure of the accuracy of the model [McGregor+01]. If the execution of a test case produces the expected result, the model is correct with respect to this test case.
- Completeness is a measure of the inclusiveness of the model [McGregor+01]. Test cases are executed to check if there are scenarios that the elements in the model can not represent. In an incremental process, completeness is considered relative to the expected maturity of the current increment.
- Consistency is a measure of whether there are contradictions within the model or between the current model and the model upon which it is based [McGregor+01]. Inconsistencies can be identified by finding different representations within the model for similar test cases, or may be identified by

execution of a test case when the current MUT is compared to its basis or when two diagrams in the same model are compared.

- Additional qualities define a number of system attributes, e.g. performance and flexibility, that the development team might wish to verify [McGregor+01].

3.3.1.2 Testing Specific Types of Models

The basic guided inspection technique can be applied throughout the analysis and design phases, with respect to each specific model.

3.3.1.2.1 Requirements Model

The usual requirements model for object-oriented software is use case. According to [McGregor+01], evaluation criteria for requirements inspection are as follows:

- Completeness: The use cases represent all of the functionality needed for a satisfactory product. No use case is included that is not a required functionality.
- Correctness: Each use case accurately represents a requirement.
- Consistency: Any system functionality is specified in the same manner everywhere it is described.

The guided inspection can identify many faults that result from errors with the requirements, which otherwise may not be captured until the acceptance testing stage. The typical errors include missing requirements (an incomplete requirements model), requirement that contradict each other (an inconsistent model), and scenarios in which the system does not behave as the client intended (an incorrect model) [McGregor+01].

3.3.1.2.2 Analysis Models

There are two types of analysis models: domain analysis models which model the knowledge in a domain; application analysis models which model knowledge about the product.

Although sometimes a domain model is only a simple class diagram, most domain models encompass standard algorithms and many refer to states that are characteristic of the concepts being represented. According to [McGregor+01], evaluation criteria for a domain model are as follows:

- Completeness: The concepts are sufficient to cover the scope of the content specified. Sufficient detail is given to describe concepts to the required depth.
- Correctness: The descriptions of domain concepts are accurate; the algorithms will produce the expected results.
- Consistency: Model elements should be consistent with the company's definitions and meanings.

Usually in a large project, a single application analysis model is developed from multiple domain models: some parts of the domain models that are outside the scope of the project will be thrown away; some pieces of the domain models will be merged into a single element in the application model. The transformation from the domain models to the application model is not a procedure of direct mapping. This makes judging completeness of the application model more difficult. On the other hand, an analysis model may be too complete for containing design information, which leads to an overly constrained design, and therefore should be removed from the model. According to [McGregor+01], evaluation criteria for an application model are as follows:

- Completeness: The ideas expressed in each use case can be represented by the concepts and algorithms in the model. No design information is included in the model.
- Correctness: Experts agree with the attributes and behaviors assigned to each concept; on the steps in each algorithms; major states for each conceptual entity.
- Consistency: Where there are multiple ways to represent a concept of action, those ways are equivalent.

3.3.1.2.3 Design Models

There are two basic design models in an object-oriented project: the architectural design model, which provides the basic structure of the application by defining how a set of interfaces are related and specifying the exact content of each interface; and the detailed class design model, which provides the precise semantics of each class and identifies the architectural interface to which the class corresponds [McGregor+01].

The architectural model represents the skeleton of the entire application, in which the nonfunctional requirements are blended with the functional requirements. This

helps to use scenarios as the basis for model performance and other important architectural constraints. Three types of information that are relationship, states, and algorithms, are widely used to represent architecture. Although the UML does not have specific syntax for describing these information, there are some tools, e.g. ObjectTime, that provide facilities for animating design diagrams and automatic checking of some aspects of models.

Test cases for the architectural model are developed from use cases. Each use case describes a family of scenarios. The test cases are essentially defined at a level of the interfaces between subsystems. Test cases can be executed manually by constructing message-sequence diagrams.

According to [McGregor+01], evaluation criteria for an architectural model are as follows:

- Completeness: A sufficient set of interfaces are defined to provide all of the services needed for the application's functionality. The relationship between the interfaces allows for the flow of control and data necessary to realize all of the uses described in the use case diagram.
- Correctness: The architecture satisfies its constraints; use the appropriate architectural patterns; represents the interactions between the interfaces.
- Consistency: Each use of the system can be handled only in one set interface.

Moreover, an architectural model may have a specific set of quality attributes, such as performance and scalability, should also be evaluated.

The detailed class design model is developed from the architectural model with classes that implement the interfaces defined in the architecture. The detailed class model typically includes a set of class diagrams, the pre- and postconditions in *Object Constraint Language* (OCL) for every method of every class, activity diagrams of significant algorithms, and state diagrams for each class [McGregor+01].

The focus of detailed class model inspection is on compliance with the architecture. According to [McGregor+01], evaluation criteria for a detailed class model are as follows:

- Completeness: Classes are defined for each interface in the architecture. The preconditions for each method specify sufficient information so that the user

can safely use the method. The postconditions for a method show error conditions as well as the normally expected result.

- Correctness: Each class accurately implements the semantics of an interface. For those classes that correspond to interfaces in the architecture, the class's specification must correspond to the interface specified by the architecture.
- Consistency: The behaviors in the interface of each class provides either a single way to accomplish a task or, if there are multiple ways, they provide the same behavior but with different preconditions.

The test cases at this level are very much like the final system test cases because much detailed information is available and need to be considered. This is the last stage prior to implementation and code-based testing.

3.3.2 Test Classes

The fundamental unit of an object-oriented program is a class. Class testing is roughly analogous to unit testing in traditional testing processes. It focuses on individual classes and small class clusters, which include several classes that are so tightly coupled that testing constituent classes in isolation is impractical. The cluster head is a single class that uses the others as instance variables or as message parameters [Binder00]. Class testing also addresses some integration issues on the method level or on the class level.

Usually, class testing is performed by its developer. A test plan is developed soon after a class is fully specified and ready for coding. After the class is developed and prior to the use of the class in other portions of the software, the class testing should be done. Classes are typically tested by a combination of code inspection and execution-based testing [McGregor+01]. Execution-based class testing requires the identification of test cases, the development of a test driver to apply the test cases against instances of the Class under Test (CUT), and the execution of the test driver [ibid].

3.3.2.1 Test Single Classes

The purpose of single class testing is to ensure the implementation of a class exactly meets the requirements set forth in its specification. If each class meets its specification, then any bugs that appear in integration testing are more likely caused by incorrect interfacing of classes than by incorrect implementations of the classes.

Single classes are also referred to *primitive classes*, which can be instantiated and the instance can be used without need to create any other instance of any class, including the primitive class itself [McGregor+01].

The set of operations of a class can be divided into four broad categories: constructor which is a class operation used to create and sometimes initialize a new object, accessor and modifier which have been discussed in section [3.2.1.1], and others which do not fall into any of the above categories, such as destructor. The first thing in testing a single class is to check if constructors and accessors are correct by executing test case for them, since if these methods are incorrect, the testing results for other methods are unreliable. The test cases for this testing purpose is called *baseline test suite* [McGregor+01].

Test cases are usually identified and generated from the class specification, which can be expressed in object constraint language (OCL) and/or state transition diagrams. In addition, code-based test cases are added as needed to improve code coverage.

- Test case construction from class constraints (pre- and postconditions)

  There are two basic design approaches to define interface of messages: *design-by-contract* which requires the sender to meet the receiver's precondition, and *defensive design* by which a method always check incoming messages and reject those violating its precondition. Generally, defensive design approach is better, although it needs more test cases than the design-by-contract approach to achieve coverage.

  The general idea for identifying test cases from pre- and postconditions for an operation is to identify scenarios for test cases for all possible combinations of situations in which a precondition can hold and postconditions can be achieved [McGregor+01]. Then create the test cases from the scenarios with specific input values, including typical values and boundary values, and determine the expected outputs. Finally, if the class is specified by defensive design approach, augment test cases with additional cases as needed to test the situations in which a precondition is violated. According to [McGregor+01], basic steps to construct test cases from pre- and postconditions are as follows:

  - Identify a list of precondition contributions
  - Identify a list of postconditions contributions
  - Make all possible combinations of the entries from the contribution lists

- Eliminate any conditions generated by the table that are not meaningful

- Test case construction from state transition diagrams

State transition diagrams describe the dynamic behavior associated with instances of a class graphically. These diagrams supplement or comprise the written specifications. Each transition on the diagrams represents a scenario for one or more test cases. Generating test cases from the state transition diagrams is more intuitive than generating them from pre- and postconditions.

- Test case construction from class source code

Code-based test cases are added to test boundaries introduced by the implementation. The typical testing techniques for code-based testing are control flow analysis and data flow analysis, which are discussed in more detail in [2.3.5.3].

- Adequacy of test cases for a class

Exhaustive testing is usually infeasible or impractical under time and resource constraints. Therefore, some measure of adequacy must be applied to indicate the level of confidence in the quality of the test cases. Three commonly used measures of adequacy are constraint-based coverage, state-based coverage, and code-based coverage [McGregor+01].

Constraint-based coverage can be measured in terms of how many pairs of pre- and postconditions have been covered by the test cases. State-based coverage can be based on how many transitions in a state transition diagram have been covered. State-based coverage analysis is quite useful for finding missed test cases if the test cases were generated from pre- and postconditions. Code-based coverage criteria, such as statement coverage, branch coverage, decision (multi-decision) coverage, and path coverage, can be used to measure the adequacy of how much of the code is executed across all test cases in the suites.

3.3.2.2 Test Interactions

Object interaction is a request by one object (the sender) to another (the receiver) to perform one of the receiver's operations and all of the processing performed by the receiver to complete the request [McGregor+01]. The purpose of interaction testing is to ensure that messaging occurs correctly with the objects, that is, the instances of the classes that have already been tested separately. An instance of a trusted class may

contain no faults, but if the services of that instance are invoked by other objects in a wrong way, then program contains faults. Therefore, the correct interaction between objects is critical to the correctness of the program.

3.3.2.2.1 Identifying interactions

Although interactions actually take place between objects at runtime, for example when one object is passed to another as parameter or when an object maintains a reference to another object as part of its state, these interactions can partly be identified by a class specification in which references are made to other classes. Generally, a class interacts with other classes in one or more of the following ways [McGregor+01]:

- A public operation names one or more classes as the type of a formal parameter.
- A public operation names one or more classes as the type of a return value.
- The method for a class creates an instance of another class as part of its implementation.
- The method for a class refers to a global instance of some class.

A nonprimitive class can be categorized into two types based on a degree of interaction with other classes: *collection class* which maintains associations with instances of other classes, but never actually interact with those instances; and *collaborating class* which interacts more extensively with other classes than a collection class.

- Collection classes

  Collection classes can be identified by a specification that refers to other objects, but that does not refer to values computed based on the state or attribute value of those objects [McGregor+01]. Examples of collection classes include container classes, such as lists, stacks, queues, and maps.

- Collaborating classes

  Collaborating classes are those classes that use other objects in one or more of their operations [McGregor+01]. When a postcondition of an operation in a class's interface refers to the state of an object, and/or specifies that some attribute of that object is used or modified, then the class is a collaborating class.

3.3.2.2.2 Test object interactions

Object interactions are tested by observing the internal state of the receiving object and those objects with which it has an association. Interaction testing is primarily based on specifications of public operations rather than based on implementations since the associated classes assumedly have already been adequately tested.

- Testing collection classes

  Collection classes can be tested using techniques for primitive classes discussed in [3.3.2.1]. Some specific testing considerations for collection classes include:

  - Ensuring instances that are passed as parameters in messages to a collection under tested are correctly incorporated into and removed from the collection.
  - Testing any limitations placed on the capacity of the collection.
  - Applying negative tests if the defensive design approach has been used.
  - Applying state-based testing techniques to test sequences of operations.

- Testing collaborating classes

  Testing collaborating classes is much complicated than testing collection classes or primitive classes. Since interactions between objects of collaborating classes change object states from an "input" state to an "output" state, state-based test cases can be used to test object interactions. [Turner93] proposed a state-based method focusing on testing the interactions between objects. The steps of using this approach are:

  - Identifying object states which are combinations of attribute values.
  - Identifying state transitions from input states to output state.
  - Building a finite state machine (FSM) to represent the object interactions.
  - Generating test cases from the FSM.

  This method also takes into account the random order in which the messages are sent, and proposes using substates, which is a combination of values of a subset of the attributes of a class, to simplify the testing process.

3.3.2.2.3 Sampling test cases

With any testing approach it is always an important issue that how to increase the level of coverage systematically. There are a number of possibilities for determining

which test cases to select [McGregor+01]. One approach is based on the probability distribution on a user profile. The higher the frequency of use, the larger the probability of selection. Another approach is to select a stratified sample in which test cases are selected from a series of categories. For example, use the actors from the use case model as the basis for stratifying test cases. These stratified test case samples provide an effective means of increasing the reliability of the system under test (SUT). Test cases can also be selected by using a random value generator. The advantage of this approach is that over iterations and reapplications of test cases, many values in the intervals will be tested rather than the same ones over time.

Most of the faults resulting from interactions are between pairs of objects rather than among several objects [McGregor+01]. One specific technique for selecting a sample is *orthogonal array testing system* (OATS), which is very efficient to limit the explosion of test cases by defining pair-wise combinations of a set of interacting objects.

- Orthogonal array testing system (OATS)

  An *orthogonal array* is an array of values in which each column represents a *factor*. In software system a factor represents a specific class family or a set of states of the class family. Each factor can take on a certain set of values called *levels*. Each level will be a specific class in a class family or a specific state of an object. In an orthogonal array, the factors are combined pair-wise rather than representing all possible combinations of the levels for the factors. This is a systematic way of reducing the number of test cases. OATS uses a balanced design in which every level of a factor will appear exactly the same number of times as every other level of that factor.

  According to [McGregor+01], the following five steps are used to identify test cases by OATS:

  - Identify all factors.
  - Determine levels for each factor.
  - Locate a *standard orthogonal array* that fits the problem.
  - Establish a mapping from each factor onto the integers in the array so that the standard array can be interpreted.
  - Construct test cases based on the mapping and the row in the table

3.3.2.3 Test Class Hierarchies

Generally, it is more straightforward to test classes in an inheritance hierarchy from the top down. That is, testing a subclass D of a superclass C that has already been tested.

It is reasonably assumed that inheritance is used only in accordance with the *substitution principle*, which will be discussed in more detail in the next section, since the substitution principle ensures that objects bound to an interface behave as expected. This results in more reliable and readable code when applying polymorphic mechanism.

3.3.2.3.1 Substitution principle

According to the substitution principle [Liskov+94], an instance of a subclass D can be used whenever an instance of the superclass C is expected, if only the following changes are allowed in defining the behavior associated with a new subclass:

- The preconditions for each operation of class D must be the same or weaker than those for the operation of class C
- The postconditions for each operation of class D must be the same or stronger than those for the operation of class C
- The class invariant for class D must be same or stronger than that for class C

3.3.2.3.2 Refinement possibilities and test case design

If inheritance is applied during design in accordance with the substitution principle, an inheritance relation also holds for the test cases for the inheritance hierarchy [McGregor+01]. With careful analysis of the incremental changes when a subclass is derived from its superclass, testers can sometimes avoid testing some parts of a subclass since the test cases applying to the superclass just exercise the same code that was inherited intact in the subclass.

Generally, inheritance permits only a small number of incremental changes in deriving a class D from a class C [McGregor+01]. The following is the allowed ways of inheritance, together with corresponding test case design considerations:

- Add one or more new operations in the interface of D and possibly a new method in D to implement each new operation.

In this way, specification-based test cases are required for each new operation. If the operation is not abstract and has an implementation, implementation-based and interaction-based test cases need to be added to comply with coverage criteria.

- Change the specification or implementation of an operation declared by C in one or two ways:
  - Change in D the specification for an operation declared in C

    New specification-based test cases, which meet any weakened preconditions and check outputs for the new expected results from any strengthened postconditions, are required for the operation. The test cases for this operation designed for C still apply, but must be re-run, and the expected results may be revised according to the strengthened postcondition.
  - Override in D a method in C that implements an operation inherited by D

    All the inherited specification-based test cases for the method can be reused. The implementation-based test cases and the interaction-based test cases need to be reviewed. Some test cases need to be revised, and new test cases need to be added as needed to meet the test criteria for coverage.
- Add into D one or more new instance variables to implement more states and/or attributes

  If a new variable is added with new operations and/or code in overriding methods, then testing will be handled in connection with them. Occasionally, a new variable is not used in any method, and no changes need to be made.
- Change the class invariant in D

  Class invariants act as implied postconditions for every operation, but usually it is not necessary to explicitly refer to them in designing test cases. Thus, if a class invariant changes, all the inherited test cases need to be rerun to verify that the new invariant holds.

Part of the specification of D is inherited from C. The substitution principle ensures that all the specification-based test cases used in testing C can be used in testing D. If an operation has not changed in any way, either in specification or in implementation, the test cases for this operation do not need to be rerun. However, if its method has changed indirectly because it uses an operation that itself has changed,

all the test cases for this operation need to be rerun, and additional implementation-based test cases might be required for such method.

The above analysis and its results can be used to determine for a subclass what test cases need to be added or revised, what inherited test cases need to be rerun, and what inherited test cases can be reused. The applying of this approach is referred as *hierarchical incremental testing* (HIT) [McGregor+01].

3.3.3 Integration Testing

Integration testing is testing performed to catch any errors when two or more individually developed components are combined to execute their functionalities [Bashir+99]. While on the class level the integration testing considerations concentrate on the structures of classes and class clusters, in the integration testing the major focus should be class interfaces, integrated functions, and integrated behaviors [Kung+98]. Typical errors in integration testing include internal and external interface errors, timing errors, and throughput errors [Bashir+99].

Traditional approaches to integration testing include top-down, bottom-up, and sandwich approaches. Since all of them were designed for integrating components in a traditional program, they might not be applicable to object-oriented software due to the differences in their structures and behaviors. As presented in section [3.2.2], there exist five integration levels in object-oriented software. The objective of this section is to discuss the integration testing on the component and system level, which are extensions of the class-level unit testing approaches discussed in section [3.3.2].

3.3.3.1 An Integration Strategy Based on a Virtual Integration Engine

How to determine the order of objects integration is a fundamental question in integration testing. [Bashir+99] proposes an integration strategy based on a virtual integration engine (see Figure 5). There are two queues going into this integration engine. One queue contains the use cases associated with the system under test; another queue contains the objects under test. The order of objects integration depends on the order of use cases. One of the benefits of this approach is that it requires minimal test stub generation.

The use cases are sorted in order of simplicity and interdependence. The use cases independent of other use cases should be on the top of the use case queue. An algorithm for ordering use cases are as follows [Bashir+99]:

- Draw a use case tree bottom-up such that the nodes represent the use cases and the edges represent the dependency between use cases. The edge is from use case A to use case B if B is dependent on A.
- Mark all the leaf use cases in the tree. These use cases are independent, therefore should be tested first.
- Order the leaf use cases in terms of simplicity.
- Put the ordered use cases into the use case queue. The simplest one is at the head of the queue. The term of simplicity can be in terms of number of classes, aggregate complexity of the classes, and/or interclass dependency.
- After ordering all leaf use cases, consider the use cases that are dependent on the leaf use cases only.
- Follow the same sequence until all use cases have been ordered.

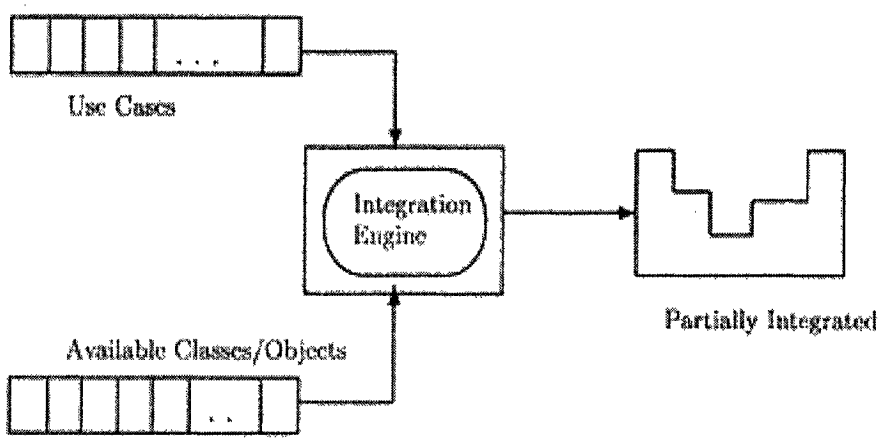

Figure 5 A Virtual Integration Engine for Objects Integration

If an error occurs during the objects integration for use case U1, the U1, together with all the use cases that depend on it directly or indirectly, will be put to the end of the use cases queue. This implies that all these use cases can not be integrated now. Then the object integration for other use cases continues. Once U1 is fixed and ready for testing, U1 and its dependent use cases can be upgraded to the front of the queue and be available for integration testing again.

3.3.4 System Testing

The objective of *system testing* is to determine whether the software system is ready for its intended users by observing its behavior, over a period of time, in a simulated or real environment [Campbell+95].

Traditionally, all software testing activities are intensely sustained during the system testing phase. In the object-oriented parallel testing life cycle, however, a large amount of these activities have been distributed throughout the software development process. The distribution of testing reduces the marathon effort required in the traditional system testing phase. In addition, it also helps to identify errors at earlier stage.

3.3.4.1 Types of testing

Typically, activities of system testing include sanity testing, functional testing, performance testing, and acceptance testing etc. Furthermore, system testing is not restricted to code testing. Documentation, including system analysis and design documents, user manuals, training material, system error catalogs, on-line help, developer's guides, and system maintenance documents, is one of the most critical pieces of a software system, and need to be tested thoroughly in the system testing phase.

- Sanity testing

  Sanity testing is also referred to installation and system startup/stop testing. It ensures a smooth installation process and system startup when a software system is initially installed and brought on-line.

- Functional testing

  In *functional testing*, the functionality of the system is checked against its requirements. It ensures that each requirement is completely satisfied by the system. The dynamic black-box testing techniques discussed in section [2.3.5.2] are generally used for testing functionality of a system.

- Performance testing

  Performance testing is an information-gathering and analysis process in which measurement data are used to predict when load levels will exhaust system resources [Nguyen+03]. The most important aspect in performance testing is defining and establishing the context within which performance will be measured.

The context means a description of the environment, such as network traffic, server load, client machine activity, and database activity etc, in which the measurement will be made.

- Acceptance testing

  Acceptance testing is performed by customers prior to the official ending of development activity. It is usually carried out in the deployment environment at the customer site. Acceptance testing is a special kind of system testing.

3.3.4.2 Test case design

The general principles for traditional system testing are also applicable to object-oriented software system testing. Test cases derived at the stage of analysis and design model testing can be used at this stage. Additional test cases might be derived from the use case model to cover all uses of the system. This approach works fine for being certain that the product does what it is supposed to do.

Usually a use case contains multiple scenarios, such as normal scenario, alternative scenarios, and exceptional scenarios. Each scenario can be converted to several test cases by substituting the data attributes in the use case model with specific values, such as typical values and boundary values. The process for identifying the specific values has four steps [McGregor+01]:

- Identify all of the data attributes in the use case model
- Identify equivalence classes of data attributes for each input condition
- Construct tables that list combinations of values from the various equivalence classes
- Construct test cases that combine a single permutation of values with the necessary environmental constraints.

3.3.5 Regression Testing

Regression testing is the process that is performed after changes have been made to a software system. The purpose of regression testing is to ensure the changed software still satisfies its intended functionality.

The iterative process employed in object-oriented software development demands effective regression testing. Object-oriented technologies can not prevent all regression bugs although they offer strong support for reuse. Especially, regression testing has become a part of, in fact a good first step, of integration testing. In the

integration process, when new components are added to the current increment, or some components in the current increment have been changed, accumulated test cases are rerun to reveal regression bugs. However, the regression testing does not replace the integration testing. It does not reduce the need to develop and run tests for new and changed components at either component or system level.

Regression testing is typically performed during iterative development process, during integration testing stage, and during application maintenance. According to [Binder00], regression testing is indicated in the following situations:

- When a new subclass had been developed
- When a superclass is changed
- When a server class is changed
- When a bug fix has been completed
- When a new increment is generated for system scope integration testing or system testing
- When a new system build has been generated
- When a system has stabilized and the final release build has been generated

Regression tests should be run in some kind of automation. After several incremental cycles, regression test cases can become quite large. It may be hard for testers to see benefits in the laborious repetition of large amounts of tests that have already passed. Regression tests run from a GUI capture/playback tool are the workhorses of testing. Test scripts developed for these tools can be quite effective.

3.3.5.1 Test suite maintenance and selection

As applications' interface, capabilities and implementation evolve, test suite decay occurs inevitably. Some test cases become obsolete and can be (or must be) discarded [Binder00]:

- Broken test cases that can not be run on the present build, usually because of changes to interface.
- Obsolete test cases that can be accepted by the SUT, but are no longer appropriate because requirements have been changed.
- Uncontrollable test cases that may not be repeatable because of dubious value.
- Redundant test cases. Two or more test cases are redundant if their input and output for a specific interface is identical.

Even conscientiously maintained test cases can grow quite large. Sometimes, rerunning entire test cases is not practical. In such situation, the test cases need to be reduced safely based on the accurate dependency analysis.

Since there exist dependencies among the components of an object-oriented system, like inheritance, association, and aggregation, changes in a part of the system may affect other parts. Many factors, including the dependency relation among the components, the changes, and the changed components, determine the scope of change impact. The main concern in regression testing is how to reuse the existing test cases, and how to effectively and efficiently identify the changes and their impact so that testing can be focused to the changed and affected components [Kung+98].

[Kung+94] proposed an approach based on a test model called object relation diagram (ORD) to analyze and identify changes and the impact of the changes on class level. A reverse engineering method has been developed to extract the classes and their dependencies from source code, present them in the ORD, and identify changes by comparing two different versions of ORD from source code. The changes then can be used to determine the changed classes and the impact of the changes. Another approach was proposed in [Rothermel+94] to select regression test cases by capturing control dependencies and data dependencies among the statements and methods of object-oriented software and representing them in interprocedural program dependence graph (IPDG) and Class dependence graph (CIDG). From these graphs the statements in the modified program that will produce different results are identified, and test cases that traverse through these statements need to be rerun. [Hsia+97] addresses the same problem as [Rothermel+94], but use a totally different approach. It proposed obtaining a relationship between test cases and classes by inserting probes, which can record which test cases touch which of the classes, into source code. Thus, the test cases related to those changed and affected classes, which can be identified using the method proposed in [Kung+94], must be rerun to test the modified program.

3.4 Summary

In this chapter, we provided a brief overview of the object-oriented paradigm and its impact on software testing. We discussed in detail the life-cycle object-oriented

software testing, which is the basis for the life-cycle testing process for web applications.

Chapter 4 Web Application and Modeling

A web application is a kind of thin-client client/server system [Nguyen+03]. Today, most web applications are running on an n-tiered architecture. Each tier is built on the component-based techniques. Web applications have been widely adopted as the basic infrastructure of information systems in many areas, such as commerce, education, and government services.

A challenge that we face in web application testing is to learn the associated technologies in order to have a better understanding of how web technologies affect the interoperability of software components, as well as web systems as a whole. Such knowledge is essential to effective grey-box testing. In this chapter, we introduce necessary background knowledge about web application architecture and techniques, and then we discuss how to model web applications with extended UML, which is the basis for the model-based testing.

4.1 Web Application Architecture

4.1.1 Distributed Systems, Client/Server Systems and Web Applications

A *distributed system* is a system in which computations are performed by separate programs, normally running on separate pieces of hardware, which cooperate to perform the task of the system as a whole [Lethbridge+01]. Distributed architectures are used for both traditional client/server (C/S) systems and web applications.

Generally, client/server systems can be divided into two types: thin-client system and thick-client system. In a *thin-client system*, the client is made as small as possible and most of the work is done in the server. By contrast, in a *thick-client system*, as much work as possible is delegated to the clients; e.g., the client application handles data processing and applies logic rules to data, while the server is responsible only for providing data access features and data storage.

A web application is a thin-client C/S system. However, Web applications can be further divided into types of thin-client and thick-client. In a thin-client web application, the server is responsible for all services, and only a plain HTML/XML

page is sent back to the client. In a thick-client one, however, components such as ActiveX controls and Java applets are downloaded and launched on the client computer when they are requested by the client for data processing.

Compared to the traditional C/S systems, web applications have some specific characteristics that make them more complex in design and testing:

- Heterogeneous and platform-independent

  A web application system may run on mixed hardware environment with multiple operating systems, utilizing multiple software packages and components. The web application is independent of the types, brands, and models of web servers and browsers.

- Concurrent natures

  Usually a web server processes concurrently tens of thousands requests from client applications.

- N-tiered application architecture

  Today, most of web applications are running on an n-tiered architecture, that is, client tier, web server tier, middle tier, and back-end tier. Furthermore, server side components, including web server tier, middle tier, and back-end tier, can be distributed among any number of physical server machines.

- Mix of Object-Oriented & Procedure-Oriented Programming

  Although a web application can be considered generally as an object-oriented system, it also contains some non-object-oriented components, e.g. scripts embedded in web pages. In fact, a web application is mixed of object-oriented programming and procedure-oriented programming.

- Server-based applications

  No special software and configurations are required on client side. Browsers, together with add-on/plug-in components, provide standard client-side applications. Deploying a web application is usually a matter of setting up server-side components on a network.

- Navigation-based interaction structure

  HTTP is the principal communication protocol used by web applications. Communication between client tier and web server tier typically revolves around the navigation of web pages. Usually there is no direct communications between

server side and client side objects. In one level of abstraction, all messaging between the client tier and the web server tier in a web application can be described as the request and reception of web page entities.

4.1.2 General N-Tiered Web Application Architecture

Generally, web application can be described from two perspectives: architectural perspective and functional perspective.

From the architectural perspective, a web application can be divided into four related tiers (see Figure 6 [Nguyen+03]):

- Client tier

  Client tier components include web browsers and add-on/plug-in Components. The web browser is the most important software running on client-side. It provides users a graphical interface to view and navigate through web pages. Netscape Navigator and Microsoft Internet Explorer are popular web browsers. The Add-on/Plug-in Components, such as Java applets and ActiveX controls, are downloaded and reside on the client side to support various forms of interactivity and animation within web pages.

- Web server tier

  Web servers are the most essential type of web server tier components. A web server stores web pages or HTML files with their associated contents, such as multimedia source files, and makes them available to client tier. Web servers may serve advanced technology components, such as JSP and ASP, CGI programs, NSAPI/ISAPI programs, Java servlets, and ActiveX controls. Web servers work with protocols such as HTTP(s) and FTP to communicate with browsers. There are many popular web server products including Microsoft IIS, Netscape web server, and Apache.

- Middle tier:

  The middle tier performs most business logic of a web application, so it is also referred as *business tier*. The middle tier consists of some business objects that execute business logic by their collaboration. Besides, the middle tier also provides services to web server tier. For example, a web server tier does not interact with a back-end tier directly. Instead, the middle tier interacts with both of

them and acts as a Web-to-Database Connector. Common component techniques utilized in a middle tier include Enterprise Java Bean (EJB) and ActiveX control.

- Back-end tier

  Back-end tier is the data repositories for web applications. Back-end tier may consist of database system, enterprise resource planning (ERP) system, or legacy applications. It provides data management for a web application.

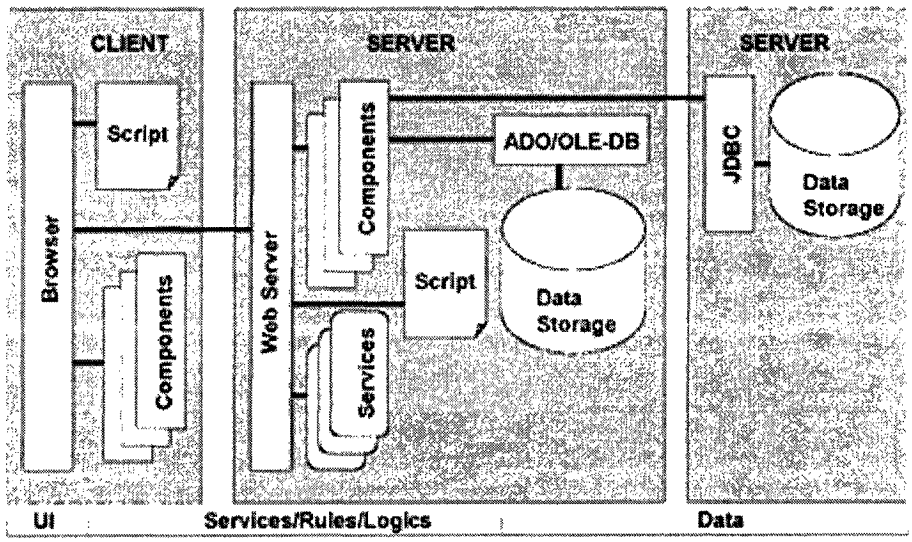

Figure 6 N-Tiered Web Application Architecture

On the other hand, from the functional perspective, all web applications do almost the same things, that is, affecting the state of business driven by user input:

- Provide users with an interface for navigation and data input
- Collect input user-data on the client side and transmit user-requests to a web server
- The web server processes the user-data and requests using some sort of middleware
- The middleware may interact with databases for data query and/or update
- The web server processes result-data
- Transmit the processed result-data back to the client side
- Finally, the returned data is displayed to the user. Display might be as simple as interpreting HTML, or as complex as performing calculations, sorting, or other manipulations of the data.

4.1.3 Web Application Architecture Based on J2EE Platform

The Java 2 Platform, Enterprise Edition (J2EE) is a Java Platform which provides a component-based approach to the design, development, assembly, and deployment of distributed, multitier enterprise applications. The J2EE programming model is flexible enough for applications that support a variety of client types, with both web container and EJB container as optional. The focus of this thesis is only on multitier web application architecture (see Figure 7 [Singh+02]).

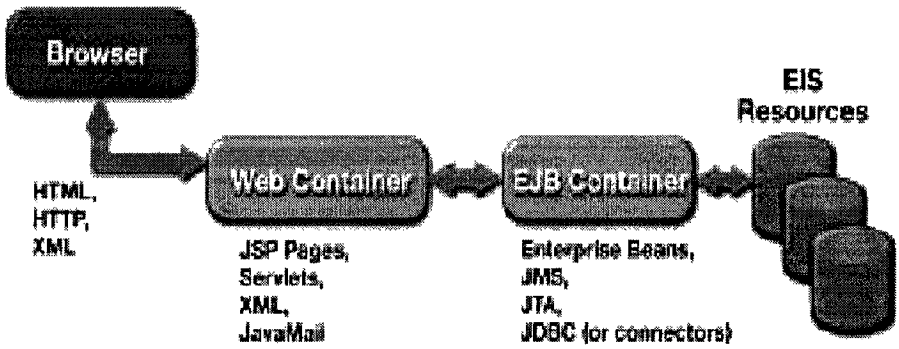

Figure 7 Web Application Architecture Based on J2EE

The web container hosts web components that are almost exclusively dedicated to handling web applications' presentation logic. JSP pages, supported by servlets, generate dynamic web content for delivery to browsers. The EJB container hosts application components which usually implement the business logic of web applications. These components use EIS resources to service requests from web-tier components. This architecture is implicitly scalable. It decouples data access from web applications' user interface. The back-office functionality of web applications is relatively isolated from end-users' look and feel.

It is worth noting that XML plays an integral role in this architecture. With the ability to produce and consume XML data, web container provides an extremely flexible way to exchange messages between client-side and server-side. Actually, Java and XML are complementary technologies: Java language offers portable code, XML provides portable data.

4.2 Basic Web Application Development Techniques

In this section we introduce basic techniques which are commonly used in web applications to support the architecture that we have discussed in the latest section.

- Protocols

The Internet is a packet-switched network. *TCP/IP* is a set of de facto standard protocols used on the Internet. The TCP/IP stack is composed of five layers (see Figure 8 [Nguyen+03]).

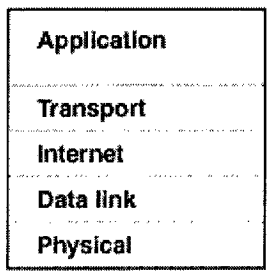

Figure 8 TCP/IP Stack Architecture

The *application layer* interacts with end-user applications. The protocols in this layer enable end-user applications to establish a connection (session) between two computers, and send, receive, and convert data into their native format. *HTTP* (HyperText Transfer Protocol), which is a connectionless protocol designed for robustness and fault tolerance instead of maximum communication throughput, is the most important protocol used in the application layer. It commonly used in browsers and web servers to transfer web pages and other related data. *HTTP(s)* (HyperText Transport Protocol Secure) is the standard encrypted communication mechanism on the Internet. It is actually just HTTP over *SSL* (Secured Sockets Layer), which is a protocol that transmits data over the Internet in an encrypted form. Other common protocols associated with the application layer include *FTP* (File Transfer Protocol), *SMTP* (Simple Mail Transfer Protocol), *MIME* (Multipurpose Internet Mail Extension), and *DHCP* (Dynamic Host Configuration Protocol) etc.

The *transport layer* breaks data into packets before sending them, and ensures that all packets arrive intact and reassembles the packets into correct order after receiving them. Two common protocols in this layer are *TCP* (Transmission Control Protocol) and *UDP* (User Datagram Protocol). TCP is a connection-

oriented protocol. It provides reliable stream of data between computers on networks. UDP is a connectionless protocol. It does not check dropped data, but can transfer data more quickly.

The *internet layer* receives data packets from the transport layer and sends them to the correct IP address. Another important task of this layer is to determine the best route for data traveling. *IP* (Internet Protocol) is a basic protocol in the internet layer. It is responsible for routing packets to their correct destinations. Other two auxiliary protocols are *ARP* (Address Resolution Protocol) and *RARP* (Reverse Address Resolution Protocol).

The *data link layer* splits outgoing data into frames and validates the successful delivery of data.

The *physical layer* is the hardware layer, locating at the bottom of the TCP/IP stack. It is composed of a network interface card and wiring.

- Markup Languages

HTML (HyperText Markup Language) is a standard markup language based on *SGML* (Standard Generalized Markup Language). It is primarily used in creating web pages.

XML (eXtensible Markup Language), which is also derived from SGML, defines the guidelines for structuring and formatting data in a standard, simple, and very flexible way. It is used to facilitate the generation and interpretation of data, and to ensure that other XML-compliant web systems can interpret and use that data unambiguously.

XSL (eXtensible Stylesheet Language) is a language for creating a style sheet that describes how data sent over the Web using XML is to be presented to the user.

DTD (Document Type Definition) is a HTML based schema specification method for SGML and XML documents. For XML, DTD will be replaced by the new XML Schema specification method --- *XSD* (XML Schema Definition), which uses XML to describe the structure of an XML document.

- Scripts

Scripts are programming instructions inserted directly into HTML pages. Scripts do not need to be complied. They can be executed in either client tier or web server tier.

Usually, client-side scripts are executed by browsers to dynamically set values for UI controls, modify web page content, validate data, and handle errors. There are a number of scripting languages supported by popular browsers, including *JavaScript* produced by Netscape and Microsoft's *Jscript* which both comply with ECMAScript standard [ECMA].

- Component Technologies

Applet, Servlet, EJB, and ActiveX Control are popular component technologies used in web applications.

Java applet is a short Java program that is attached to a web page, downloaded and executed by a web browser. On the contrary, *Java servlet* is a Java program which resides and executes on a web server to extend the capabilities of the server via a request-response programming model. JSP is the extension of Java servlet. In fact, JSP pages are compiled into servlets before they are used. *EJB* (Enterprise Java Beans) is Java component architecture for the development and deployment of object-oriented, distributed, enterprise-level applications. EJB is usually used to build the middle tier of web applications. Applications written using the EJB architecture are scalable, transactional, and secure. EJB is consisted of a set of *JavaBeans*, which is a Java class that can be manipulated in a visual builder tool and composed into applications. A JavaBeans component must adhere to certain property and event interface conventions.

ActiveX controls are components using Microsoft *COM* (Component Object Model) technologies. ActiveX contols can execute in client tier, web server tier, and middle tier, but they are dependent on the windows platform.

- Common Gateway Interface (CGI)

CGI (Common Gateway Interface) is a standard communication protocol that defines how a web server communicates with CGI programs on the same machine. CGI programs are any piece of software handling input and output according to the CGI standard. Usually a CGI program is a small program that takes data from a web server and does something with it, like putting the content of a form into an e-mail message, or turning the data into a database query. CGI applications are usually implemented in *PERL* (Practical Extraction and Reporting Language), although they can be written in other script languages such as Python and *PHP*

(Hypertext Preprocessor) etc, and programming languages such as C, C++ and Visual Basic etc. in the form of DLL.

- Web Server Extension-Based Programs

NSAPI, which means the Netscape server API, and *ISAPI,* which means the Microsoft Internet API for IIS, are the commonly used web server extension-based APIs. They provide a programmer's interface to the core operations of a web server. The NSAPI/ISAPI applications can take advantage of a web server's native API to access and modify the internal state of the web server, and may contribute to processing a user's request, authenticating it, locating resources, generating content and logging any results. The NSAPI/ISAPI applications can be DLLs that run in the same memory space as web server.

- Web Server Extension-Based Scripts

ASP (Active Server Page) and *JSP* (Java Server Page) are the technology that allows the dynamic creation of web pages by using the web server extension-based scripts. ASP is a Microsoft technology. It provides the capability to combine HTML, scripting and other reusable components. JSP is similar to ASP but from Sun Microsystems. A JSP page is simply an HTML file that contains additional bits of code that execute application logic to generate dynamic content. This application logic may involve JavaBeans, JDBC objects, EJB, and RMI (Remote Method Invocation) objects.

- Web Service

A *web service* is a software system identified by a *URI* (Uniform Resource Identifier). The public interfaces and bindings of web services are based on XML. *WSDL* (Web Services Description Language) defines an XML grammar for describing web services, and make them to be discovered by other software systems over the Web. These systems may then interact with the web services in a manner prescribed by their definitions, using XML based messages conveyed by Internet protocols.

SOAP (Simple Object Access Protocol) is a lightweight application protocol that uses XML and HTTP(s) together for exchange of information in a decentralized, distributed, and heterogeneous environment. Web services usually use SOAP to facilitate their communication with client applications.

4.3 Modeling Web Applications with Extended UML

Web applications are becoming increasingly complex and mission critical. They need to be modeled effectively to manage the complexity, and help us understand the system by abstracting away some of the details.

UML is the standard language for modeling software-intensive systems [Booch+99]. The basic UML modeling language has the advantage that it can be used for both design modeling as well as analysis modeling. This eliminates the need to learn multiple notations. Furthermore, UML is sufficiently simple that no special tools are required, although for large models, tool support quickly becomes necessary

UML is broadly used for traditional software modeling, and is also well suited to model web applications. However, it is apparent that some web application-specific components, such as web pages, hyperlinks and functional units in scripting languages, don't fit nicely into standard UML modeling elements. UML must be extended to provide one modeling notation for the entire system, including client tier components, web tier components and traditional middle tier components.

A web application typically has multiple models, each representing a different viewpoint, level of abstraction, and detail. The proper level of abstraction and detail depends on the artifacts and activities in the development process. There have been many efforts made to model web applications at multiple levels by using UML or extended UML [Baresi+01] [Conallen99] [Li+00] [Gornik02]. Some specific models for web applications are as follows:

- Use Cases

  A use case is a description of set of sequence of actions that a system performs [Booch+99]. It yields an observable result to a particular actor. Use cases are used to capture high-level user requirements. One of the characteristics of web applications is their navigation structure, which represents an abstraction of possible navigation routes through a system. Therefore, for web applications, the user requirements are related to both functional and navigational aspects. The traditional requirements analysis is extended to consist of two sub-activities [Baresi+01]: functional requirements analysis and navigational requirements analysis. Functional requirements analysis focuses on main user operations, while

navigational requirements analysis focuses on navigation structures needed by different users.

- State Diagrams

State diagram shows a state machine, including simple states, transitions, and nested composite states [Rumbaugh+99]. State diagrams address the dynamic view of a system.

In designing web applications, it is very important to identify all possible states the system can be in and transitions between the states. Each state usually corresponds to a web page that indicates a stable state. Each state indicates which actions can be taken or not.

- Client Tier and Web Server Tier Design Models

In the client tier and web server tier, there are web specific components such as web pages, forms, frames, functional units in scripting languages, and hyperlink relationships between components. An extension to the UML was proposed in [Conallen99] so that these elements can be modeled with the rest of web applications in a coherent and complete way:

  - Web pages can be classified into two types: server pages which contain a set of functions and collaborations that exist only on the server side, and client pages which contain a set of functions and collaborations that exist only on the client side. The server page can be modeled with a class stereotype of <<Server page>>, and the client page can be modeled with a class stereotype of <<Client page>>. The server page and client page are related to each other by a directed association. This association is stereotyped as <<build>>, since it can be said that a server page builds a client page. The key advantage of separating the server page and client page into different classes is that it helps to distinguish the server side components that can only interact with the server page from the client side components that can only interact with the client page. For example, client pages are modeled with associations to client side resources such as DOM, Java Applets, ActiveX controls and plug-ins etc., while server pages are modeled with associations to server side resources such as middle tier components, database access components, server operating system, and so on. Operations of a client or server page class are the functions in the

page's scripts, and its attributes are page-scoped variables that are globally accessible by the page's functions.

- A common relationship between web pages is hyperlink, which represents a navigation path through a web application system. The hyperlink relationship can be expressed with a <<link>> stereotyped association. This association always originates from a client page and points to either a client or server page. Tagged values are used to define the parameters that are passed along with a hyperlink.
- A form is the principal data entry mechanism in a client page. It is modeled as a class with the stereotype of <<Form>>. There is no operation belonging to a form since any operations that might be defined in the form are really owned by the client page. All input elements, such as "input", "select' and "textarea", are stereotyped attributes of the "Form" class. A form can have relationships with Applet or ActiveX controls that act as input controls. Each form also has a relationship with a server page, which processes the form's submission. The relationship is stereotyped as <<submit>>.
- Frames allow multiple pages to be active and visible to users. A frame contains client pages and targets. Two class stereotypes, <<Frameset>> and <<Target>>, and an association stereotype <<targeted link>> are used to model frames. A frameset class represents a container object and maps directly to the HTML "frameset" tag. A target class is a named frame or browser instance that is referenced by other client pages. A targeted link association is a hyperlink to another page which gets rendered in a specific target.

[Li+00] furthered the discussion of using extended UML for modeling web applications. The modeling method proposed in [Li+00] is aimed at facilitating the design, implementation and maintenance of large, complex web applications. The method consists of three models: business model which is discussed in more detail in the next section, navigation model and implementation model.

The navigation model and implementation model are used to describe the navigation structure of web sites, which represent the front-end of web applications and include the client tier and web server tier.

The navigation model is a kind of tree structure. A node in the tree corresponds to a branch of a web site. The leaves belonging to a node represent the information to be displayed, which are usually the objects from the business tier. For simplicity, navigation modeling can be divided into several parts which can be modeled separately. After all parts have been finished, they are combined into a complete navigation model. The navigation model pictures the organizational structure of the navigational space of web applications – sometimes called a site map.

The implementation model can be derived from the navigation model. It is responsible for describing the implementation of web applications with web components such as pages, forms, frames and hyperlinks. This is described in [Conallen99].

- Middle Tier Design

The middle tier is usually used to implement application business logic. It may serve as a basis for many applications, and does not include any navigation-specific information. The main concern during this step is to capture the domain semantics as independent of the implementation as possible. Therefore, the modeling for the middle tier is not much different than that for general object-oriented software.

Class diagrams are usually used to build the business model. Class attributes represent the object's intrinsic properties. A slight variation here is that some class attributes are stereotyped with <<anchor>>, which implies that these attributes are information for presentation and are used in the navigation model and implementation model [Li+00].

- Back-end Tier Design

Back-end tier includes the data repositories for web applications. It can also be modeled using extended UML [Gornik02]. Since back-end tier testing is not the emphasis of our thesis, we do not elaborate further.

The research works mentioned above try to build models for web applications at different levels: system level, design level and source code level. These models capture and abstract web-specific elements to provide testable artifacts to the life-cycle testing process. For example, we will show how to test a typical e-commerce

system in a life-cycle process by utilizing use cases and object-relation diagrams in Chapter 7.

4.4 Summary

In this chapter, we presented web application architecture and basic web techniques. We also discussed how other researchers have suggested ways to extend UML to describe web-specific components and how to use such extended UML for web application modeling at multiple levels.

Chapter 5 Formal Methods and Object-Oriented TTCN-3

Formal methods can be used in software testing for the purpose of improving testing quality. TTCN is an international standard testing language which is used in formal methods. In this chapter, we discuss briefly the formal testing method; give out an introduction to TTCN-3; present the application of TTCN-3 both in industry and academic institutes; and finally propose an object-oriented extension for TTCN-3, that is OO-TTCN-3.

5.1 Introduction to Formal Testing Method

Formal methods differ from many software engineering techniques in that they stress the importance of a rigorous semantic basis for the tools and notations used [FME04]. Such sound foundations enable a means of formal proof, which permits the analysis of computing system designs to a depth that is otherwise impossible to achieve. By formalizing different products and processes in the development cycle, software and system quality, consistency and integrity can be improved [FME04]:

- improve quality and rigor of the whole development process
- improve integrity and reliability of the system
- reduce specification errors
- improve requirements definition
- improve documentation and understanding of designs
- provide a firmer foundation for maintenance and enhancement
- explore the properties of a design architecture
- provide a more rational basis for choosing test data
- be as certain as possible that the design and implementation are error-free
- meet particular customer or standards requirements

Formal methods have traditionally been used for specification and development of software, e.g. SDL, Petri Nets, Z etc. However, formal methods also potentially are of benefit to software testing [Bowen+02]. TTCN, which stands for *Tree and Tabular*

Combined Notation (version 1 and 2), is widely used in telecom industry for conformance testing since 1980s. TTCN was developed by ISO (1984 - 1997) as part of the widely-used ISO/IEC 9646 conformance testing standard, and now is maintained by ETSI (TTCN-2++).

TTCN can be used with other formal methods, e.g. SDL (Specification and Description Language) and MSC (Message Sequence Chart), for the purpose of validation and verification. This process is facilitated by using tools from IT industry, for example telelogic TAU [Telelogic04]. In telelogic TAU, a system can be specified in SDL. Then a simulator for the system can be generated from SDL. Running the simulator will create a set of MSCs, which are used to compare with MSCs derived directly from the system requirements to ensure the system design conforms to the system requirements. Meanwhile, a validator for the system can be generated from SDL as well. When we run the validator, together with the MSCs created from the simulator, a set of test cases (test suite) in TTCN can be generated automatically for the system. The test suite then can be used to validate the implementation of the system.

5.2 Introduction to TTCN-3

Since TTCN was first developed, information technology has changed much. TTCN-3, which stands for *Testing and Test Control Notation version 3*, is developed and standardized by the *ETSI* (European Telecommunication Standards Institute) since 1998. Unlike TTCN version 1 and 2 which are only used for conformance testing, TTCN-3 is developed for general testing purpose. It supports black-box testing for reactive and distributed systems, such as functional, regression, interoperability testing, as well as conformance testing etc. Complex distributed test behavior can be specified in TTCN-3 flexibly and easily in terms of sequences, alternative, loops and parallel stimuli and responses at abstract level. It is supposed to be the first choice for test specifiers, implementors and users both for standardized test suites and as a generic solution in industrial product.

Although TTCN-3 shares the same acronym with TTCN version 1 and 2, it is quite different from them. TTCN-3 is a modular language and has a look and feel of a modern programming language. In addition to the typical programming constructs, it

contains all the important features necessary to specify test suites. It has much of well-proven testing-specific capabilities, such as [TTCN304]:

- Dynamic concurrent testing configurations
- Various communication mechanisms (synchronous and asynchronous)
- Data and signature templates with powerful matching mechanisms
- Specification of encoding information
- Display and user-defined attributes
- Test suite parameterization
- Test case control and selection mechanisms
- Assignment and handling of test verdicts
- Harmonized with ASN.1
- Different presentation formats
- Well-defined syntax, static semantics and operational semantics

TTCN-3 is built from a textual core notation on which a number of different presentation formats are possible, see Figure 9 [Wiles00]. One of the standardized presentation formats is based on the tree and tabular format from previous TTCN versions and another standardized presentation format are based on MSCs. Besides, TTCN-3 can be integrated with other 'type and value' systems, see Figure 10 [Wiles00]. For example, it fully harmonizes with ASN.1, and it is possible to harmonize with other type and value systems (possibly from proprietary languages). These two features makes TTCN-3 quite universal and application independent.

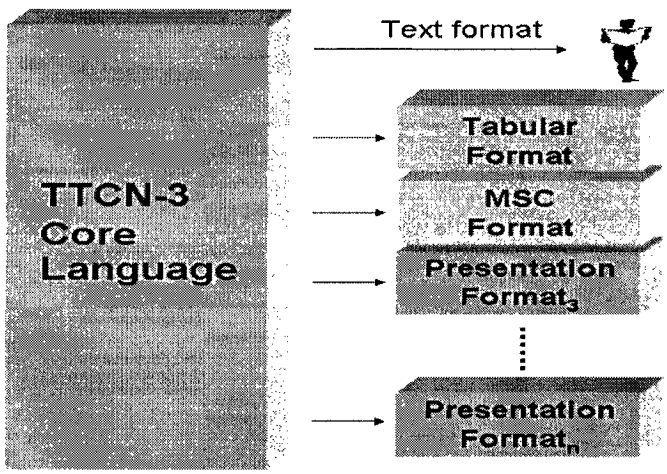

Figure 9 TTCN-3 Presentation Formats

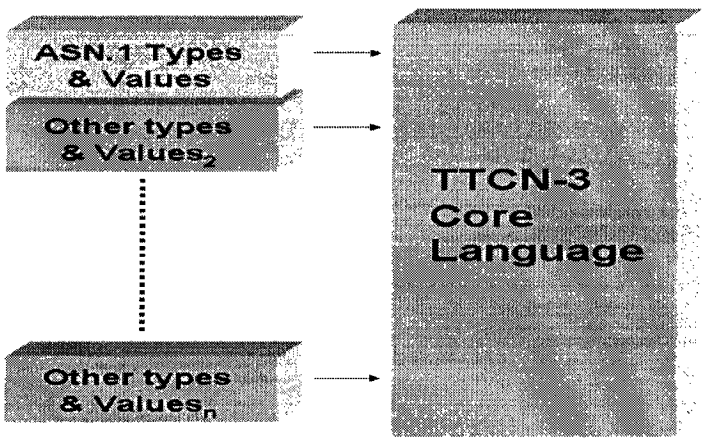

Figure 10 Integrated with Other "Type and Value" Systems

The basic definition elements of TTCN-3 include built-in and user-defined data types, templates and signatures which define actual test data, communication ports and test components which are used to build various testing configurations, and functions, named alternatives and test cases which define the dynamic test system behavior, see Figure 11 [Wiles00].

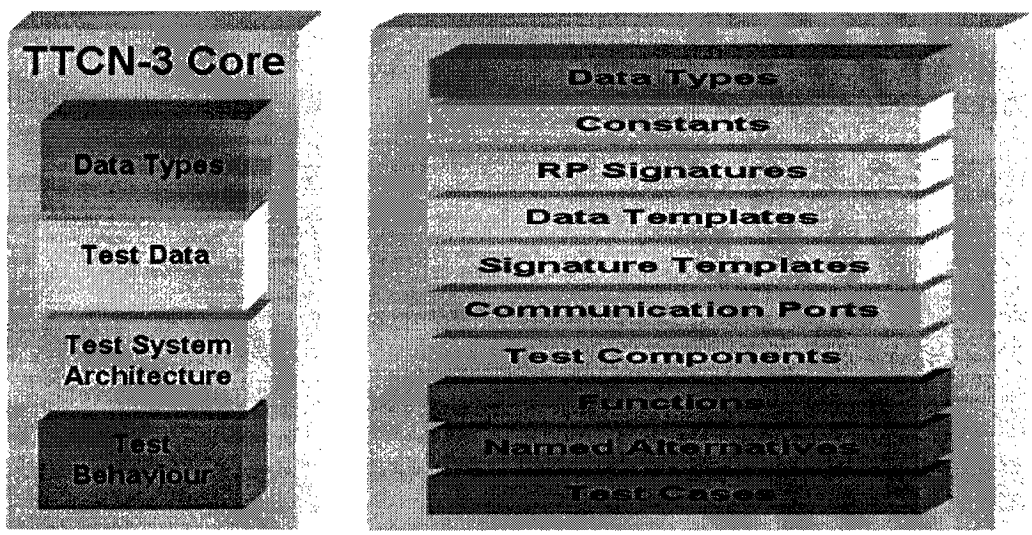

Figure 11 TTCN-3 Major Elements

Modules are the building blocks of all TTCN-3 specifications. Usually, a test suite is a module. A module has a definitions part and a control part, and it can have attributes, see Figure 12 [Wiles00]. Modules can be parameterized. The definitions in a module can be reused by other modules by an import mechanism.

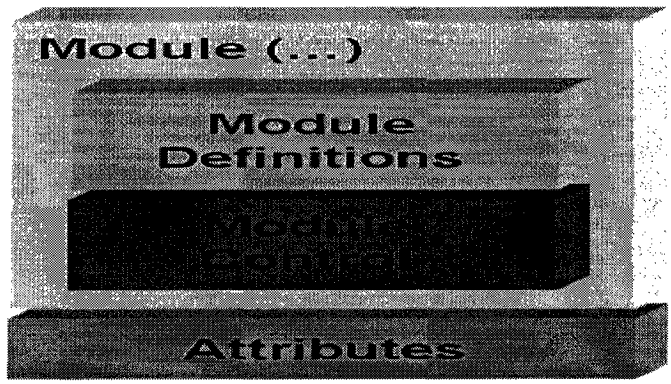

Figure 12 TTCN-3 Modules

5.3 Application of TTCN-3 and Tool Support

Typical areas of application for TTCN-3 are protocols, web services, APIs, and software modules etc. TTCN-3 can be used for the systematic testing of Internet Protocols, e.g. using TTCN-3 for the testing of SIP (Session Initiation Protocol) [Schiefer+01]. TTCN-3 can be used for the testing of web services: [Schiefer+03A] presented a mapping between XML data descriptions to TTCN-3 to enable the automated derivation of test data. TTCN-3 can be used for the testing of distributed real-time systems, with some extensions [Dai+02]. TTCN-3 can also be used to build an online test and validation platform for Internet services, as discussed in [Deussen+03].

Now there are several tools from IT industry that support TTCN-3. *TTCN-3 Toolbox*, a product of Danet Company, provides a rich set of features for Datacom testing (synchronous client/server communications), Telecom testing (asynchronous message exchange) and Web Service testing (XML-based communications) [Danet04]. *TERZO* is the product of Da Vinci Communications Company, which provides the first tabular TTCN-3 browser for test engineers [DVC04]. *OpenTTCN Tester for TTCN-3* is the product of OpenTTCN Company that supports the construction and execution of TTCN-3 test suites, both on windows and Linux platforms [OpenTTCN]. *OpenTTCN Xpress*, another product of OpenTTCN Company, is a web-based test management and control application to make the use of OpenTTCN Tester for TTCN-3 easy and accessible through intranet or Internet [OpenTTCN04]. *Telelogic TAU/Tester,* a standards-based product of Telelogic Company, offers an integrated test execution and test development environment to

support full test lifecycle: from test design through to development, analysis, execution and debug [Telelogic04]. Testing Technologies Company offers a set of TTCN-3 tools --- *TT Tool Series*: *TThree* is used to compile TTCN-3 to Java, *TTanalyze* is used to validate the correctness of test suites written in TTCN-3, *TTspec* provides a graphical test development environment for test definition, documentation and visualization of test executions in TTCN-3, and *TTtwo2three*, a TTCN-2 to TTCN-3 translator, helps a seamless migration of TTCN-2 to TTCN-3 technology [Testingtech04]. A test framework in TTCN-3 by using TT Tool Series is presented in [Schiefer+03B]

5.4 Object-Oriented TTCN-3

TTCN-3 is powerful. However, some extensions are needed to handle specific testing issues, e.g. testing real-time behavior. [Dai+02] extended TTCN-3 with a concept of absolute time to support the test and measurement of real-time requirements. [Dai+03] proposed an approach for graphical real-time test specification based on MSC and TIMEDGFT, which is the continuation of the extension in [Dai+02]. [Schmitt+03] proposed the concept of parallel operator and new matching mechanisms for records, arrays, and sets. [Deussen+03] suggested the extensions of dynamic test system interfaces and port arrays to make TTCN-3 suitable for specifying test system interfaces adapting to target systems.

As we will discuss in the following, there exist inheritance and aggregation relationships between test cases when we test object-oriented systems. In this section, we propose an object-oriented extension for TTCN-3, *OO-TTCN-3*. We do not intend to make TTCN-3 "fully" object-oriented. Instead, the extension only focuses on how to make TTCN-3 capable of specifying inheritance hierarchy and aggregation relationship in test modules.

5.4.1 Inheritance

When we test object-oriented systems, if an inheritance relationship in the system under test is applied during design in accordance with the substitution principle [Liskov+94], an inheritance relation also holds for the test cases for the inheritance hierarchy [McGregor+01]. As shown in Figure 13, class B is derived from class A in accordance with the substitution principle, and TM_A and TM_B are test modules in

TTCN-3 for class A and B, respectively. We say TM_B is derived from TM_A. The allowed ways of inheritance, together with corresponding test case design considerations are as follows [McGregor+01]:

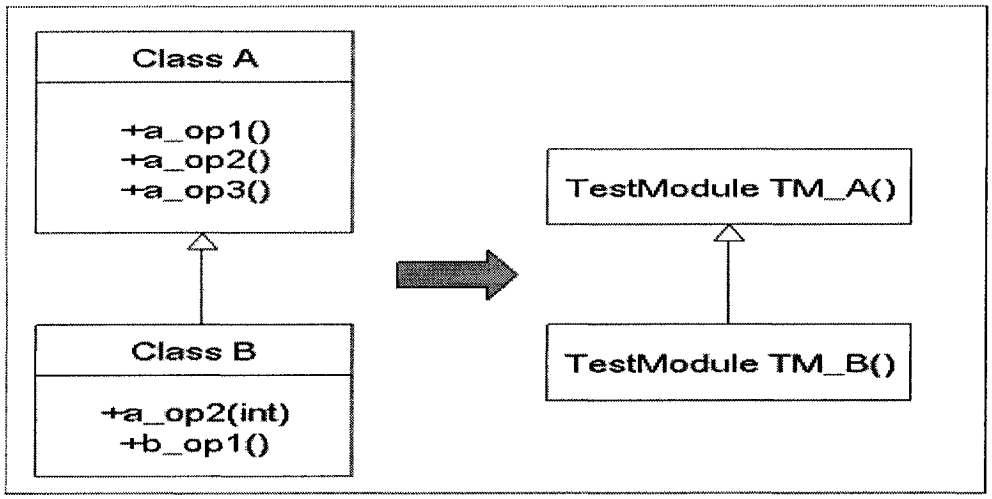

Figure 13 Class Inheritance Hierarchy and Corresponding Test modules

- A new operation, say b_op1, is added in the interface of B and possibly there is a method to implement the operation. In this way, specification-based test cases are required for the operation in TM_B. If the operation has an implementation, implementation-based test cases need to be added to comply with coverage criteria.
- If an operation in B, say a_op1 which is inherited from A, has not changed in any way, either in specification or in implementation, the test cases for this operation in TM_A still apply in TM_B, which means the test cases do not need to be rerun in TM_B if they have passed in the execution of TM_A.
- The specification of an operation in B, say a_op2 which is inherited from A, has changed. In this way, new specification-based test cases, which meet any weakened preconditions and check outputs for the new expected results from any strengthened postconditions, are required for the operation in TM_B. The test cases for this operation in TM_A must be re-run. If the expected results need to be revised according to the strengthened postcondition, the test cases in TM_B need to be overridden.
- An operation in B, say a_op3 which is inherited from A, has been overridden. In this way, all the specification-based test cases for the operation in TM_A

still apply in TM_B. The implementation-based test cases need to be reviewed. Some test cases need to be overridden, and new test cases need to be added as needed to meet the test criteria for coverage.

In short, the test cases in TM_A are possible to be reused in TM_B. TTCN-3 provides a way to reuse definitions in different modules by using the import statement. However, as a procedure-oriented language, TTCN-3 is incapable of specifying the inheritance relationship between TM_A and TM_B.

Therefore, we intend to extend TTCN-3 with fundamental object-oriented mechanism, e.g. inheritance (extend and override), to make it capable of specifying derived test modules. The extension helps to specify the inheritance hierarchies in test modules clearly, and it facilitates the reuse of test case definitions for object-oriented software in life-cycle testing process, e.g. the unit testing on class level. For example, for a simple inheritance hierarchy shown in Figure 14, we can develop test cases against the specification and implementation (if any) of the abstract class Account, and specify them in OO-TTCN-3 in test module AccountTest, even before the creation of the two subclasses: EXAccount and BankAccount. After the two subclasses have been designed and implemented, the test cases in AccountTest are ready to be reused in the test module EXAccountTest and BankAccountTest which are inherited from AccountTest. In addition, if there is any subclass derived from the class Account in the next development iteration, the test module for the new subclass can also benefit from the existing testing module hierarchy.

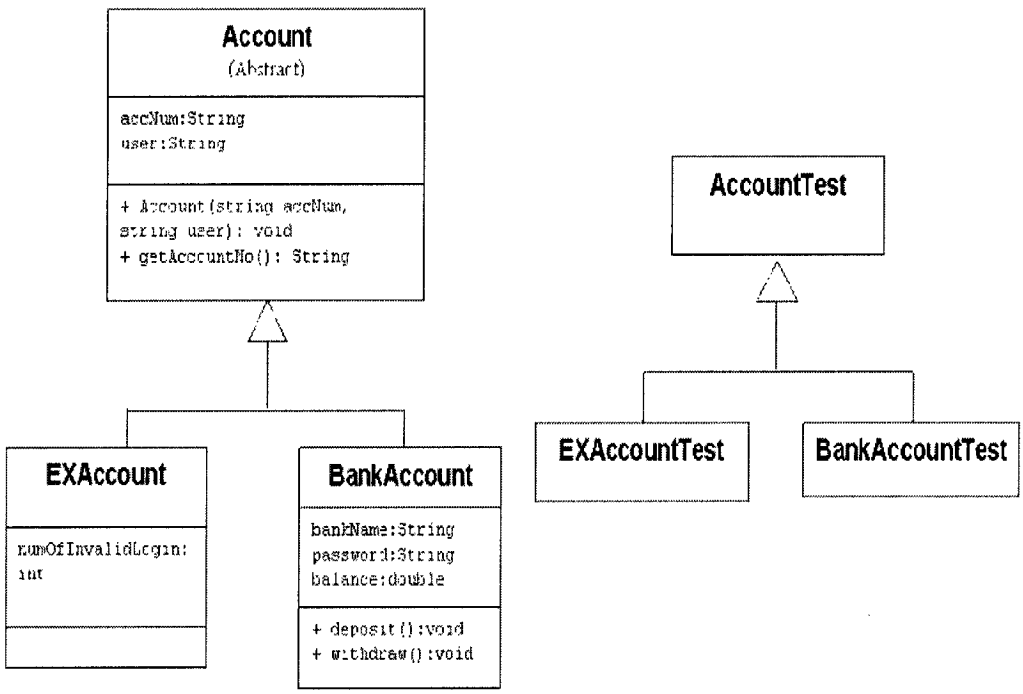

Figure 14 Partial Class Diagram for Account, EXAccount and BankAccount

In the section 7.3.3, we will show how to use OO-TTCN-3 to specify a derived test module for the middle tier of a web application.

5.4.2 Extend TTCN-3 with inheritance mechanism

To make TTCN-3 capable of specifying the inheritance relationship between test modules, first, we investigate which elements [see Figure 11 in section 5.2] in a test module are likely to be reused, and which elements can be extended or overridden in its derived test module. The result is shown in Table 1. From the result, it looks like almost all of the elements are possible to be reused directly, and some elements can be reused by means of extend or override mechanism.

We extend TTCN-3 by adding two key words: ***private*** and ***public***, to indicate if an element is ready to be reused in a derived test module, and we assume that if an element is not specified explicitly with key word private, the element is default to be public. We also propose to add a key word ***extends***, which is used to specify a test module is inherited from another test module (we do not discuss the multiple inheritance relationship in this thesis). The modified TTCN-3 syntax in BNF form is listed as follows (the sequence numbers correspond to those defined in Annex A in [ETSI201 873-1]):

```
1. TTCN3Module ::= TTCN3ModuleKeyword TTCN3ModuleId [extends
TTCN3ModuleId]
 BeginChar
 [ModuleDefinitionsPart]
 [ModuleControlPart]
 EndChar
 [WithStatement] [SemiColon]
52. PortDefBody ::= PortTypeIdentifier [extends PortTypeIdentifier]
PortDefAttribs
73. ComponentDef ::= ComponentKeyword ComponentTypeIdentifier
 [extends ComponentTypeIdentifier]
 BeginChar
 [ComponentDefList]
 EndChar
```

Table 1 Possible elements for reuse in a TTCN-3 test module

			Reuse	Extend	Override
Definition Part	**Type Definition**	Built-in	N	N	N
		User-defined	Y	N	N
		RP Signature	Y	N	Y
	Test Data	Constant	Y	N	N
		Data Template	Y	N	N
		Signature Template	Y	N	Y
	Test Configuration	Communication Port	Y	Y	N
		Component	Y	Y	N
	Behavior	Function	Y	N	Y
		Named Alternative	Y	N	Y
		Test Case	Y	N	Y
Control Part			N	N	N

5.4.3 Aggregation

There also exists aggregation relationship between test modules, e.g. between test modules for functional testing and those for unit testing. In Figure 15, a functional test scenario derived from the "User Login" use case consists of four steps. Each step corresponds to one test case defined in the test module for unit testing. The functional test scenario can be realized by executing these test cases. This relationship can be expressed as aggregation relationship in UML. In the section 7.5, we will show how to specify a test module for functional test scenario by reusing test cases defined in unit testing.

Functional Test Scenario	Unit Test Modules & Test Cases	
Use Case: User Login	Test Module	Test Case
Step 1: open index page; index page displays	hyperlinkCheck	Hyperlink_tc
Step 2: click "login" hyperlink; login page displays		Hyperlink_tc
Step 3: input "user ID" and "password", click "login" button; logonSuccess page displays	LoginServletTest	Login_tc
Step 4: click "logout" hyperlink; logout page displays	hyperlinkCheck	Hyperlink_tc

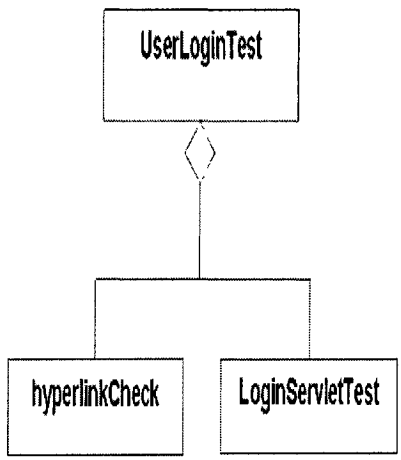

Figure 15 Aggregation Relationship between Test Modules

5.5 Summary

In this chapter, we presented a brief introduction to formal testing method. Then we introduced an international standard test specification and implementation language, TTCN-3, and its applications. Finally, we proposed an object-oriented extension for TTCN-3 to make it capable of specifying object-oriented test cases.

Chapter 6 Life-Cycle Testing of Web Applications via Object-Oriented TTCN-3

Testing web applications is a challenge because of their heterogeneous, distributed and concurrent natures. Life-cycle testing is imperative for building high-quality web applications. In such a process, various models need to be built and tested, multiple test methods are applied, and multiple testing tools may be used.

In this chapter, we propose a life-cycle testing process with object-oriented TTCN-3 and discuss how to implement testing activities at each testing phase: from the verification and validation of system analysis and design to functional and non-functional system testing.

6.1 Life-Cycle Testing Process Model

The complexity of testing web applications requires the integration of different testing methods and testing tools. Life-cycle testing for web applications is a process of applying multiple test methods on multiple models and implementation in different testing phases, with assistance of multiple test tools, as shown in Figure 16.

In this life-cycle testing process, however, we may ask two questions. One is "can the test cases, or test scripts that describe the test cases, be reused among different test phases, and if they can, how can we facilitate this reuse." Another question is "can we provide a unified interface between test cases and test tools, so that testers can write test scripts in one unified language, and do not need to keep learning new script languages when new test tools are introduced."

In [Jia+02], the authors proposed an approach of formally specifying test cases by using XML. However, XML, as a kind of text markup language, is inherently insufficient of essential features, or at least not so easy, to specify test modules, such as: (1) the ability of providing profound data types, including user-defined data types (2) flexible flow control (3) timer handling, which is essential for real-time system testing and performance testing (4) dynamic test configurations (5) reuse of test scripts. In addition, the approach lacks the ability to present test scenarios graphically. Therefore, TTCN-3, the only international standard test specification and

implementation language, is more suited to be used to specify ATS in the life-cycle testing process.

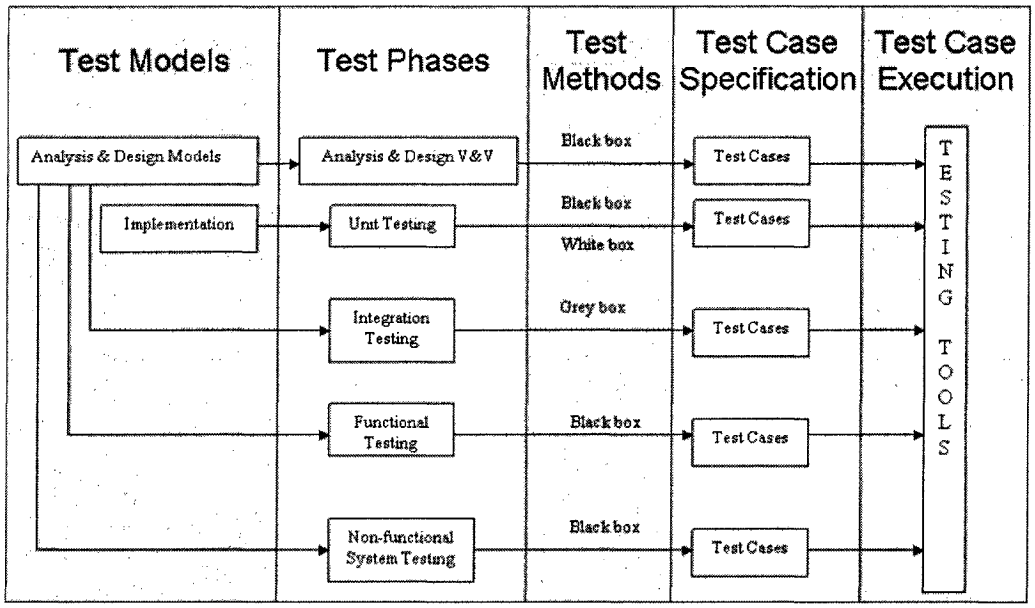

Figure 16 General Life-cycle Testing Process for Web Applications

We improve the life-cycle testing process model shown in Figure 16 by integrating the process with OO-TTCN-3 (see Figure 17. We specify all test cases in OO-TTCN-3 at an abstract level, which is also referred as ATS (Abstract Test Suite). The dash lines in Figure 17 indicate the possible ATS reuse, from the block where the line comes out to the block where the line ends. ATS in OO-TTCN-3 provides a unified interface to test tools. The ATS can be consumed by test tools directly if the test tools support OO-TTCN-3, or via specific OO-TTCN-3 parsers. This separates test scripts from specific test tools, and makes the test tools transparent to testers. Testers can write ATS for a specific e-commerce system, independent of applied test tools. This also eases the ATS reuse between different test phases. For example, test scripts for unit testing can possibly be reused by integration testing without any modification, although different test tools may be used for the unit testing and integration testing.

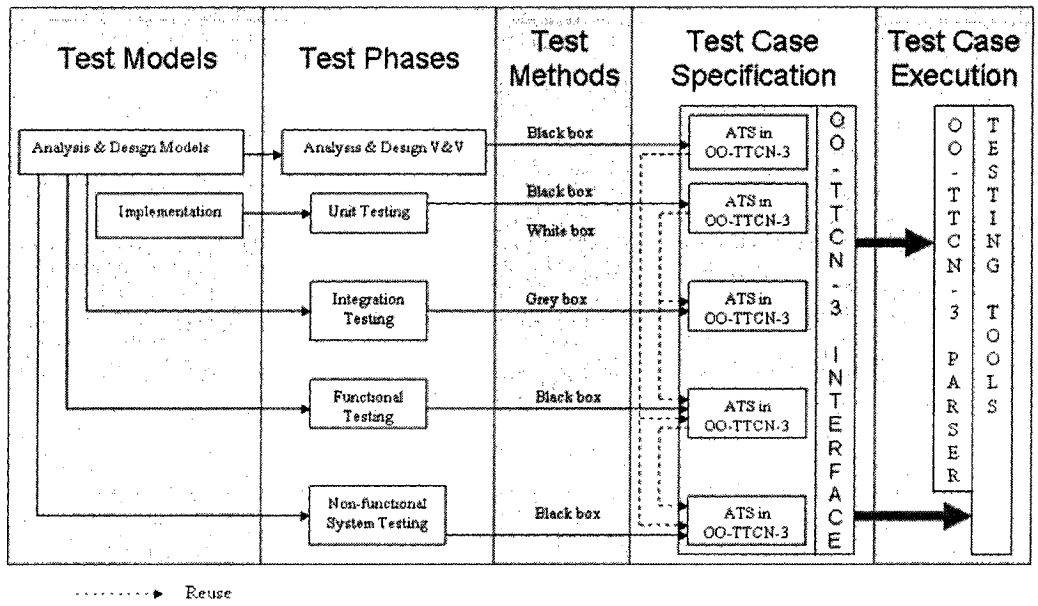

Figure 17 Life-cycle Testing Process with OO-TTCN-3 for Web Applications

6.2 Verify and Validate the Analysis and Design Models

As we have discussed in the previous chapters, conducting the verification and validation of analysis and design models as early as in the analysis and design phases is cost-effective.

Guided inspection, a static testing method which has been discussed in section 3.3.1, is commonly used to verify and validate system analysis and design models, e.g. use cases, state diagrams, class diagrams.

Besides, test scenarios can be derived from requirement models, e.g. use cases, in this phase. Each test scenario is ranked on yield and risk. These test scenarios can be specified in GFT, the graphic format of TTCN-3 which is extended from MSC, and then they can be used to compare with MSCs that have been produced in the development process.

Furthermore, test cases for functional and non-functional system testing can be developed in this phase to achieve 100 percent test scenario coverage.

6.3 Unit Testing

Unit testing is the first test of an executable module. It verifies the code against the component's high-level design and low-level design. White-box testing methods are used at this stage.

In the life-cycle e-commerce testing process, the level of unit testing corresponds to the n-tier architecture of web applications, that is, there are four types of units testing: client tier testing, web tier testing, business tier testing, and EIS tier testing (We will not discuss EIS tier testing in this thesis). The reason of defining the unit testing level corresponding to the n-tier architecture is that each tier of web applications is a relatively independent set of components. The techniques used with these components are similar in the same tier, but may be different in the different tiers. Also, each tier has different responsibilities: the client tier is a presentation tier; the web tier is in charge of receiving http requests, dispatching them and sending back appropriate responses to the client side; the business tier implements specific business logic of web applications; and the EIS tier provides data management. In fact, the scope of unit testing for web applications is extended than that of traditional unit testing, since some integration activities may exist within each tier.

6.3.1 Client Tier Testing

Typical components in the client tier are html files, scripts embedded in an html file, and Java Applets. They run on the client side, usually within the content of a browser. From the functional perspective, common testing activities in the client tier are to validate if every hyperlink in an html file is valid and if scripts and applets act as expected. Some testing models based on the analysis of data flow and control flow of source code, e.g. Block Branch Diagrams and Function Cluster Diagrams introduced in [Kung+00A], can be used to derive test cases for testing scripts. These test cases can be specified in TTCN-3 core language.

6.3.2 Web Server Tier Testing

Components running in the Web tier are JSP files, Java Servlet, and CGI programs. They are also identified by urls, same as html files, but run on the server side. Besides, Servlet and CGI programs may utilize parameters wrapped in the HTTP

request. These components are usually referred as "server pages", while html files are referred as "client pages".

Test modules in TTCN-3 for server pages are similar to those for client pages, but with a significant difference: using procedure-based ports which are based on synchronous communication mechanism to specify procedure call, instead of message-based ports which are based on asynchronous message exchange.

6.3.3 Business Tier Testing

The objects in the business tier are used to implement business logic of web applications. The objects can be represented by class diagrams, probably with constraints in the Object Constraint Language (OCL). Test cases then can be derived from the class diagrams and constraints. Usually, there exist inheritance relationships between these test cases, as we have discussed in section 3.3.2.3. Test modules for specifying these test cases can be specified in OO-TTCN-3, which can handle the inheritance relationship between test modules appropriately.

6.4 Integration Testing

Integration testing is conducted to determine whether or not all components of a software system are working together properly. It is aimed to catch any errors that occur when there are interactions between tested components. Therefore, the integration testing tries to achieve the coverage of the interactions. Grey-box testing methods are used at this stage.

Basically, there are two approaches of integrating web applications: from client tier to EIS tier (top-down), or from EIS tier to client tier (bottom-up). However, no matter which approach is adopted, the ATS defined in Unit testing can be reused at integration testing stage. Object Relation Diagram (ORD) proposed in [Kung+00A] can be utilized to choose test cases and measure coverage.

6.5 Functional Testing

The purpose of functional testing is to ensure the behaviors of a software system meet its functional requirements. It verifies proper execution of the entire set of application components including interfaces to other applications. TTCN can be used for the purpose of specifying functional test cases [Probert+99].

Test scenarios in GFT developed at the system analysis and design phases can be used to generate test cases for functional testing. Actually, a bidirectional mapping between core language and GFT is defined in [ETSI201 873-3], which makes it is possible to generate test script in core language from the scenarios in GFT automatically, or vice-versa, given specific tool supporting.

On the other hand, test modules in TTCN-3 can be built manually. We can specify test cases developed at the system analysis and design phases in TTCN-3. In addition, test scripts produced in unit testing can to be reused in functional testing, as well as in non-functional system testing.

6.6 Non-functional System Testing

Non-functional system testing is the process of testing an integrated software system to verify that the system meets its specified non-functional requirements. Common non-functional tests include scalability testing, reliability testing, availability testing, usability testing, performance testing, compatibility testing and last but no means least – security testing.

Performance testing and security testing are particularly important to web applications. Web applications are often intended to handle thousands of simultaneous users. Multiple users may request different services and gain access to varying functionalities concurrently. It is vital to evaluate a web application's capability to perform critical functions during periods of normal and peak usage. Internet, an open network environment, is the infrastructure on which web applications are running. Therefore, web applications are more vulnerable than traditional applications to malicious attacks. Security testing is to test the effectiveness of the overall web system security defense [Nguyen+03].

Test cases for non-functional system testing, e.g. performance testing, can be specified in TTCN-3. Test scripts developed in the previous testing stages, e.g. functional testing, may be reused for non-functional testing, as we will show in the case study.

6.7 Summary

In this chapter, we introduce a life-cycle testing approach for web applications. The life-cycle testing process parallels the development process. In this process, we need to apply multiple testing methods, build and test multiple models as well as implementation, and utilize multiple testing tools. To facilitate the reusability of test cases in each testing phase, we specify all the test cases at an abstract level with TTCN-3 and its object-oriented extension – OO-TTCN-3. In addition, these abstract test suites are independent of testing tools. In the next chapter, we show how to apply the life-cycle approach to a typical e-commerce system.

Chapter 7 Case Study

In this chapter, a case study is presented to evaluate the life-cycle testing process proposed in the previous chapter. First, we briefly introduce the EX system, a web application that is used as the SUT in our case study. Then we discuss how to test the system at different stages of development life-cycle. We use TTCN-3/OO-TTCN-3 to specify test cases and show how to reuse these test case definitions between different test stages. Our examples mainly focus on the "User Login" use case and associated design models and implementation.

7.1 Overview and Justification of EX System

The EX system is an on-line currency exchange system that acts as a broker to provide customers the best exchange rate between Canadian Dollar (CND) and U.S. Dollar (USD), among its linked banks at remote sites [EXAD].

The entire EX system can be divided into three subsystems: *EX web application subsystem* (EAS), *bank subsystem* (BS), and *postal office subsystem* (PS). EAS obtains quotes from banks and presents the best one to web clients (WC). The customers may accept the best quote and order USD through EAS. In addition, EAS also allows customers to manage their EX account, e.g. viewing account information, viewing trade history, checking order status, modifying and deleting user account. Meanwhile, other two types of users, bank employee (BC) and postal office employee (PC) that reside at the remote sites, can access EX to obtain customers' orders and update the status of the orders by using BS or PS respectively.

We chose the EX system as a case study for several reasons. First, the EX system is a typical e-commerce system that is built on J2EE n-tiered architecture. Second, the development of the EX system sticks with software engineering principles. Complete documents such as use cases, class diagrams and message sequence diagrams are available. This is important to our life-cycle testing approach. Finally, the EX system is available on site with source code. We can conduct unit testing based on the implementation.

In the following sections, we briefly describe the system architecture and the analysis and design models of the EX system.

7.1.1 System Architecture

The EX system is a typical n-tiered distributed system built on J2EE platform, as shown in Figure 18 [EXAD]:

- Client tier: Web browsers are the standard application in this tier. It provides a front-end interface for web clients to interact with EX system to do online trading.
- Web server tier: A web server (WS) runs in this tier. WS functions as an HTTP server and a JSP/Servlet container. As a HTTP server, it directly interacts with client tier, handling HTTP request and response; while as a JSP/Servlet container, it coordinates the request/response with objects in the middle tier by executing JSPs and Servlets.
- Business tier: EX application server (EAS) runs in this tier. The EAS is the core and center of EX system. It implements most of the business logic, and provides services for the WS, BC and PC.
- Data tier: It provides persistent data storage and management.

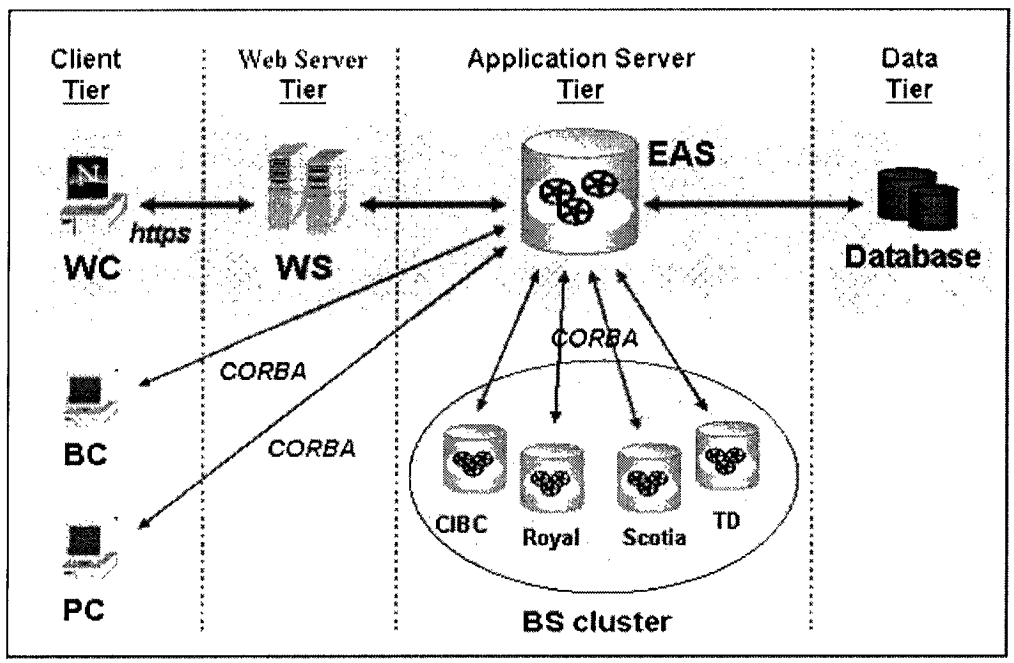

Figure 18 EX System Architecture

7.1.2 Analysis and Design Models

In this section, only parts of the analysis and design models are presented briefly. For the complete and detailed documents, please refer to [EXAD].

7.1.2.1 Use Cases

Use cases are widely used to describe the functional requirements of a system. Table 2 [EXAD] is the list of use cases related to the EAS. Figure 19 [ibid] is the use case diagram for EX system.

Table 2 Use Cases for EAS subsystem

Actors	Web Client (WC), Bank Employee (BC), and Postal Employee (PC)
Use cases	1. User registration 2. User login 3. Check exchange rate 4. Ask quote 5. Submit order 6. Check order status 7. View trade history 8. Modify user account 9. Delete user account 10. User logout 11. Unlock user account

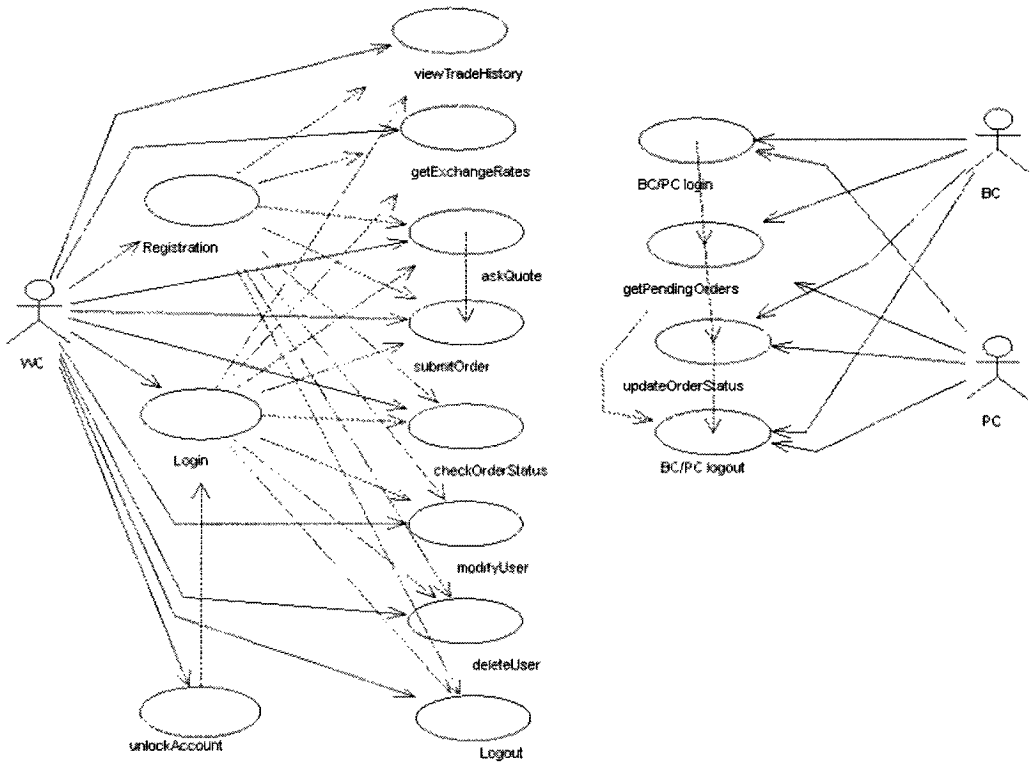

Figure 19 Use Case Diagram for EX system

7.1.2.2 State Diagrams

State diagrams address the dynamic view of a system. Two state diagrams are used to illustrate all possible states and transitions in the EX system. Figure 20 [EXAD] shows the system level state diagram for EX system. Figure 21 [ibid] shows the sub state diagram for the state "EX in Use".

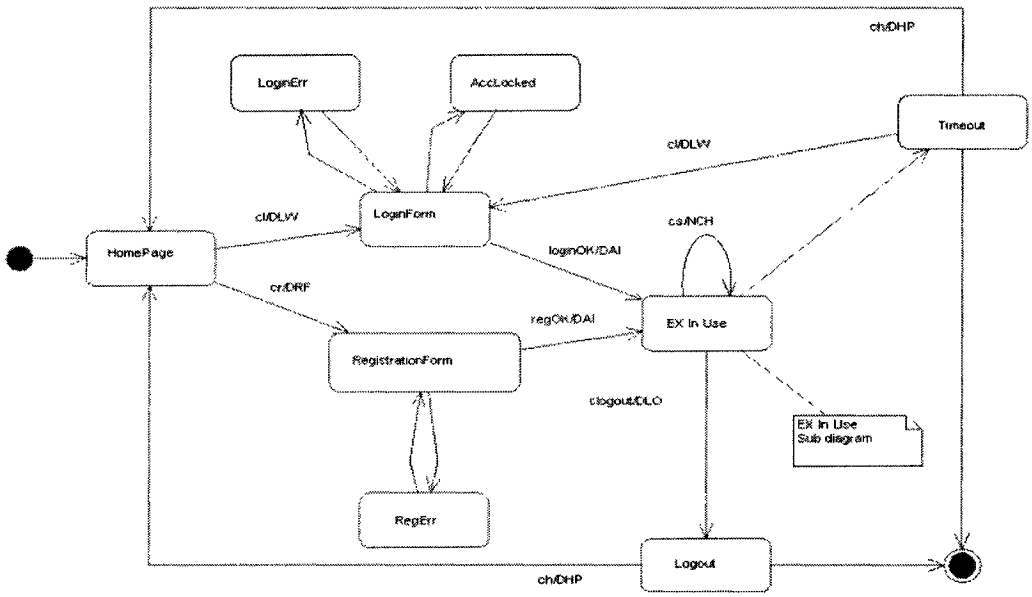

Figure 20 System Level State Diagram for EX System

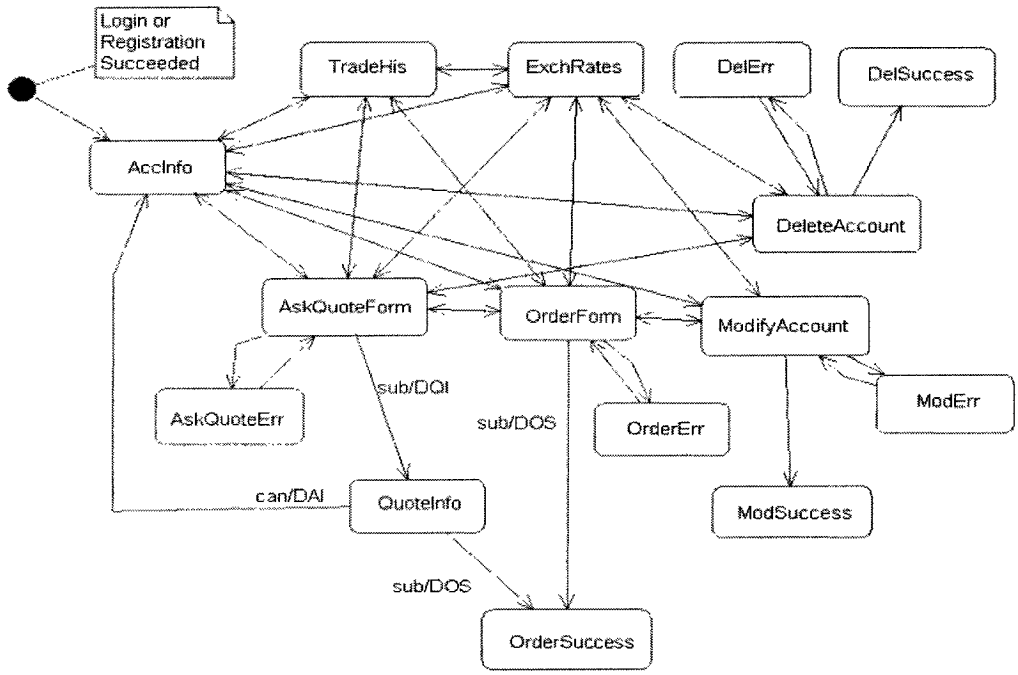

Figure 21 Sub State Diagram for EX in Use

7.1.2.3 Class Diagrams for EAS Subsystem

The EX application server (EAS) is the center of the entire EX system. It directly interacts with every subsystem, including WS, BS, BC and PC. EAS subsystem is divided into four packages: EAS, Includes, EXServer, and BankServer. Each package implements a set of relevant classes as a part of EAS subsystem. One of the detailed

class diagrams for the EAS package is shown in Figure 22 [EXAD]. In section 7.3.3, we will give an example for illustrating how to specify test cases for the inheritance hierarchy between classes Account, EXAccount and BankAccount.

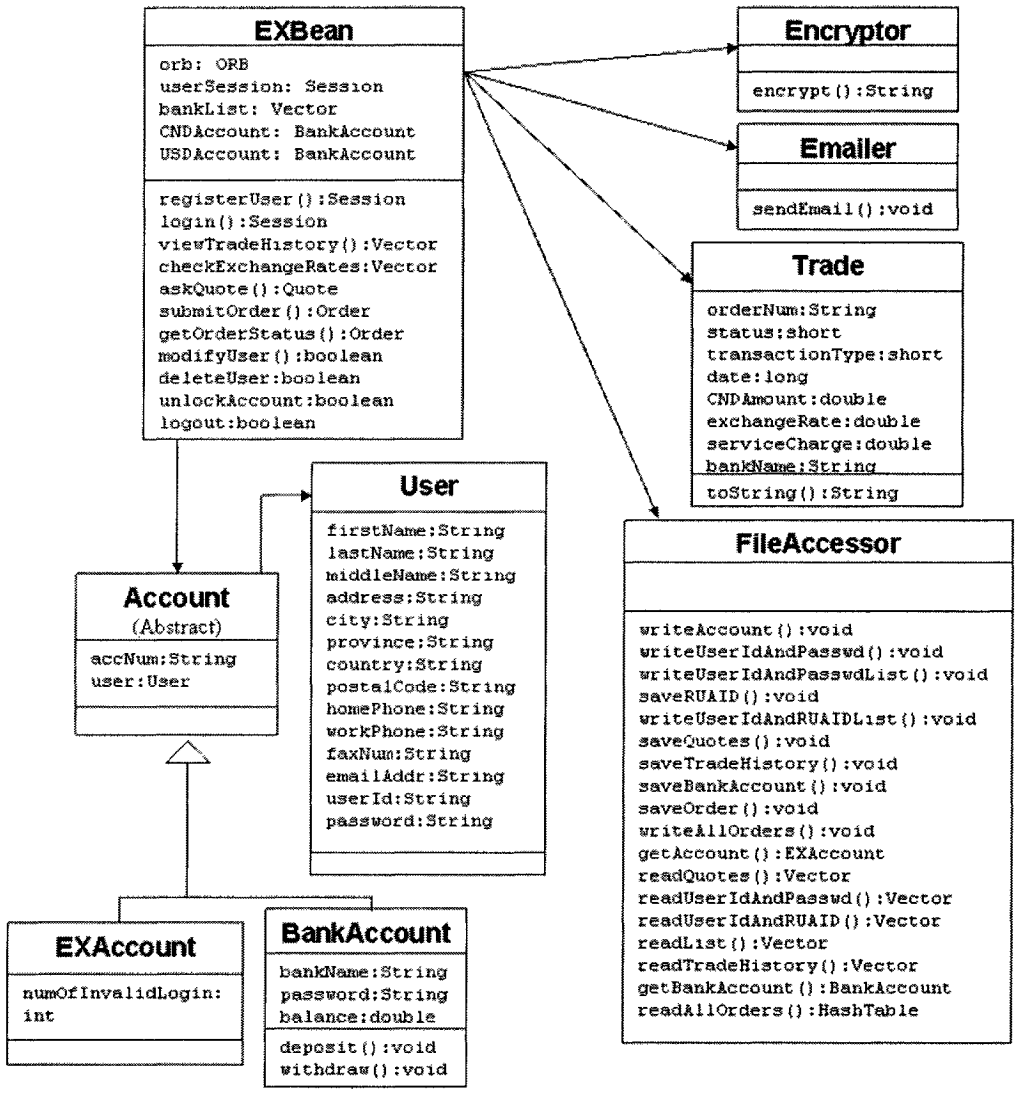

Figure 22 Detailed Class Diagram for EAS Package – Part 1

7.1.2.4 Sequence Diagrams

Package and class diagrams show the static view of a system. Sequence diagram, which shows a set of messages arranged in time sequence [Rumbaugh+99], can be used to illustrate the dynamic view of a system, that is, how objects work together to accomplish the requested functionality. Figure 23 [EXAD]shows the sequence diagram for the "User Login" scenario in which the objects from client tier, web server tier, and middle tier collaborate to accomplish the "login" function.

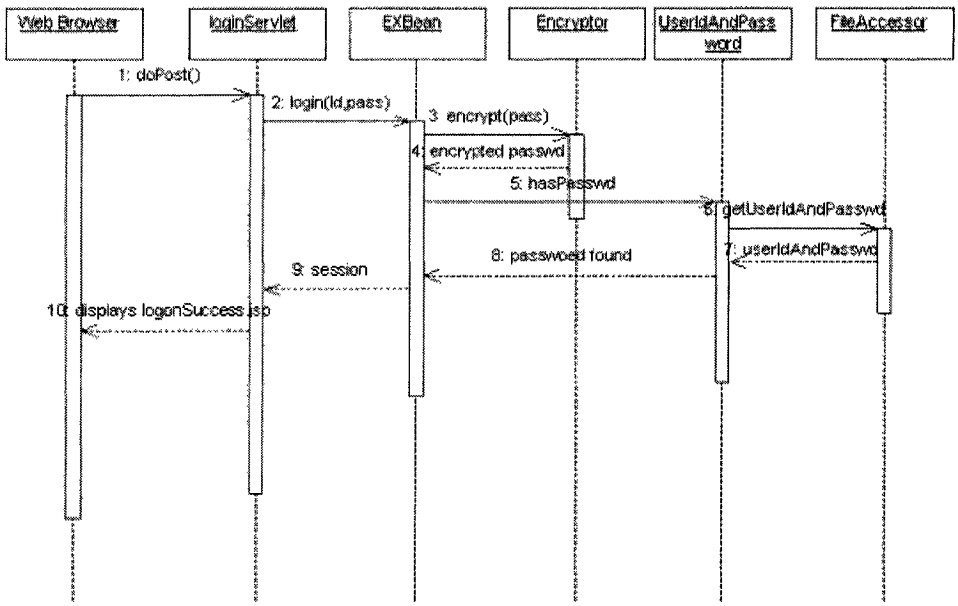

Figure 23 Sequence Diagram for Login

7.2 Validate the Analysis and Design Models

At the analysis and design phases, we should test if the requirement model is correct, complete and consistent, and if the design models conform to the requirement model. In addition, we can prepare test cases in advance for functional system testing. In this section, we discuss how to validate the use case model, and how to develop high-yield and risk-directed test cases for functional testing.

7.2.1 Validate the correctness, completeness and consistency of use cases

Use cases are commonly used to capture system requirements. Table 3 describes the "User Login" use case [EXAD].

Table 3 Use Case – User Login

Goal	User provides user ID and password to access EX system
Precondition	Login page is displayed
Post Condition (success)	Web page with account information is displayed, and a session is created
Post Conditions	1.Web page with login failure message is displayed

(failure)	2. Web page with account locked message is displayed	
Scenario Descriptions		
Success Path	1	USER enters user ID
	2	USER enters password
	3	USER presses Login button
	4	Login success message is displayed
Exception/error Paths	1e	1.e.1. USER doesn't enter user ID. Post condition 1 is presented.
		1.e.2. USER enters invalid user ID. Post condition 1 is presented.
	2e	2.e.1. USER doesn't enter password. Post condition 1 is presented.
		2.e.2. USER enters invalid password. Post condition 1 is presented.
	3e	USER consecutively failed to login 10 times. At 11th time, USER enters valid user ID and password, then presses Login button. Post condition 2 is presented.

A commonly used static method to validate and verify use cases is guided inspection, which is discussed in section 3.3.1. By inspecting the "User Login" use case, we found there are at least four omissions in the use case:

- One more precondition: user account is not locked
- One more post condition: user account is locked
- For the exceptional scenario 2e, we need to specify when a user account is locked.
- For the exceptional scenario 3e, we need to specify that the scenario only occurs if the precondition 2 does not hold.

We revised the use case according to the above analysis. We also modified the exceptional scenario 2e to make it clearer that an empty or invalid password is regarding to a valid user ID. The revised use case is shown in Table 4. The modification is in black and italic style.

Table 4 Use Case – User Login (Revised)

Goal	User provides user ID and password to access EX system	
Preconditions	1. Login page is displayed ***[2. User account is not locked]***	
Post Condition (success)	Web page with account information is displayed, and a session is created	
Post Conditions (failure)	1.Web page with login failure message is displayed 2. Web page with account locked message is displayed ***[3. User account is locked]***	
Scenario Descriptions		
Success Path	1	User enters valid user ID
	2	User enters valid password
	3	User presses Login button
	4	Login success message is displayed
Exception/error Paths	1e	1.e.1. User doesn't enter user ID. The failure post condition 1 is presented.
		1.e.2. User enters invalid user ID. The failure post condition 1 is presented.
	2e	2.e.1. User ***[enters a valid user ID]***, but doesn't enter password. The failure post condition 1 is presented. ***[If the user consecutively failed to login 10 times, the failure post condition 3 is true.]***
		2.e.2. User ***[enters a valid user ID]***, but enters invalid password. The failure post condition 1 is presented. ***[If the user consecutively failed to login 10 times, the failure post condition 3 is true.]***
	3e	3.e.1 ***[When the precondition 2 does not hold],*** user enters valid user ID and password ***(valid or invalid)***, then presses Login button. The failure post condition 2 is presented***; [the failure post condition 3 is true.]***

7.2.2 Derive test scenarios from use cases

Several test scenarios may be derived from one use case. A test traceability matrix is used for tracing test scenarios and test cases from requirements. Table 5 shows eight test scenarios derived from the "User Login" use case. A priority is assigned to each test scenario according to its yield and associated risk. Usually normal scenarios are considered as low-yield and low-risk/medium-risk, and exceptional scenarios are considered as high-yield/medium-yield and high-risk/medium-risk [Probert+00]. The field of "Test Case ID" is reserved for the future use to refer to the test cases which are against the test scenario.

Table 5 Test Traceability Matrix for the "User Login" Use Case (Uncompleted)

Use Case	Test Scenario	Y	R	P	Test Case ID
User Login	**[TS001]** **Pre-conditions:** 1. Login page is displayed 2. User account is not locked **Action:** Enter valid *User ID* and *Password*, click *Login* button **Post-conditions:** Web page with account information is displayed, and a session is created	L	M	3	
	[TS002] **Pre-conditions:** 1. Login page is displayed 2. User account is not locked **Action:** Enter empty *User ID* and arbitrary *Password*, click *Login* button **Post-conditions:** 1.Web page with login failure message is displayed	M	H	2	

	[TS003] **Pre-conditions:** 1. Login page is displayed 2. User account is not locked **Action:** Enter invalid *User ID* and arbitrary *Password*, click *Login* button **Post-conditions:** 1.Web page with login failure message is displayed	M	H	2	
	[TS004] **Pre-conditions:** 1. Login page is displayed 2. User account is not locked **Action:** Enter valid *User ID* and empty *Password*, click *Login* button **Post-conditions:** 1.Web page with login failure message is displayed	M	H	2	
	[TS005] **Pre-conditions:** 1. Login page is displayed 2. User account is not locked **Action:** Enter valid *User ID* and invalid *Password*, click *Login* button **Post-conditions:** 1.Web page with login failure message is displayed	M	H	2	
	[TS006] **Pre-conditions:** 1. Login page is displayed	H	H	1	

	2. User account is not locked **Action:** Enter valid *User ID* and empty/invalid *Password*, click *Login* button. Repeat 10 times **Post-conditions:** 1. Web page with account locked message is displayed 2. User account is locked				
	[TS007] **Pre-conditions:** 1. Login page is displayed 2. User account is **locked** **Action:** Enter valid *User ID* and *Password*, click *Login* button **Post-conditions:** 1. Web page with account locked message is displayed 2. User account is locked	H	H	1	
	[TS008] **Pre-conditions:** 1. Login page is displayed 2. User account is **locked** **Action:** Enter valid *User ID* and invalid *Password*, click *Login* button **Post-conditions:** 1. Web page with account locked message is displayed 2. User account is locked	H	H	1	
Yield (Y) = High/Medium/Low Risk (R) = High/Medium/Low Priority (P) = 1-5, 1 stands for the highest priority and 5 stands for the lowest.					

After we created test scenarios from use cases, we may evaluate the coverage that the test scenarios achieved against a specific design model. For example, activity diagrams can help to analyze coverage.

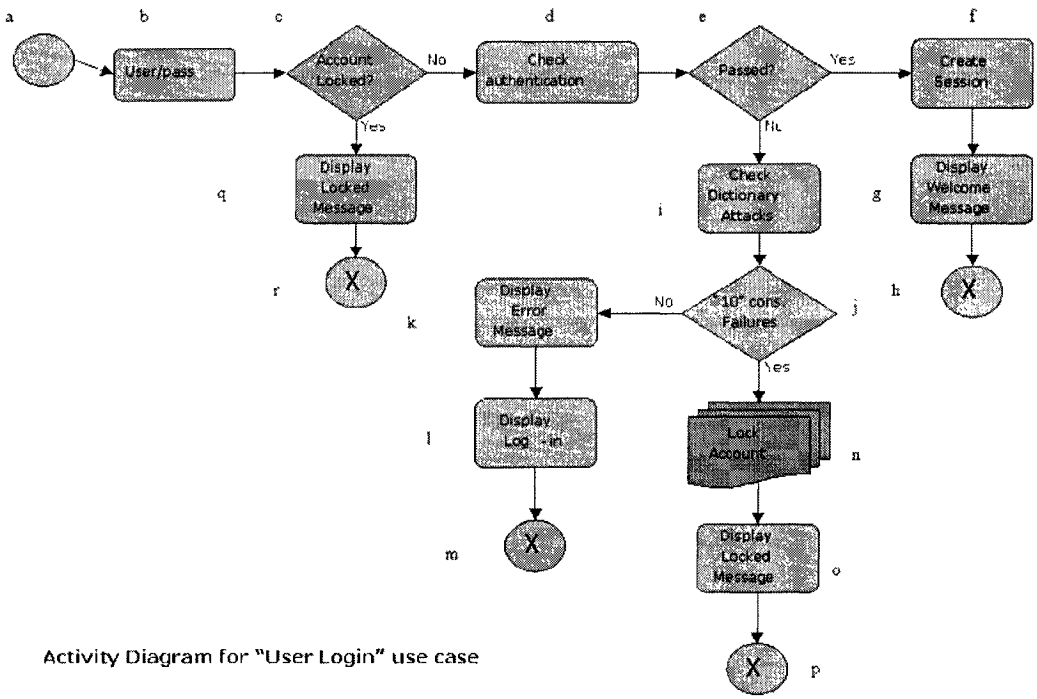

Figure 24 Activity Diagram for "User Login" Use case

Figure 24 is the activity diagram for "User Login" use case [EXTP]. Table 6 shows that the test scenarios cover 100% nodes and edges of the activity diagrams.

Table 6 Test Scenario Coverage

Test Scenarios	Path
TS001	a, b, c, d, e, f, g, h
TS002-TS005	a, b, c, d, e, i, j, k, l, m
TS006	a, b, c, d, e, i, j, n, o, p
TS007	a, b, c, q, r

7.2.3 Specify test scenarios in GFT

The test scenarios can be represented in MSC, which helps to understand the system's behavior in more detail. Then, the MSCs derived from the test scenarios can

be compared with those created in the development process to ensure that the right models are produced in the development process. Such an approach is discussed in [Probert+03].

In our thesis, we would like to use GFT, the Graphical Presentation Format of TTCN-3, to specify the test scenarios. GFT is an extension of MSC with test features. It is more suited than MSC to specify scenarios for testing purpose. Furthermore, since there exists a mapping from TTCN-3 GFT to TTCN-3 core language [ETSI201 873-3], test scenarios in GFT can be converted into core language automatically.

The GFT in Figure 25 specifies the test scenario TS001. The GFT in Figure 26 specifies the test scenario TS002. GFTs that specify test scenarios TS003 to TS006 are similar to that in Figure 26. The GFT in Figure 27 specifies the test scenario TS007, which also shows how to express the "for" statement in GFT. These GFTs can be used to validate that the design models conform to the requirements by comparing them with the MSC in Figure 23.

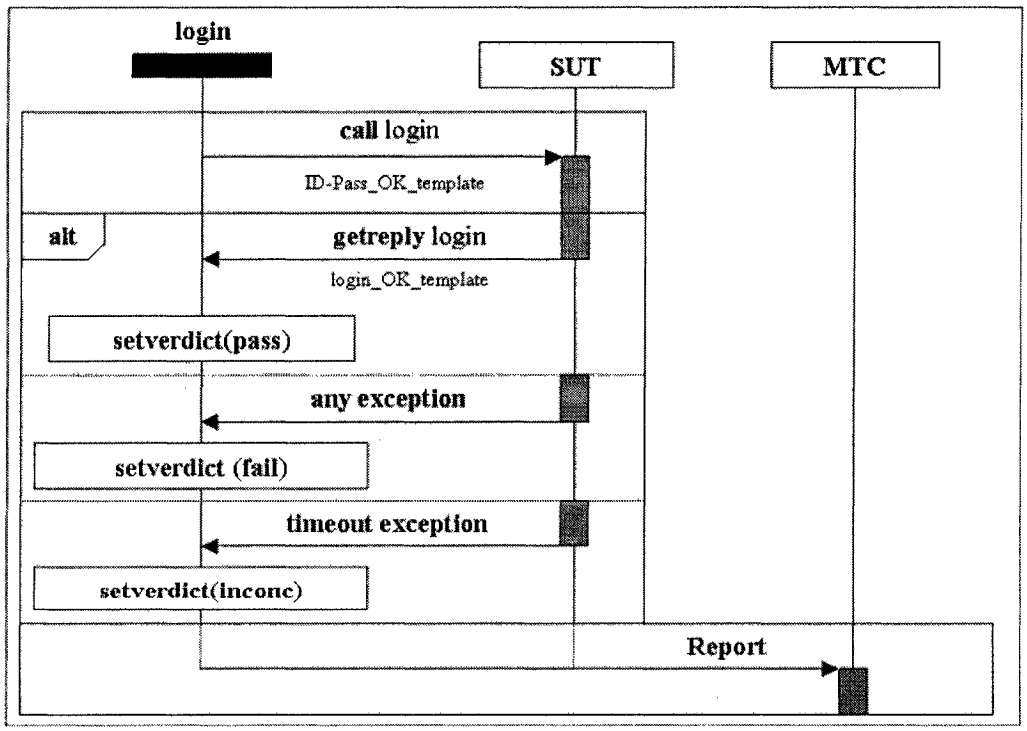

Figure 25 Test Scenario TS001

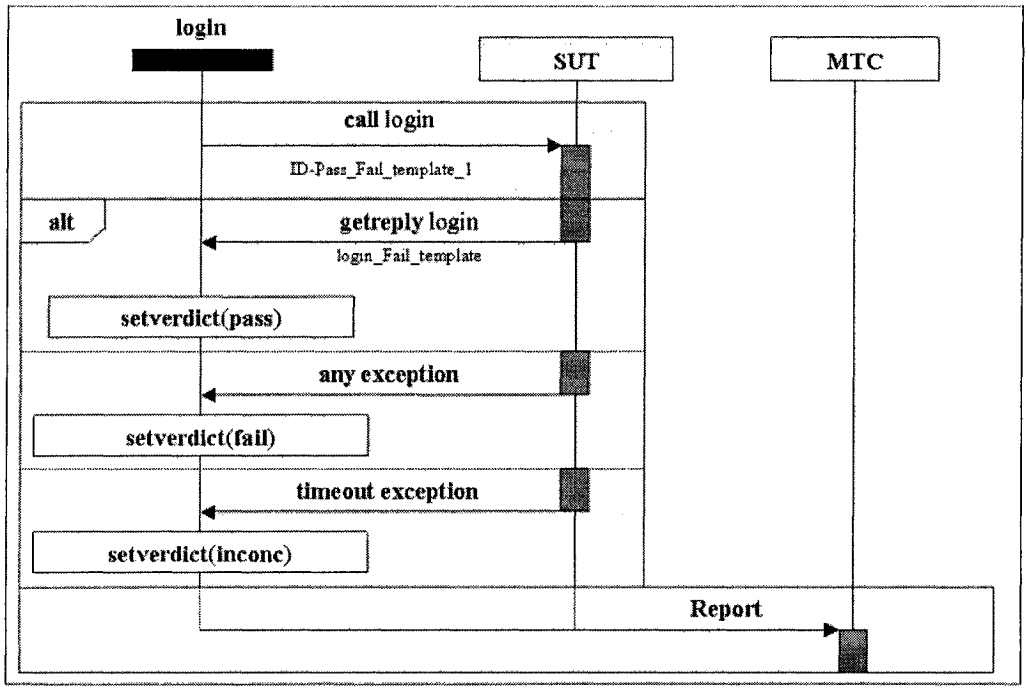

Figure 26 Test Scenario TS002

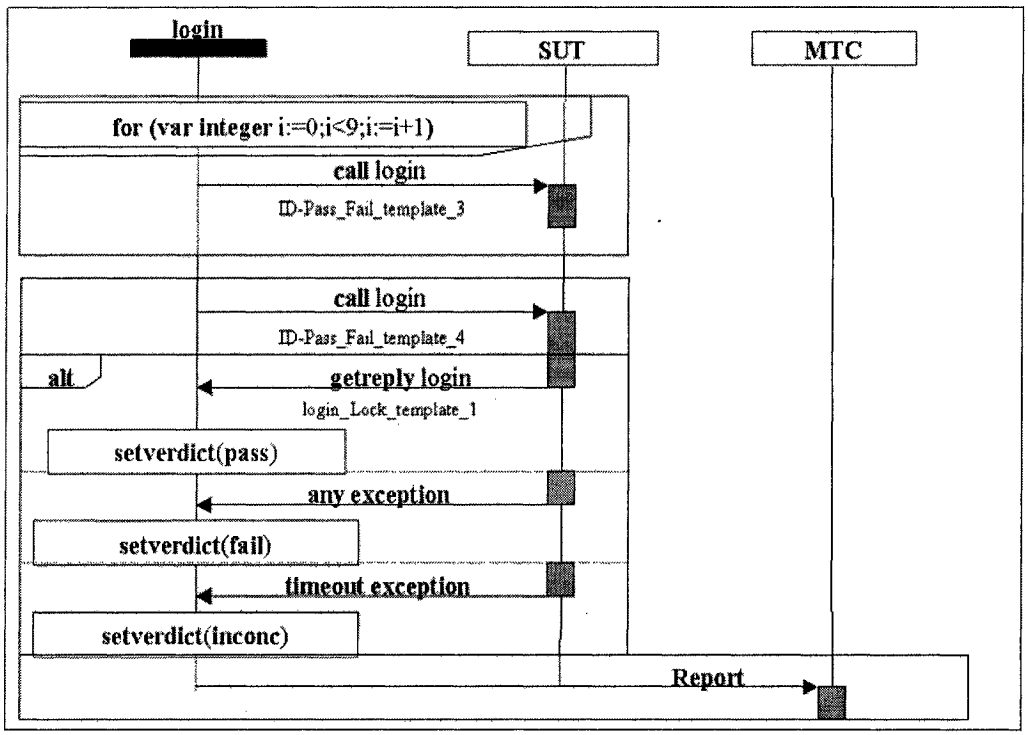

Figure 27 Test Scenario TS007

7.2.4 Develop test cases

After GUI design has been done, we can develop test cases from test scenarios and the GUI design. The development of test cases consists of three basic steps:

1. Equivalence-class partitioning
2. Boundary-value analysis
3. Design test cases to cover all test scenarios

Figure 28 is the screenshot of "Login" web page. The related non-functional requirements for the input "User ID" and "Password" are as follows [Probert03]:

User ID (mandatory): 10 characters maximum, 4 characters minimum, no special character except email-address-like characters are allowed. The email-address-like characters include letters, digits and {@, _}.

Password (mandatory): 6 characters maximum, at least one digit, but the first and the last characters can not be digits.

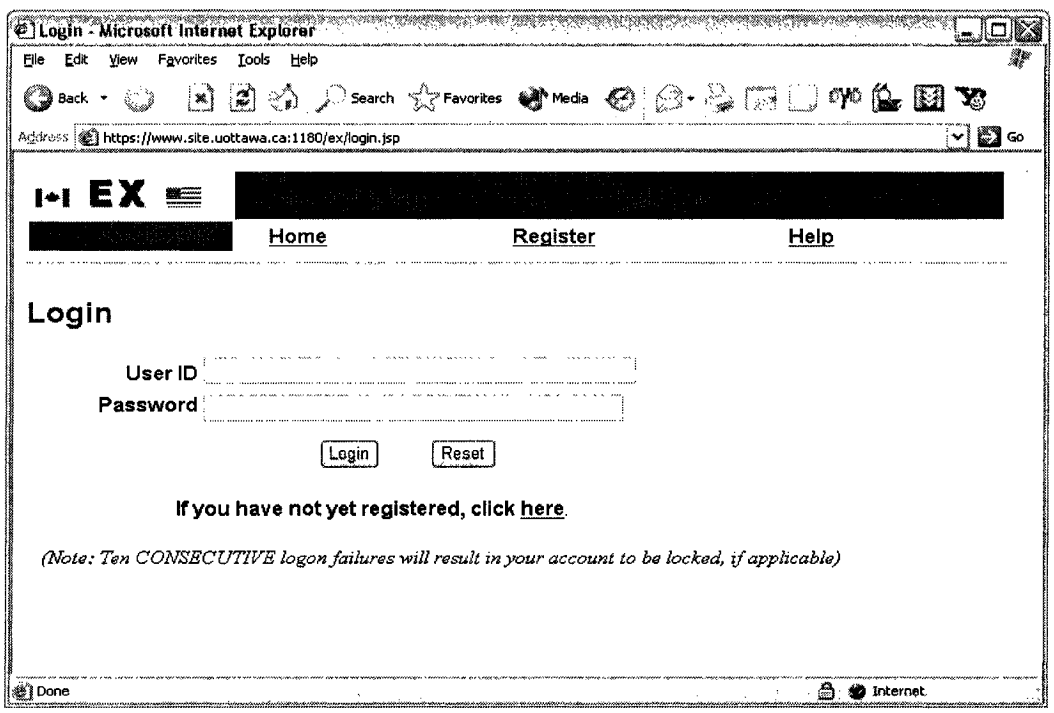

Figure 28 Login Web Page

At the first step, we partitioned each input condition into valid and invalid equivalence classes (see Table 7).

Table 7 Equivalence Classes of User ID and Password

GUI Name	Input Condition	Valid E.C.		Invalid E.C.	
Login Web Page	User ID	Length <= 10	1	Length > 10	2
		Length >= 4	3	Length < 4	4
		Contains email-like-chars	5	Contains non-email-like-chars	6
		Present	7	Absent	8
	Password	Length <= 6	9	Length > 6	10
		At least one digit	11	No digit	12
		First char. is non digit	13	First char. is digit	14
		Last char. is non digit	15	Last char. is digit	16
		Present	17	Absent	18

Table 8 Test Data to Cover the E.C.s (Uncompleted)

No.	Test Data		E.C. Covered	Test Case ID
	User ID	Password		
TD001	C@12	P1234P	3,5,7,9,13,15,17	
TD002	C_12345678	P1P	1,5,7,11,13,15,17	
TD003	C@123456789	P1234P	2	
TD004	C@1	P1234P	4	
TD005	C*12	P1234P	6	
TD006		P1234P	8	
TD007	C@12	P12345P	10	
TD008	C@12	P	12	
TD009	C@12	1P	14	
TD010	C@12	P1	16	
TD011	C@12		18	

Second step, we created a set of test data to cover these equivalence classes based on boundary-value analysis, as shown in Table 8. The test data cover as many valid equivalence classes as possible, or cover one and only one invalid equivalence class.

The field of "Test Case ID" is reserved for the future use to refer to the test cases which utilize the test data.

At the third steps, we designed test cases (see Table 9) to cover all the test scenarios defined in Table 5, and try to utilize test data constructed in Table 8.

Table 9 Test Cases

Test Case ID	Test Case Description					
	Test Setup	Input		Expected Output	Actual Output	P/F
		User ID	Password			
TC001	Login page is displayed, C@12 is a valid user ID and is not locked, P1234P is the valid password associated with User ID "C@12"	C@12	P1234P	Web page with account information is displayed, a session is created		
TC002	Login page is displayed		P1234P	Web page with login failure message is displayed		
TC003	Login page is displayed, C@123456789 is a invalid user ID	C@123456789	P1234P	Web page with login failure message is displayed		
TC004	Login page is displayed, C@12 is a valid user ID and is not locked, P1234P is the valid password associated with User ID "C@12"	C@12		Web page with login failure message is displayed		
TC005	Login page is displayed, C@12 is a valid user ID and is not locked, P1234P is the valid password associated with User ID "C@12"	C@12	P12345P	Web page with login failure message is displayed		

Test Case ID	Test Case Description					
	Test Setup	Input		Expected Output	Actual Output	P/F
		User ID	Password			
TC006	Login page is displayed, C@12 is a valid user ID and is not locked, P1234P is the valid password associated with User ID "C@12"; have tried to login with "C@12" and arbitrary invalid password for 9 times	C@12	P	Web page with account locked message is displayed, User account is locked		
TC007	Login page is displayed, C@12 is a valid user ID but is locked, P1234P is the valid password associated with User ID "C@12"	C@12	P1234P	Web page with account locked message is displayed, User account is locked		
TC008	Login page is displayed, C@12 is a valid user ID but is locked, P1234P is the valid password associated with User ID "C@12"	C@12	P	Web page with account locked message is displayed, User account is locked		

Finally, we completed the test traceability matrix (Table 5) and the test data table (Table 8) after we worked out the test cases, as shown in Table 10 and Table 11, respectively. This helps to analyze test case coverage. From the Table 10, we can see that the "User Login" use case is 100 percent covered by the test cases. From the Table 11, however, we can see that the test data are not completely covered by the test cases. Actually, the test data may be covered more by test cases for the "User Registration" use case.

Table 10 Test Traceability Matrix for the "User Login" Use Case (Completed)

Use Case	Test Scenario	Y	R	P	Test Case
User Login	**[TS001]** **Pre-conditions:** 1. Login page is displayed 2. User account is not locked **Action:** Enter valid *User ID* and *Password*, click *Login* button **Post-conditions:** Web page with account information is displayed, and a session is created	L	M	3	TC001
	[TS002] **Pre-conditions:** 1. Login page is displayed 2. User account is not locked **Action:** Enter empty *User ID* and arbitrary *Password*, click *Login* button **Post-conditions:** 1.Web page with login failure message is displayed	M	H	2	TC002
	[TS003] **Pre-conditions:** 1. Login page is displayed 2. User account is not locked **Action:** Enter invalid *User ID* and arbitrary *Password*, click *Login* button **Post-conditions:** 1.Web page with login failure message is displayed	M	H	2	TC003

	[TS004] **Pre-conditions:** 1. Login page is displayed 2. User account is not locked **Action:** Enter valid *User ID* and empty *Password*, click *Login* button **Post-conditions:** 1.Web page with login failure message is displayed	M	H	2	TC004
	[TS005] **Pre-conditions:** 1. Login page is displayed 2. User account is not locked **Action:** Enter valid *User ID* and invalid *Password*, click *Login* button **Post-conditions:** 1.Web page with login failure message is displayed	M	H	2	TC005
	[TS006] **Pre-conditions:** 1. Login page is displayed 2. User account is not locked **Action:** Enter valid *User ID* and empty/invalid *Password*, click *Login* button. Repeat 10 times **Post-conditions:** 1. Web page with account locked message is displayed 2. User account is locked	H	H	1	TC006
	[TS007] **Pre-conditions:**	H	H	1	TC007

	1. Login page is displayed 2. User account is **locked** **Action:** Enter valid *User ID* and *Password*, click *Login* button **Post-conditions:** 1. Web page with account locked message is displayed 2. User account is locked				
	[TS008] **Pre-conditions:** 1. Login page is displayed 2. User account is **locked** **Action:** Enter valid *User ID* and invalid *Password*, click *Login* button **Post-conditions:** 1. Web page with account locked message is displayed 2. User account is locked	H	H	1	TC008
Yield (Y) = High/Medium/Low Risk (R) = High/Medium/Low Priority (P) = 1-5, 1 stands for the highest priority and 5 stands for the lowest.					

Table 11 Test Data to Cover the E.C.s (Completed)

No.	Test Data		E.C. Covered	Test Case ID
	User ID	Password		
TD001	C@12	P1234P	3,5,7,9,13,15,17	TC001
TD002	C_12345678	P1P	1,5,7,11,13,15,17	
TD003	C@123456789	P1234P	2	TC003
TD004	C@1	P1234P	4	
TD005	C*12	P1234P	6	
TD006		P1234P	8	TC002

No.	Test Data		E.C. Covered	Test Case ID
	User ID	Password		
TD007	C@12	P12345P	10	TC005
TD008	C@12	P	12	TC006
TD009	C@12	1P	14	
TD010	C@12	P1	16	
TD011	C@12		18	TC004

7.3 Unit Testing

In the life-cycle e-commerce testing process, the unit testing is performed separately on the four tiers: the client tier test, the web tier test, the business (middle) tier test, and the EIS tier test. We will not discuss the EIS test in this thesis.

7.3.1 Client Tier Test

The purpose of the client tier test is to check if every hyperlink is valid and if scripts and applets act as expected. The following TTCN-3 script demonstrates how to check the validity of a hyperlink.

```
module hyperlinkCheck{
modulepar {charstring testtype};
//definition part
type record url {charstring protocol, charstring host, charstring portNum,
charstring path}
template url hyperlink_index := {protocol:="https://",
host:="www.site.uottawa.ca", portNum:=":1180", path:="ex/index.html"}
template charstring status_ok := "200";
template charstring status_timeout := "408";
type port hyperlinkPortType message {out url; in charstring}
type port mCPType message {in verdicttype}
type port pCPType message {out verdicttype}
type component hyperlinkComponent {port hyperlinkPortType hyperlinkPort;
port pCPType CP}
type component mtcComponent {port mCPType CP; port hyperlinkPortType
hyperlinkPort; var integer activePTCs := 0;}
type component TSI {port hyperlinkPortType hyperlinkTSI}
```

```
//behavior of hyperlinkComponent
function hyperlink_check (in url hyperlink, mtcComponent theSystem) runs on hyperlinkComponent {
  map(self:hyperlinkPort, theSystem:hyperlinkPort);
  hyperlinkPort.send(hyperlink);
  alt {
    [] hyperlinkPort.receive(status_ok) {setverdict(pass)}
    [] hyperlinkPort.receive(status_timeout) {setverdict(inconc)}
    [] hyperlinkPort.receive() {setverdict(fail)}
  }
  CP.send(getverdict);
}
//definition of test case
testcase hyperlink_tc(in url hyperlink, integer loops, out integer passedTCs, integer failedTCs, integer inconcTCs) runs on mtcComponent system mtcComponent {
  var integer i;
  var verdicttype theVerdict;
  var hyperlinkComponent theNewPTC[loops];
  for (i:=1;i<=loops;i:=i+1)
  {
   theNewPTC[i]:= hyperlinkComponent.create;
   activePTCs := activePTCs + 1;
   connect(mtc:CP, theNewPTC[i]:CP);
   theNewPTC[i].start(hyperlink_check(hyperlink, system));
  }
  while (activePTCs > 0) {
    CP.receive(verdicttype:?)-> value theVerdict;
    activePTC := activePTC - 1;
    setverdict(theVerdict);
    if (theVerdict == pass) { passedTCs := passedTCs + 1; }
  else if (theVerdict == fail) { failedTCs := failedTCs + 1; }
  else if (theVerdict == inconc) { inconcTCs := inconcTCs + 1; }
  }
  all component.done;
 }
 function basicFunctionality() return verdicttype {
  var verdicttype localVerdict;
```

```
  var integer nrP := 0, nrF := 0, nrI := 0;
  localVerdict := execute(hyperlink_tc(hyperlink_index,1,nrP,nrF,nrI));
  return localVerdict;
 }
 //control part
 control {
  var verdicttype overallVerdict;
  if (testtype == "basicFunctionality") {
   overallVerdict := basicFunctionality();
  }
 }
 }
```

7.3.2 Web Tier Test

Test modules in TTCN-3 for server pages are similar to those for client pages, except they use procedure-based ports, instead of message-based ports, to communicate with SUT. The following is a piece of code showing the test of "Login Servlet" by using a procedure-based port:

```
module LoginServletTest(){
//import type url from hyperlinkCheck;
type record url {charstring protocol, charstring host, charstring
portNum, charstring path};
type enumerated reasonType {invalidID, invalidpassword};
signature loginServlet(in url url_loginServlet, charstring ID, charstring
password) return boolean exception (reasonType);
template loginServlet validLogin := {url_loginServlet := url_login_template,
ID := "C001", password := "C001"}
template url url_login_template := {protocol:="https://",
host:="www.site.uottawa.ca", portNum:=":1180", path:="ex/loginServlet"}
type port loginPortType procedure {out loginServlet}
type component mtcType { port loginPortType loginPort;}
testcase validLogin_tc() runs on mtcType system mtcType {
 var reasonType theReason;
 loginPort.call(loginServlet:validLogin);
 alt {
  [] loginPort.getreply(loginServlet:{?,?,?} value true) {setverdict(pass)}
  [] loginPort.catch(loginServlet,reasonType:?) -> value theReason
{setverdict(fail)}
 }
```

```
}
}
```

7.3.3 Business Tier Test

Test modules for testing objects in the business tier can be specified in OO-TTCN-3, which can appropriately describe the inheritance relationship between test modules.

In EX system, "Account" is an abstract class (see Figure 14 and Figure 22). It contains two attributes: accNum and user. One of the methods defined and implemented in the class "Account" is getAccountNo(), which returns the account number of the current user. Two subclasses, "BankAccount" and "EXAccount", are derived from "Account". "BankAccount" is used to describe the attributes and behaviors of bank accounts. "EXAccount" is used to describe the accounts in the EX system. The signature and implementation of getAccountNo() are not changed in the two derived classes. Therefore, test cases developed for getAccountNo() can be reused by the test modules for "BankAccount" and "ExAccount". In addition, we only need to run the test suites once, either in the test module for "BankAccount" or in the module for "EXAccount", but to validate the method in the three classes. It also avoids testing the abstract class "Account", which can not be instantiated and is difficult to be tested directly. The following is piece of code which shows how to specify the test modules in OO-TTCN-3:

```
module AccountTest {
signature Acc_constr (in charstring AccNum, charstring user) exception
(charstring);
signature getAccountNo () return charstring exception (charstring);
template Acc_constr AccValue := {AccNum := "1234567890", user := "bob"}
type port AccountPortType procedure {out Acc_constr}
type port getAccountNoPortType procedure {out getAccountNo}
type component mtcType {port AccountPortType AccPort; port
getAccountNoPortType getAccPort}
testcase getAccountNo_tc() runs on mtcType system mtcType{
 var charstring theResult;
 AccPort.call(Acc_constr: AccValue);
 alt {
  [] AccPort.getreply (Acc_constr:{?,?}) {setverdict(pass)}
  [] AccPort.catch() {setverdict(inconc); return}
 }
```

```
  getAccPort.call(getAccountNo);
  alt {
    [] getAccPort.getreply (getAccountNo value "1234567890")
{setverdict(pass)}
    [] getAccPort.catch(getAccountNo, charstring:?) -> value theResult
{setverdict(fail)}
  }
}
}
module BankAccountTest() extends AccountTest {
control {execute(getAccountNo_tc());}
}
```

7.4 Integration Testing

The purpose of integration testing is to make sure all components of a software system work together properly. ATS defined for the unit testing can be reused directly for integration testing. Figure 29 is the partial ORD for EX system. Test module "LoginServletTest" defined for web tier testing is ready for integration testing, which covers components *login server page, LoginServlet servlet, logonSuccess server page,* and *EXBean*, and interactions between these components.

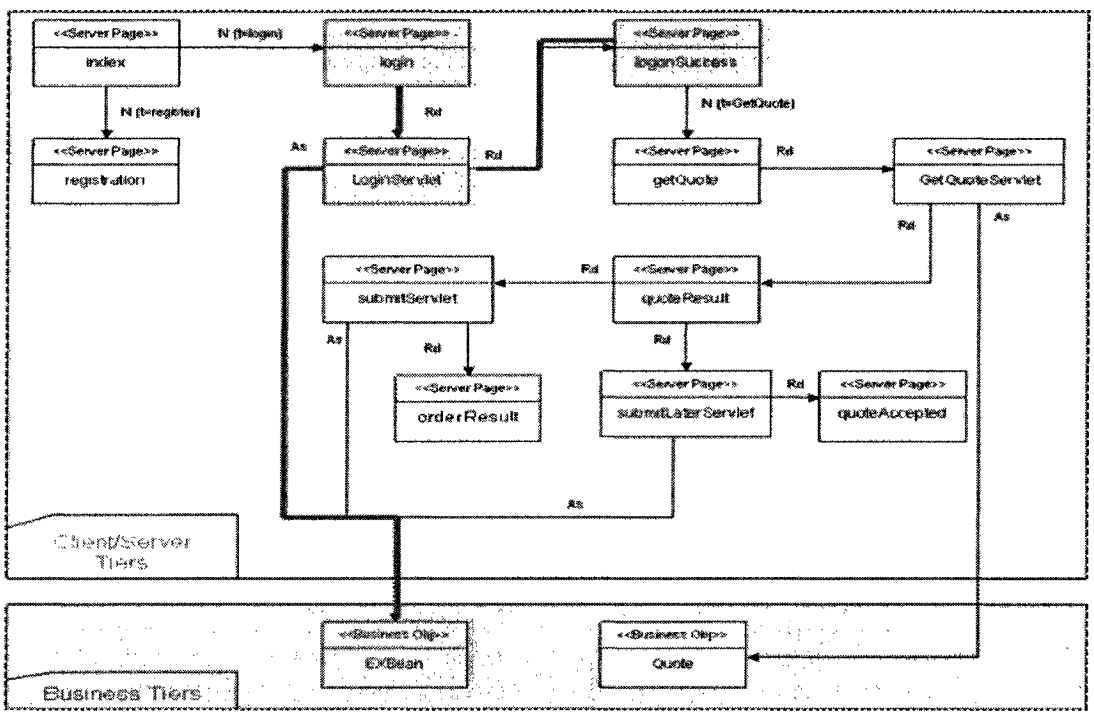

Figure 29 Object Relation Diagram for EAS Subsystem (Part)

7.5 Functional Testing

Test scripts for unit testing can be used to construct test modules for functional testing. The following is a piece of code to test "User Login" functionality, which utilizes part of definitions from test modules of "hyperlinkCheck" and "LoginServlettest".

```
module UserLoginTest {
import from hyperlinkCheck all;
import from LoginServletTest all;
function validLogin() return verdicttype {
 var verdicttype localVerdict;
 localVerdict :=
execute(hyperlinkCheck.hyperlink_tc(hyperlink_index,1,0,0,0));
 if (localVerdict != pass) {return localVerdict;}
 //template hyperlink_login need to be defined in the test module
"hyperlinkCheck"
localVerdict := execute(hyperlinkCheck.hyperlink_tc(hyperlink_login,1,0,0,0));
 if (localVerdict != pass) {return localVerdict;}
 localVerdict := execute(LoginServletTest.validLogin_tc());
 if (localVerdict != pass) {return localVerdict;}
 //template hyperlink_logout need to be defined in the test module
"hyperlinkCheck"
 localVerdict :=
execute(hyperlinkCheck.hyperlink_tc(hyperlink_logout,1,0,0,0));
 return localVerdict;
}
}
```

7.6 System Testing

By reusing definition in test modules for unit and functional testing, creating tests for system testing becomes simple. The following is an example of performance testing: adding a function in the "hyperlinkCheck" test module to simulate 1000 times of click on "index.html", and then observe how many requests are timeout or failure:

```
module hyperlinkCheck {
function performanceTesting() return verdicttype {
 var verdicttype localVerdict;
 var integer nrP := 0, nrF := 0, nrI := 0;
 localVerdict := execute(hyperlink_tc(hyperlink_index,1000,nrP,nrF,nrI));
```

```
    if ((nrF + nrI) > 5) {localVerdict := fail;}
    else { localVerdict := pass;}
    return localVerdict;
  }
  }
```

7.7 Preliminary Evaluation of Approach

Through the case study, we demonstrate the life-cycle testing process is feasible for web application testing. Test cases in the life-cycle process can be specified on the abstract level with OO-TTCN-3, and more important, the ATS can be reused between different test phases.

Compared with non-life-cycle testing methods for web applications, the proposed life-cycle e-commerce testing approach brings the following benefits:

- Tests web applications at each development phase to find and fix errors as early as possible, which is a well-proven cost-effective approach.
- Increases the reusability of test cases by specifying the test cases at an abstract level. The ATS can be reused between different test phases, and they can be implemented on different platforms in different programming languages since the ATS is independent of platforms and languages.
- Provides a unified test case interface to testing tools. This should facilitate more support from tool vendors.

As presented in section 1.2.2, other approaches to testing do not provide all of the benefits.

However, further development of the life-cycle test methodology should be done. For example, ATS in TTCN-3 can not be executed directly. It needs to be compiled into code in specific programming languages, e.g. Java or C, which then can be executed against a SUT. Several commercial TTCN-3 compilers are available. We have introduced them briefly in section 5.3. In addition, Software components such as Platform Adapter (PA), SUT Adapter (SA) [ETSI201 873-5] and Test Management (TM) [ETSI201 873-6] may also need to be developed in specific programming languages to realize test execution. A general structure of a TTCN-3 test system is shown in Figure 30.The TTCN-3 Runtime Interface (TRI) and the TTCN-3 Control Interface (TCI) in Java and C for these components have been standardized in [ETSI201 873-5] and [ETSI201 873-6].

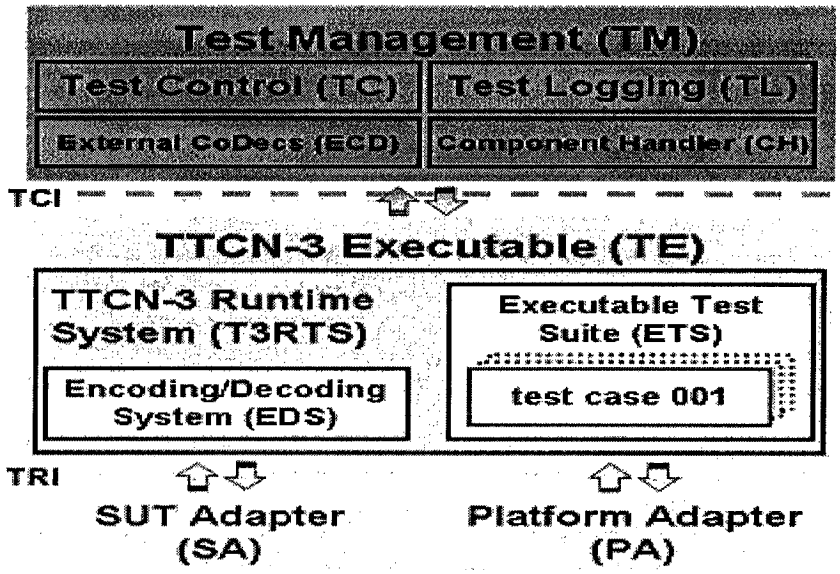

Figure 30 General Structure of a TTCN-3 Test System

Testing tasks in the life-cycle testing process may be distributed to different teams. For example, the verification and validation of system analysis and design are performed by both development and test teams, the unit and integration testing are performed by the development team, and the functional and system testing are performed by the test team. Therefore, both testers and developers need to use TTCN-3 and a specific programming language such as Java and C to specify and implement test cases.

However, the cost of developing and executing test cases in the life-cycle testing process can be reduced for the following reasons. First, since TTCN-3 is quite similar to a programming language e.g. Java, it is not difficult for both testers and developers to learn and use it. Second, specifying ATS in TTCN-3 is also simpler than writing concrete test cases in specific programming languages because quite lots of platform dependent and language dependent details are abstracted off in ATS. Finally, although PA, SA and TM Components may need some programming work, the reusability of these components have been improved than before since the interfaces of these components are standardized.

7.8 Summary

Through the case study in this chapter we illustrate how to apply the life-cycle test process to a typical web application. We see that as early as at the analysis and design

phases, we can verify and validate the analysis and design models, and create test cases ready for functional testing. In addition, although that is out of the scope of this thesis, it is worth to mention that test cases for non-functional system testing, which can be derived from non-functional requirements, can also be developed at the analysis and design phases. We specified all test cases at abstract level using TTCN-3/OO-TTCN-3, and showed how the approach increased the reusability of the test cases.

Chapter 8 Conclusions and Future Work

8.1 Conclusion

In this thesis, we proposed a life-cycle testing process with object-oriented TTCN-3 for e-commerce systems. We illustrated how to apply the process to e-commerce systems via a case study. In this case study, a life-cycle testing process on a typical e-commerce system is achieved, and the test cases for every testing phase are specified in OO-TTCN-3 on an abstract level. It's demonstrated that the life-cycle testing process is feasible for e-commerce testing, OO-TTCN-3 is applicable of specifying ATS at every testing phase, and more important, the ATS can be reused between test phases.

As part of the results of this thesis research, a fast abstract, "A Multi-Method Testing Approach in TTCN-3 for Web Applications" [Xiong+03], and a paper, "Life-cycle E-Commerce Testing with OO-TTCN-3" [Probert+04], have been accepted by ISSRE 2003 and TheFormEMC, respectively.

8.2 Contributions of the Thesis

The contributions of the thesis can be summarized as follows:

- Proposed a life-cycle e-commerce testing process which is adapted to the n-tiered architecture of web applications.
- Proposed an object-oriented extension to TTCN-3. The extension makes TTCN-3 capable of specifying object-oriented test cases, and therefore applicable to the complete life-cycle process.
- Illustrated how to integrate the life-cycle testing process by specifying ATS with OO-TTCN-3 at every phase of the life-cycle testing process. Specifying the ATS in OO-TTCN-3 improves the reusability of test cases and provides a unified ATS interface to testing tools.
- Illustrated how to apply the life-cycle testing process to e-commerce testing via a case study.

8.3 Future Work

Making a complete object-oriented extension to TTCN-3 would be quite complex. In this thesis we present a preliminary such extension. The extension only focuses on the mechanisms of inheritance and aggregation. However, as software systems are evolving on scale and complexity, testing systems have become more and more complex. It may be worthwhile to investigate other object-oriented natures existing in the testing systems, such as polymorphism. A more object-oriented testing language may improve the reusability and maintainability of testing systems, just like the benefits brought by object-oriented techniques to software development.

Tool support is critical to software testing. There are several TTCN-3 compilers that compile ATS in TTCN-3 into code in a programming language such as Java. In [Probert+04], we used TTthree to compile and execute some test modules in TTCN-3. A prototype tool that supports OO-TTCN-3 needed to be considered in the future work. Furthermore, a tool supporting test cases management may facilitate the reuse of test cases.

Another future work may be conducting more empirical studies to evaluate the efficiency of the life-cycle testing approach. For example, metrics in the Personal Software Process (PSP) [Humphrey97] may be applied to measure the productivity of the approach.

Even though much more study and empirical investigation is important, this thesis has suggested some important directions for follow-up and taken a first step in the search for more cost-effective techniques for testing e-commerce systems and web applications.

References

[Baresi+01] Luciano Baresi, Franca Garzotto, and Paolo Paolini, Extending UML for Modeling Web Applications, *Proc. of 34th Hawaii Int'l Conf. on System Sciences*: pp. 1285-1294, 2001

[Bashir+99] Imran Bashir and Amrit L. Goel, *Testing Object-Oriented Software: Life-Cycle Solutions*, Springer, 1999

[Beizer90] Boris Beizer, *Software Testing Techniques*, 2nd Edition, Van Nostrand Reinhold, 1990

[Benett+02] Simon Benett, Steve McRobb and Ray Farmer, *Object Oriented Systems Analysis and Design Using UML*, 2nd Edition, McGraw Hill, 2002

[Binder00] Robert V. Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley, 2000

[Booch+99] Grady Booch, James Rumbaugh and Ivar Jacobson, *The unified modeling language user guide*, Addison-Wesley, 1999

[Bowen+02] Jonathan P. Bowen, Kirill Bogdanov, John Clark, Mark Harman, Robert Hierons, and Paul Krause, FORTEST: Formal Methods and Testing, *Proc. of the 26th IEEE Annual Intel. Computer Software and Applications Conf.*, Oxford, UK, 2002

[Brown02] David W. Brown, An Introduction to Object-Oriented Analysis: Objects and UML in Plain English, 2nd Edition, Wiley, 2002

[Campbell+95] M.C. Campbell, D.K. Hinds, A.V. Kapetanakis, D.S. Levin, S.J. McFarland, D.J. Miller, and J.S. Southworth, Object-Oriented Perspective on Software System Testing in a Distributed Environment, *Hewlett-Packard Journal*, 46(6), Dec. 1995

[Conallen99] Jim Conallen, Modeling Web Application Architectures with UML, *Rational Software White Paper*, June 1999

[Dai+02] Z.R. Dai, J. Grabowski, and H. Neukirchen, Timed TTCN-3 – A Real-Time Extension for TTCN-3, *Testing of Communicating Systems*, volume 14, Berlin, March 2002. Kluwer

[Dai+03] Zhen Ru Dai, Jens Grabowski, and Helmut Neukirchen, TIMEDTTCN-3 Based Graphical Real-Time Test Specification, *Proc. of Testing of Communicating Systems 2003, Sophia Antipolis (France)*, 2003. Lecture Notes in Computer Science (LNCS) 2644, Springer, 2003, pp. 110-127

[Danet04] Danet Company web site, *http://www.danet.de*, May, 2004

[DVC04] Da Vinci Communications Company web site, *http://www.davinci-communications.com/index.shtml*, May, 2004

[Deussen+03] Peter H. Deussen, George Din, and Ina Schieferdecker, A TTCN-3 Based Online Test and Validation Platform for Internet Services, *Proc. of the 6th Intel. Symposium on Autonomous Decentralized Systems*: pp. 177-184, 2003

[ECMA] ECMA262, *ECMAScripts Language Specification*, Standard ECMA-262 3rd edition, 1999

[Elbaum+03] Elbaum S., Karre S., Rothermel G., Improving Web Application Testing with User Session Data, *Proc. of the 25th Intel. Conf. on Software Engineering*: pp. 49-59, Portland USA, 2003

[ETSI201 873-1] ETSI ES 201 873-1, The Testing and Test Control Notation version 3, Part1: TTCN-3 Core Language, V2.2.1, 2003-02

[ETSI201 873-2] ETSI ES 201 873-2, The Testing and Test Control Notation version 3, Part2: TTCN-3 Tabular presentation Format (TFT), V2.2.1, 2003-02

[ETSI201 873-3] ETSI ES 201 873-3, The Testing and Test Control Notation version 3, Part3: TTCN-3 Graphical Presentation Format (GFT), V2.2.1, 2003-02

[ETSI201 873-5] ETSI ES 201 873-5, The Testing and Test Control Notation version 3, Part5: TTCN-3 Runtime Interface (TRI), V1.1.1, 2003-02

[ETSI201 873-6] ETSI ES 201 873-6, The Testing and Test Control Notation version 3, Part6: TTCN-3 Control Interface (TCI), V1.1.1, 2003-02

[EXAD] EX Group, Detailed Design and Development Documents for EX System (Release 1.0), Aug. 2002, University of Ottawa

[EXTP] EX Group, Online Exchange System Functional Test Plan (Version 1.2), Nov., 2002, University of Ottawa

[FME04] Formal Method Europe web site, *http://www.fmeurope.org/*, May, 2004

[Gornik02] Davor Gornik, UML Data Modeling Profile, *Rational Software White Paper*, May 2002

[Hsia+97] Pei Hsia, Xiaolin Li, David Chenho Kung, Chin-Tung Hsu, Liang Li, Yasufumi ToyoShima and Cris Chen, A Technique for Selective Revalidation of OO Software, *Software Maintenance: Research and Practice*, Vol. 9, 1997, pp. 217-233

[Humphrey97] Watts S. Humphrey, *Introduction to the Personal Software Process*, Addison-Wesley, 1997

[IBMRational] IBM Rational Software web site, http://www-306.ibm.com/software/rational/, Aug., 2004

[IEEE/ANSI90] IEEE/ANSI(1990), IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990

[Jacobson+99] Ivar Jacobson, Grady Booch, and James Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999

[Jia+02] Xiaoping Jia and Hongming Liu, Rigorous and Automatic Testing of Web Applications, *6th IASTED Intel. Conf. on Software Engineering and Applications*, Nov. 2002

[Kan02] Stephen H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd Edition, Addison-Wesley, 2002

[Kit95] Edward Kit, *Software Testing in the Real World*, Addison-Wesley, 1995

[Kung+94] D. Kung, J. Gao, P. Hsia, F. Wen, Change Impact Identification in Object Oriented Software Maintenance, *Proc. IEEE Int'l Conf. Software Maintenance*, 1994, pp. 202-211

[Kung+98] David C. Kung, Pei Hsia, and Jerry Gao, *Testing Object-Oriented Software*, IEEE Computer Society Press, 1998

[Kung+00A] Kung, D.C., Chien-Hung Liu, and Pei Hsia, An Object-Oriented Web Test Model for Testing Web Applications, *Proc. of the First Asia-Pacific Conf. on Quality Software*: pp. 111-120, 2000

[Kung+00B] Chien-Hung Liu, Kung, D.C., Pei Hsia, and Chih-Tung Hsu, Structural Testing of Web Applications, *Proc. of the 11th Intel. Symposium on Software Reliability Engineering:* pp. 84-96, 2000

[Lethbridge+01] Timothy C. Lethbridge and Robert Laganière, Object-Oriented Software Engineering: Practical Software Development Using UML and Java, McGraw Hill, 2001

[Li+00] Jingfeng Li, Jian Chen, and Ping Chen, Modeling Web Application Architecture with UML, *36th Int'l Conf. on Technology of Object-Oriented Languages and Systems (TOOLS-Asia'00)*: pp. 265-274, Nov. 2000

[Liskov+94] Barbara H. Liskov and Jeannette M. Wing, A behavioral notion of subtyping, *ACM Transactions on Programming Languages and Systems* 16(6): 1811-1841, Nov. 1994

[Lucca+02] Di Lucca, G.A., Fasolino, A.R., Faralli, F., and De Carlini, U., Testing Web Applications, *Proc. of Intel. Conf. on Software Maintenance*: pp 310-319, 2002

[McGregor+01] John D. McGregor and David A. Sykes, *A Practical Guide to Testing Object-Oriented Software*, Addison-Wesley, 2001

[Microsoft04] Microsoft Windows 2000 Server Documentation, *http://www.microsoft.com/windows2000/en/server/iis/default.asp?url=/windows2000/en/server/iis/htm/core/iigloss4.htm*, June, 2004

[Myers79] Glenford Myers, *The Art of Software Testing*, Wiley, 1979

[Nguyen+03] Hung Q. Nguyen, Bob Johnson and Michael Hackett, *Testing Applications on the Web: Test Planning for Mobile and Internet-Based Systems*, 2nd Edition, Wiley, 2003

[OpenTTCN04] OpenTTCN Company web site, *http://www.openttcn.com/*, May, 2004

[OULU04] University of OULU Web Site, *http://www.ee.oulu.fi/research/ouspg/sage/glossary*, May, 2004

[Patton01] Ron Patton, *Software Testing*, SAMS, 2001

[Probert+99] R.L. Probert, A.Williams, Fast Functional Test Generation Using an SDL Model, *Proc. of the 12th annual International Workshop on the Testing of Communicating Systems (IWTCS '99)*, Budapest Hungary, September 1999, pp. 299-315.

[Probert+00] R.L. Probert, D. P. Sims, B. Ghazizadeh, W. Li, A Risk-Directed E-Commerce Test Strategy, *Proc. of Quality Week Europe 2000 Conf. (QWE)*: pp. 388-401, 20-24 Nov. 2000

[Probert+03] Robert L. Probert, Yanping Chen, Behrad Ghazizadeh, D. Paul Sims, Maurus Cappa, Formal Verification and Validation for E-Commerce: Theory and Best Practices, *Information and Software Technology 45 (2003) 763-777*, 2003

[Probert03] Robert L. Probert, *Software Quality Engineering Lecture Notes*, University of Ottawa, 2003

[Probert+04] Robert L. Probert, Pulei Xiong, Bernard Stepien, Life-cycle E-Commerce Testing with OO-TTCN-3, *1st Intel. Workshop on Theory Building and Formal Methods in Electronic/Mobile Commerce (TheFormEMC)*, Oct. 2004

[Ricca+01] Ricca, F. and Tonella, P., Analysis and Testing of Web Applications, *Proc. of the 23rd Intel. Conf. on Software Engineering:* pp. 25-34, 2001

[Rothermel+94] G. Rothermel and M.J. Harrold, Selecting Regression Tests for Object-Oriented Software, *Proc. IEEE Int'l Conf. Software Maintenance*, 1994, pp. 14-25

[Rumbaugh+99] James Rumbaugh, Ivar Jacobson and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999

[Satzinger+01] John W. Satzinger and Tore U. Orvik, The object-oriented approach: concepts, system development, and modeling with UML, Course Technology, 2001

[Schach02] Stephen R. Schach, *Object-Oriented and Classical Software Engineering*, 5th Edition, McGraw Hill, 2002

[Schiefer+01] I. Schieferdecker, S. Pietsch, and T. Vassiliou-Gioles, Systematic Testing of Internet Protocols - First Experiences in Using TTCN-3 for SIP, *5th IFIP Africom Conference on Communication Systems*, Cape Town (South Africa), May 2001

[Schiefer+03A] I. Schieferdecker and B. Stepien, Automated Testing of XML/SOAP based Web Services, Proc. of the 13th. Fachkonferenz der Gesellschaft f˙ur Informatik (GI) Fachgruppe "Kommunikation in verteilten Systemen" (KiVS), Leipzig (Germany), 2003

[Schiefer+03B] Ina Schieferdecker and Theofanis Vassiliou-Gioles, Tool Supported Test Frameworks in TTCN-3, *the 8th intel. Workshop on Formal Methods for Industrial Critical Systems (FMICS03)*, Trondheim (Norway), 2003. Electronic Notes in Theoretical Computer Science, volume 80

[Schmitt+03] Michael Schmitt and Michael Ebner, The TTCN-3 module and template concepts revisited, *Technical Report IFI-TB-2003-02*, ISSN 1611-1044

[Singh+02] Inderjeet Singh, Beth Stearns, Mark Johnson, and the Enterprise Team; *Designing Enterprise Applications with the J2EETM Platform*, 2nd Edition, Addsion & Wesley, 2002

[Telelogic04] Telelogic Company web site, *http://www.telelogic.com/*, May, 2004

[TestingTech04] Testing Technologies Company web site, *http://www.testingtech.de/*, May, 2004

[TTCN304] TTCN-3 Home Page, *http://www.etsi.org/ptcc/ptccttcn3.htm*, Mar. 2004

[Turner93] C.D. Turner and D.J. Robson, The State-Based Testing of Object-Oriented Programs, *Proc. IEEE Conf. Software Maintenance*, IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 302-310

[Webopedia04] Webopedia web site, http://www.webopedia.com/TERM/E/electronic_commerce.html, June, 2004

References

[Wen01] Wen, R.B., URL-Driven Automated Testing, *Proc. of the 2nd Asia-Pacific Conf. on Quality Software:* pp.268-272, 2001, Hong Kong

[Wiles00] Anthony Wiles, *An Introductory Overview of TTCN-3*, ETSI PTCC/ETSI STF 156, 2000

[Xiong+03] Pulei Xiong, Robert L. Probert, A Multi-Method Testing Approach in TTCN-3 for Web Applications, *Proc. of the 14th IEEE Intel. Symposium on Software Reliability Engineering (ISSRE 2003),* Fast Abstract, November, 2003.

[Yang+99] Ji-Tzay Yang, Jiun-Long Huang, Feng-Jian Wang, and Chu, W.C., An Object-Oriented Architecture Supporting Web Application Testing, *Proc. of the 23rd Annual Intel. Computer Software and Applications Conf.*: pp. 122-127, 1999