1. Introduction

Over the last several decades, as software systems become more and more large-scale, complex and uneasily controlled, software community has faced the challenge of high development cost, low productivity, uncontrollable software quality and risk to move to new technology. This has created a fast growing demand for rapid and cost-effective development of large-scale, complex and highly maintainable software systems [Pour99c]. It also causes searching for a new, efficient, and cost-effective software development paradigm.

The most promising solution now is component-based software development approach. This approach is based on the idea that developing software systems by selecting building blocks of a new system from off-the-shelf components and assembling the selected components with an appropriate software architecture rather than implementing the system from scratch [Pour98]. These components can be existing subsystems by internal or external sources, or commercial off-the-shelf (COTS) components developed by different in-house developers using different languages and different platforms.

Component-based software development (CBSD) has potential to reduce significantly development cost and time-to-market, and improve maintainability, reliability and overall quality of applications [Pour99a] [Pour99b]. So it has raised a tremendous amount of interests both in the research community and in the software industry.

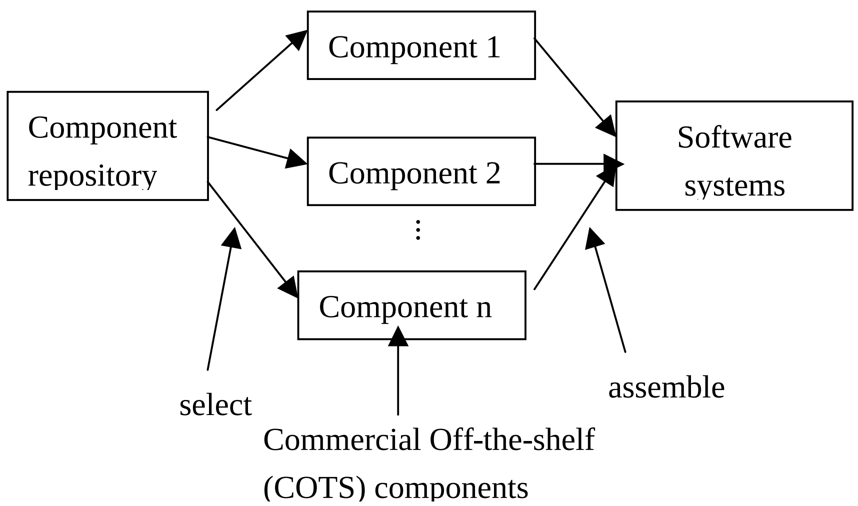


Figure 1.1 Component-Based Software Development

The concept of using software components is not so new: as operating systems, compilers, database systems, networking systems, and software tools are all indeed software components, and they have well-defined functions and interfaces that can easily tested and integrated with other software systems [Pour98]. What is new about component-based software development approach is its use of commercial off-the-shelf (COTS) software components as the building blocks of new systems. And this involves new major activities such as evaluation, selection, customization, and integration of off-the-shelf components; and evaluation, selection, and creation of software architecture. As a result, the life cycle and software engineering model of Component-Based Software Development is much different with the traditional ones, that's what the Component-Based Software Engineering (CBSE) is focused.

As yet, the software component technologies is far from matured, there is no existing standards or guidelines in this new area, and we don't even have a unified definition of the key item “component” in component-based software development [Brow98]. In general, a component has three main features: 1) a component is a independent and replaceable part of a system that fulfills a clear function; 2) a component works in the context of a well-defined architecture; 3) it communicates with other components by the interfaces.

To ensure a component-based software system to run properly and effectively, the system architecture is the most important. From both research community [Gris97] and industry practice [IBM00], the system architecture of component-based software systems should be a layered and modular architecture. A top application layer consists of related application systems supporting a business. Below the application layer are components reusable only for the specific business or application domain area, includes components usable in more than a single application. A third layer of cross-business middleware components includes common software and interfaces to other established entities. The lowest layer of system software components includes interfaces to hardware.

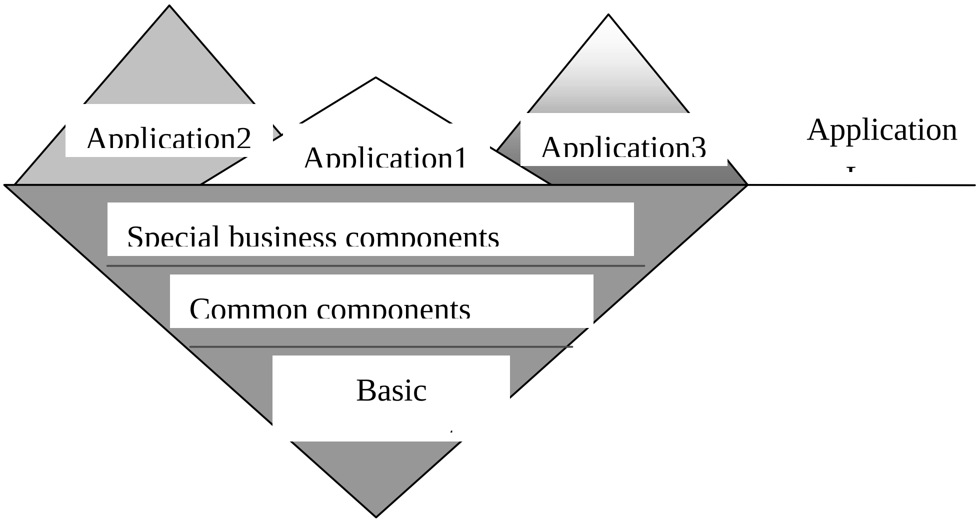


Figure 1.2 System Architecture of Component-Based Software Systems

2. Current Component Technologies

Some languages, such as Visual Basic, C++ and Java, and the supporting tools, make it possible to share and distribute application pieces through approaches such as Visual Basic Controls (VBX), ActiveX controls, class libraries, and JavaBeans. But each of these approaches relies on some underlying services to provide the communication and coordination necessary to piece together applications. The infrastructure of components (sometimes called a component model) acts as the "plumbing" that allows communication among components [Brow98]. Among the component infrastructure technologies that have been developed, three have become somewhat standardized: the OMG's CORBA, Sun's JavaBeans and Enterprise JavaBeans, and Microsoft's Component Object Model (COM) and Distributed COM (DCOM) [Koza98].

2.1 Common Object Request Broker Architecture (CORBA)

CORBA is an open standard for application interoperability that is defined and supported by the Object Management Group (OMG), an organization of over 400 software vendor and object technology user companies [OMG00]. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by OMG and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0 adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

The ORB is the middleware that establishes the client -server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In so doing, the ORB pr ovides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

In fielding typical client/server applications, developers use their own design or a recognized standard to define the protocol to be used between the devices. Protocol definition depends on the implementation language, network transport and a dozen other factors. ORBs simplify this process. With an ORB, the protocol is defined through the application interfaces via a single implementation language -independent specification, the IDL. And ORBs provide flexibility. They let programmers choose the most appropriate operating system, execution environment and even programming language to use for each component of a system under construction. More importantly, they allow the integration of existing components. In an ORB -based solution, developers simply model the legacy component using the same IDL they use for creating new objects, then write "wrap per" code that translates between the standardized bus and the legacy interfaces.

CORBA is widely used in Object -Oriented distributed systems including component-based software systems because of the features that it offers a consistent distributed prog ramming and run -time environment over most commonly used programming languages, operating systems and networks. Its Interface Definition Language (IDL) is suitable for specifying the component interfaces without implementation details [Yau98].

2.2 Component Object Model (COM) and Distributed COM (DCOM)

Introduced in 1993, Component Object Model (COM) provides platform-dependent - based on Windows® and Windows NT, and language -independent component -based applications [Micr00].

COM defines how compo nents and their clients interact. This interaction is defined such that the client and the component can connect without the need of any intermediary system component. Specially, COM provides a binary standard that components and their clients must follow to ensure dynamic interoperability. This enables on -line software update and cross-language software reuse [Wang97].

As an extension of the Component Object Model (COM), Distributed COM (DCOM), introduced in 1996, is a protocol that enables software com ponents to communicate directly over a network in a reliable, secure, and efficient manner. Previously called "Network OLE," DCOM is designed for use across multiple network transports, including Internet protocols such as HTTP.

When client and compone nt reside on different machines, DCOM simply replaces the local interprocess communication with a network protocol. Neither the client nor the component is aware that the wire that connects them has just become a little longer.

2.3 Sun Microsystems's JavaBeans and Enterprise JavaBeans

Sun's Java-based component model consists of two parts: the JavaBeans for client-side component development and the Enterprise JavaBeans (EJB) for the server-side component development. The JavaBeans component architecture is designed to enable enterprises to build scalable, secure, multiplatform, business -critical applications as reusable, client-side and server-side components [SUN00].

Java platform offers an elegant and efficient solution to the portability and se curity problems through the use of portable Java bytecodes and the concept of trusted and untrusted Java applets. Java provides a universal integration and enabling technology for enterprise application development. This includes:

1) interoperating across multivendor servers;
2) propagating transaction and security contexts;
3) servicing multilingual clients; and
4) supporting ActiveX via DCOM/CORBA bridges.

JavaBeans and EJB extend all native strengths of Java including portability and security into the area of component-based development. The portability, security, and reliability of Java are well suited for developing server objects that are robust, and independent of operating system, Web servers and database management servers.

2.4 Comparison between existing component technologies

2.4.1 EJB versus DCOM and CORBA

EJB has several advantages for enterprise application development, as it provides [Pour99a]:

1) efficient data access across heterogeneous server;
2) faster Java client connections, transaction state management, caching and queuing;
3) connection multiplexing, and
4) transaction load balancing across servers.

Developing Web-based applications with EJB is significantly easier than with CORBA and DCOM for the following reasons:

1) EJB is portable across Java virtual machines (VMs) and EJB servers. And the EJB transaction server concept allows scalability, reliability, and atomic transactions of enterprise applications on various platforms.
2) Application development with EJB does not involve low-level system programming such as thread-aware programming. Scalability requirements are automatically addressed by the EJB server implementation.
3) Application development with EJB does not involve creating and using Interface Definition Language (IDL) files, as EJB defines the interfaces between a server-side component and its container. As a result, modification and maintenance of applications using JavaBeans and EJB are easier than those using CORBA or COM/DCOM.
4) Application development with EJB does not deal with transactional and security semantics in the bean implementation and security rules for an EJB can be defined

at the time of assembly and deployment. Furthermore, the transaction semantics are defined declaratively through a bean's deployment descriptor rather than programmatically. The EJB server automatically manages the start, commit, and rollback of transactions on behalf on the EJB according to a transaction attribute specified in the EJB deployment descriptor.

2.4.2 DCOM versus CORBA

Comparison of d ifference between Microsoft 's and OMG 's technologies is listed in table 2.1 [Brow98].

3. Case study

People have used current component technologies to their component software development, such as object -oriented distributed component software developm ent [Yau98] and Web-based enterprise application development [Pour99a]. And there are some commercial players involve in the software component revolution, such as BEA, Microsoft, IBM and Sun [Koza98].

In order to solve the high cost and low efficiency problems when software developers want to modernize their current applications or maintain the complex specific software system, IBM SanFrancisco project provides application developers with a distributed object infrastructure and a set of application components which can be expanded and enhanced by application developer to provide competitive differentiation [IBM00]. The business process components, written in the Java language, are intended to lower the barriers to widespread commercial implementation of distributed object solutions.

	OMG	Microsoft
Component interface	CORBA IDL for defining component interfaces	Microsoft IDL for defining component interfaces
Underlying mode	The basic CORBA client-component model	The basic COM client/component model
Connection protocol	IIOP, the interoperability standard that allows different CORBA vendors to work together	DCOM for distributing components across a network

Lifecycle	Life Cycle Service, to define how component instances are instantiated	Microsoft Transaction Service(MTS) to provide a secure runtime environment, transaction management, and scalability
Service provided	1.Naming Service, to define how component instances are shared 2.Security Service, to define how clients and component instances work together securely 3.Transaction Service, to define how distributed transactions are controlled	1.DTC for distributed transaction coordination 2.Microsoft Message Queue for asynchronous messaging.
Platform dependency	Platform independent	Platform dependent
Implementation	Creates no reference implementations and depends on vendors for actual delivery	Creates implementations

Table 2.1 Comparison of technologies from Microsoft and OMG

SanFrancisco components are pre-tested to enable developers to build and modify business applications quickly. Cross-platform applications can be built once and run on a wide range of servers, including Windows NT, OS/400, AIX, Solaris, HP-UX and Reliant UNIX.

SanFrancisco includes an application Foundation layer, plus hundreds of Common Business Objects (such as company, address, currency, business partner, unit of measure, cash balances, etc.). In addition, application-specific support is provided for common business processes such as general ledger, order processing, inventory management, product distribution and accounts payable/receivable.

SanFrancisco is building three layers of reusable code for use by application developers.

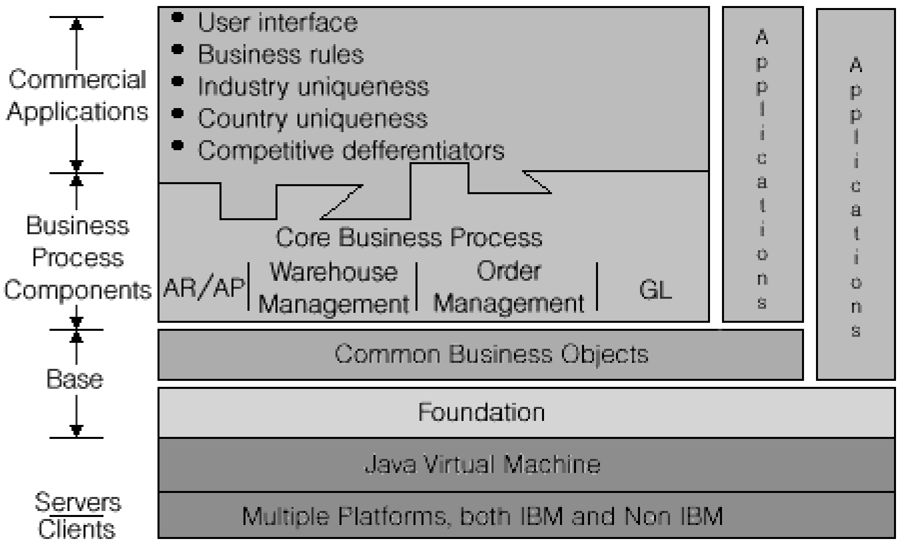


Figure 2.1 System Infrastructure of IBM SanFrancisco

- The lowest layer, called the Foundation, provides the infrastructure and services that are required to build industrial -strength applications in a distributed, managed-object, multi-platform applications.
- The second layer, called the Common Business Objects, provides definitions of commonly used business objects that can be used as the foundation for interoperability between applications.
- The highest layer, called the Core Business Processes, provides business objects and default business logic for selected vertical domains. Initially, IBM SanFrancisco is delivering business components in the domains of accounts receivable, accounts payable, general ledger, order management (sales and purchase), and warehouse management. Over time, these components will be extended and enhanced with additional business processes, objects, and access to more framework interfaces, providing greater application flexibility.

Together, the Common Business Objects, the Foundation, and associated utilities form the Base. The Base layers isolate an application from the complexities of multi-platform network technology and free the application provider to focus on unique elements that drive value to their customers.

So far, SanFrancisco is the largest server-side Java initiative in the industry and is a key element in IBM's Application Framework for e-business.

4. Software Quality Assurance

4.1 Traditional QA

Traditionally quality is defined as conformance to specification or requirements, and failures arise when the software is not met the requirements. The International Standard Quality Vocabulary (ISO 8402) defines quality as: “The totality of features and characteristics of a product or service that bear on its ability to meet stated or implied needs.” According to ISO9126, the definition of quality characteristics includes: functionality, reliability, usability, efficiency, maintainability and portability.

According to Sanders and Curran [Sand94], Software Quality Assurance is a planned and systematic pattern of actions to provide adequate confidence that the item or product conforms to established technical requirements. In a more specific project context, it is about ensuring that project standards and procedures are adequate to provide the required degree of quality, and that they are adhered to throughout the project..

Quality Assurance focused on both the product and the process. The product-oriented part of SQA (often called Software Quality Control) should strive to ensure that the software delivered has a minimum number of faults and satisfies the users' needs. The process-oriented part (often called Software Quality Engineering) should institute and implement procedures, techniques and tools that promote the fault-free and efficient development of software products.

Quality assurance activities include:

1) Management
   Analysis of the managerial structure that influences and controls the quality of the software is an SQA activity. It is essential for an appropriate structure to be in place and for individuals within the structure to have clearly defined tasks and responsibilities.
2) Documentation
   It is essential to analyze the documentation plan for the project, to identify deviations from standards relating to such plans, and to discuss these with project management.

3) Standards and Practices
   It is essential to monitor adherence to all standards and practices throughout the project.

- Documentation standards.

- Design standards.
- Coding standards.
- Code commenting standards.
- Testing standards and practices.
- Software quality assurance metrics.
- Compliance monitoring.

4) Reviews and Audits

   It is essential to examine project review and audit arrangements, to ensure that they are adequate and to verify that they are appropriate for the type of project.

5) Testing

   Unit, integration, system and acceptance testing of executable software are an integral part of the development of quality software.

6) Problem Reporting and Corrective Action

   It is essential to review and monitor project error-handling procedures to ensure that problems are reported and tracked from identification right through to resolution, and that problem caused are eliminated where possible. It is also important to monitor the execution of these procedures and examine trends in problem occurrence.

7) Tools, Techniques and Methods

   Tools, techniques and methods for software production should be defined at the project level.

8) Code and Media Control

   It is essential to check that the procedures, methods and facilities used to maintain, store, secure and document controlled versions of software are adequate and are used properly.

Software Quality Assurance aims at cost-effective, flexibility, rich functionality, certain reliability and safety of software systems. To achieve software quality, the life cycle of software design is promoted, it mainly includes [Smit95]:

- requirements specification;
- system and module design;
- coding and implementation;
- test.

Also, there are formal methods in software requirements specification, formal methods permit each stage of design to be checked against the previous stage(s) from consistency and correctness. Three main types of Formal Method are: 1) data-oriented Formal Method, including model-based notation (VDM, Z) and algebraic notation (OBJ); 2) process-oriented Formal Method, including communications sequential processes (CSP) and calculus of concurrent systems (CCS); 3) state-oriented formal methods, such as Petri-net.

Moreover, different metrics can be applied to project control, predicting coding and test times, productivity and machine usage; and quality assurance related to reliability and safety. There are two main types of metrics: process-related metrics and product-related metrics [Jaco92]. Process-related metrics measure things like cost, effort, schedule time and number of faults found during testing. While product-related metrics predict coding and test times, productivity and machine usage. Some traditional metrics are as follows: 1) lines of code; 2) percentage comment; 3) module complexity; 4) subjective complexity; 5) control path cross; 6) design complexity; 7) design to code expansion rate; 8) fan-in, fan-out; 9) fault detection rate; 10) number of changes by type; 11) staff quality and etc. [smit95].

Testing is the last procedure to detect the existing faults in software. There are some test tools, such as test drivers, test beds, emulators, and some packages like ADATEST, Cantana, FX, Mans, Orion ICE designed by different companies to test software developed by different languages.

Standards and guidelines are used to control the quality activities. The two most famous and widely-used software quality standards are ISO 9000-3 and CMM model. ISO 9000 is an international series of standards, developed by the International Organization for Standardization, that specifies a basic set of requirements for a quality system to provide consistent, acceptable quality products [Schm94]. Its emphasis is on the development process and the management responsibilities associated with the process. It focuses on establishing, documenting, and following a well-controlled, reviewed, and improved. ISO9000-3 provides guidance on how to apply ISO 9000 standards to software development. The guidance is excellent and has adopted widely by software community when designing quality software systems.

The Capability Maturity Model (CMM), developed by the Software engineering Institute (SEI), is a framework that describes the elements of an effective software process and an evolutionary path that increases an organization's software process maturity [Sand94]. A fundamental principle underlying the CMM is that the quality of a software product can be improved by improving the process which produces it. The CMM characterizes five levels of increasing process maturity, they are the Initial, Repeatable, Defined, Managed and Optimizing maturity levels, by the extent to which the organization's processes comply with specified key practices. The CMM is something like a type of metric, in that it involves scoring criteria which enable a project or organization to assess its maturity level in terms of software engineering practice.

Besides ISO9003 and CMM, there are many localized and customized guidelines or models of software quality assurance in different countries or areas. Particularly in Hong Kong, Hong Kong Productivity Council has developed *Hong Kong Software Quality Assurance Model*, a framework of standard practices that a software organization in Hong Kong should have to produce quality software [HKPC00]. The HK Software Quality Assurance Model provides the standard for local software organizations (independent or internal; large or small) to:

- Meet basic software quality requirements;
- Improve on software quality practices;
- Use as a bridge to achieve other international standards;
- Assess and certify them to a specific level of software quality conformance.

The seven practices that form the basis of the HK Software Quality Assurance Model are: 1) Software Project Management; 2) Software Testing; 3) Software Outsourcing; 4) Software Quality Assurance; 5) User Requirements Management; 6) Post Implementation Support; and 7) Change Control.

4.2 QA for Object-Oriented software systems

4.2.1 Differences between Object-Oriented software and traditional systems

Object-Oriented technology is a technique for system modeling [Jaco92]. Different from traditional procedure-based approach, OO views the system as a number of objects that interact. Interest in the object-oriented method has grown rapidly over the last few years. This is mainly due to the fact that it has shown many good qualities. Amongst the most prominent qualities of a system designed with an object-oriented method are the followings:

- **Understanding** of the system is easier as the semantic gap between the system and reality is small;

- **Modifications** to the model tend to be local as they often result from an individual item, which is represented by a single object.

It is widely accepted that the OO paradigm significantly increase software reusability, extendibility, interoperability, and reliability.The key concepts in OO paradigm are:

Object

A object encapsulates the data and operations on the data. A object communicates with other objects by sending messages between them.

Class

A class is sometimes called the object's type, a class represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their operations and for their information structure.

Polymorphism

Polymorphism means that the sender of a stimulus does not need to know the receiving instance's class. The receiving instance can belong to an arbitrary class.

Inheritance

If class B inherits class A, then both the operations and the information structure described in class A will become part of class B.

Object-Oriented design is the process of identifying objects and their attributes, identifying operations suffered by and required of each object, and establishing interfaces between object [Booc86]. The design of objects involves three steps:

1) definition of objects;
2) attributes of objects;
3) communication between objects.

The fundamental concepts of OO design are shown in Figure 4.1.

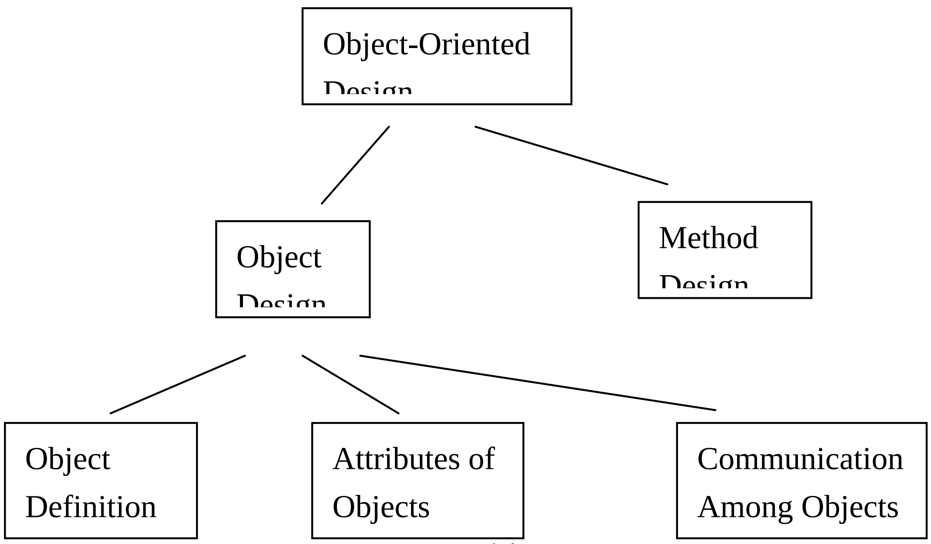


Figure 4.1 Elements of Object Oriented Design

4.2.2 A Quality-focussed Object-Oriented Process --- OPEN

OPEN is a third-generation, full lifecycle process framework that is ideally suited for both object -oriented and component -based development. OPEN standards for Object-Oriented Process, Environment and Notation. It is fully described in a series of books including [Grah97] and [Hend98].

Firstly, it is important to note that OPEN is a full lifecycle methodological approach. Software can be considered to have a lifecycle from birth to death. The need for software can arise when business problems need solution. Business decision making, requirements engineering and systems analysis are all "early lifecycle" activities. Similarly, a methodology should cover the late lifecycle activities. Whilst most are good at program design and coding, they tend to tail off in their coverage of issues such as deployment and user training and future enhancements/maintenance. Here, product metrics are better developed.

Secondly, OPEN is a process -focussed methodological approach for software and component development. From a methodological viewpoint, process is the key to good software development practices. And OPEN is a framework defined by a process metamodel. It means that OPEN is not rigidly specified, and the framework constraints metalevel connections between Stages, Activities, Tasks, Deliverables, Producers and so on. The actual Activities, Tasks etc. to be used are chosen by the user. In this way, the user can tailor the OPEN process framework to fit exactly the requirements of their project and organization.

OPEN is also well-suited for component-based development. It provides support for components designing on the web.

4.2.3 Metrics for Object-Oriented Software Systems

The characteristics of software quality are not exhaustive and not even independent of each other. Additionally, they often tend to conflict in a development. Therefore, when starting a development, it is often a good idea to decide which characteristics are the most important for this specific product and then focus on these throughout the development. In Object-Oriented the focus is on mainatainability characteristics, as well as suitability [Jaco92].

A necessary method for controlling a development is to use metrics. The metrics c an measure either the process of development or various aspects of the product. Because of the characteristics of OO development, the "internal" attributes in the products includes: modularity, low coupling, high cohesion, encapsulation and other, while th e "external" attributes expected by software users, are reliability, maintainability, and reusability [Raja92a].

Some Process-related metrics for OO development are as followings [Jaco92]:

- Total development time;
- development time in each process and subprocess;
- time spent modifying models from previous processes;
- time spent in all kinds of subprocesses, usch as use case specification, object specification, use case design, block design, block testing and use case testing for each particular object;
- number of different kinds of fault found during reviews;
- number of change proposals for previous models;
- cost for quality assurance;
- cost for introducing new development process and tools;

Traditional metrics on products (including code) may to some extent also be used in object-oriented software. However the most common metric, lines of code, is actually less interesting for object-oriented software. Here are some examples of metrics that are more appropriate for object-oriented software [Jaco92]:

- Total number of classes;
- Number of classes reused and the number newly developed;
- Total number of operations;
- Number of operations reused and the number newly developed;
- Total number of stimuli sent;
- Number, width and height of the inheritance hierarchies;
- Number of classes inheriting(or using) a specific operation;
- Number of classes that a specific class is dependent on;
- Number of classes that are dependent on a specific class;

- Number of direct users of a class or operation (the highest scored are candidates for components).
- Average number of operations in a class;
- Length of operations (in statements);
- Stimuli sent from each operation;
- Average number of descendants for a class;
- Average number of inherited operations

Besides all these general process-related and product-related metrics, many metrics have been proposed to measure OO software complexity and assure software quality. [Shih97] introduces a factor and a method to realize and measure the object-oriented software complexity of a class hierarchy. Because inheritance and polymorphism are key concepts in object-oriented programming, and are essential for achieving reusability and extendibility, in [Raja92a] [Raja92b], the authors define four measures of coupling:

1) Class inheritance-related Coupling(CIC)
2) Class Non-Inheritance-related Coupling (CNIC)
3) Class Coupling (CC)
4) Average Method Coupling (AMC)

Metamata is a company who provides some metrics and audits, Table 4.1 are some example metrics for Java, one of OO languages [meta00].

Based on two approaches to software quality assurance used ---fault prevention and fault detection, the authors of [Lo98] examines the factors that affect software testability in object-oriented software and proposes a preliminary framework for the evaluation of software testability metrics. They are listed in Table 4.2.

4.2.5 Testing for Object-Oriented Software Systems

Software testing is an important software quality assurance activity to ensure that the benefits of OO programming will be realized. OO software testing has to deal with new problems introduced by the powerful new features of OO languages, such as encapsulation, inheritance, polymorphism, and dynamic binding [Kung98]. The dependencies occurring in conventional systems are:

1) Data dependencies between variables;
2) Calling dependencies between modules;

3) Functional dependencies between a module and the variables it computes;
4) Definitional dependencies between a variable and its type.

OO systems have additional dependencies:

1) Class to class dependencies;
2) Class to method dependencies;
3) Class to message dependencies;
4) Class to variable dependencies;
5) Method to variable dependencies;
6) Method to message dependencies; and
7) Method to method dependencies.

OO testing problems can be summarized to be: 1) the understanding problem; 2) the complex interdependency problem; 3) the object state behavior testing problem; and 4) the tool support problem. The understanding problem is introduced by the encapsulation and information hiding features. The dependency problem was caused by the complex relationships that exist in an OO program. Objects have states and state dependent behaviors. That is, the effect of an operation on an object depends also on the state of the object and may change the state of the object. Thus, the combined effect of the operations must be tested.

Test strategy

A test strategy can be defined as the order to unit testing and integration testing of the classes in an OO program. The test order problem for the classes in an OO program can be stated as finding an order to test the classes so that the effort required is minimum. The example methodology consists of the following steps:

Step 1. Initially, base classes having no parents are chosen and a test suite is designed that tests each member function individually and also the interactions among member functions.

Step 2. A testing history associates each test case with the attributes it tests. In addition to inheriting attributes from its parents, a newly defined subclass "inherits" its parent's testing history.

Step 3. The inherited testing history is incrementally updated to reflect differences from the parent and the result is a testing history for the subclass.

Step 4. With this technique, new attributes can be easily identified in the subclass that must be tested along with inherited attributes that must be retested.

Metric	Measures	Description
Cyclomatic Complexity	Complexity	The amount of decision logic in the code
Lines of Code	Understandablility, maintainability	The length of the code; related metrics measure lines of comments, effective lines of code, etc.
Weighted Methods per Class	Complexity, understand-ability, reusability	The number of methods in a class
Response for a Class	Design, usability, testability	The number of methods that can be invoked from a class through messages
Depth of Inheritance Tree	Reusability, testability	The depth of a class within the inheritance hierarchy
Coupling Between Objects	Design, reusability, maintainability	The number of other classes to which a class is coupled
Number of Attributes	Complexity, maintainability	The amount of state a class maintains as represented by the number of fields declared in the class

Table 4.1 Metamata Metrics for Java

Types of Factors	Testability Factors		
Intra-method	Execution Rate	Propagation Rate	
Inter-method	Cohesion		
Intra-class	No of methods	Depth in inheritance tree	No of children
Inter-class	No of coupling		
Program	No of disjoint inheritance trees		

Table 4.2 testability factors according to different types

Step 5. The inherited attributes are retested in the context of the subclass by identifying and testing their interactions with newly defined attributes in the subclass.

Step 6. The test cases in the parent class's test suite that can be reused to validate the subclass and attributes of the subclass which require new test cases can also be identified in the process.

Unit Test and Integration Testing

In OO development, new test generation methods, test models, test coverage criteria for classes are needed in unit tests. Several methods are proposed using flow graph-based or data bindings of class.

When software components (or parts) are separately tested, they are integrated together to check if they can work together properly to accomplish the specified functions. The major testing focus here is their interfaces, integrated functions, and integrated behaviors. A number of software integration testing approaches have been used to perform software integration testing, such as top-down, bottom-up, sandwich, and "big bang". There are many differences between object-oriented programs and traditional programs.

The first is the structural differences between an object-oriented program and a traditional program. A conventional program consists of three levels of components: 1) functions (or procedures); 2) modules; and 3) subsystems. However, an object-oriented program consists of four levels of components: 1) function members defined in a class; 2) classes; 3)groups of classes; and 4) subsystems.

The other major difference between an object-oriented program and a conventional program is their behaviors. In a dynamic view, a conventional program is made a number of active processes. Each of them has its control flow. They interact with one and another through data communications. An object-oriented program consists of a collection of active objects that communicate with one and another to complete the specified functions. In a multiple-thread program, there are a number of object message flows executing at the same time.

A method for integration testing suggests five distinct levels of object-oriented testing, including a method, message quiescence, event quiescence, thread testing, and thread interaction testing. The basic idea is to model the behaviors of an object-oriented program using an object network.

Object State Testing

Object State Testing is an important aspect of object oriented software testing. It is different from the conventional control flow testing and data flow testing methods. In control flow testing, the focus is testing the program according to the control structures(i.e., sequencing, branching, and iteration). In data flow testing, the focus is testing the correctness of individual data define-and-use. Object state testing focuses on testing the state dependent behaviors of objects.

Regression testing

The main concern in regression testing is how to effectively and efficiently identify the changes and their impact so that testing can be focused to the changed and affected components. Another consideration in regression testing is reuse of existing test cases and test suites.

Testing tools

Different tools are developed to assist testers in testing and regression testing. Roong-Ko Doong and Phyllis G. Frankl [Kung98] reported their systematic approach to unit testing of object -oriented programs and a set of test tools, called ASTOOT. The major focus of this approach is how to automate the unit testing of abstract data types (ADTs) in object -oriented programs in test data generation, test ex ecution, and test checking.

5. Quality Assurance for component-based software systems

5.1 The life cycle of component-based software systems

Component-based software systems are developed by selecting various components and assembling them together rather than programming from scratch, thus the life cycle of component-based software systems is much different from that of the traditional software systems. The life cycle of component -based software systems is as follows [Pour98]:

- Requirements analysis;
- Software architecture selection, creation, analysis, and evaluation;
- Component evaluation, selection, and customization
- Integration
- Component-based software system testing
- Software maintenance

In the life cycle above, the two major activities are: 1) software architecture selection, creation, analysis, and evaluation; 2) component evaluation, selection, and customization. The architecture of software defines that system in terms of computational components and interactions among components. The focus is on composing or assembling components that are likely to have been developed separately, even independently. Component evaluation, selection and customization is a crucial activity in the life cycle of component systems, it includes two main parts: 1) evaluation of each candidate off-the-shelf component based on the functional and quality requirements that will be used to assess that component; 2) customization of those candidate off-the-shelf components that should be modified before being integrated into the new component-based software systems. And Integration is to make key decisions on how to provide communication and coordination among various components of a software system.

Quality Assurance for component-based software systems should addresses the life cycle and key activities to analysis the components and achieve high quality component-based software systems. Quality Assurance technologies for component-based software systems is currently premature because of the specific characteristics of component systems from traditional systems.

Although some QA technique such as reliability analysis model for distributed software systems [Yaco99a] [Yaco99b], and component-based approach to Software Engineering [Ning94] have been reached, there is still no clear and well-defined standards or guidelines for component-based software systems. The identification of the QA characteristics, along with the models, tools and metrics, are all under research.

5.2 Differences between components and objects

Software components represent a new concept in how to build software applications, but the foundations on which they are based have been around for quite some time as objects. That is, component-base technology is based on OO technology, but there still are some differences between component and objects.

Objects are generally (though not always) defined at too low a level to be easily related to a business process, and components are a higher-level, coarser-grained software entity. A crucial difference between objects and components revolves around inheritance. Objects support inheritance from parent objects, when an inherited attribute is changed in the parent object, the change ripples through all the child objects that contain the inherited attribut e. While inheritance is a powerful feature, it can also cause serious complications that result from the inherent dependencies it creates. In contrast to the multiple inheritance model of objects, components are characterized by multiple interfaces. Thus, components effectively eliminate the problem of dependencies related to object inheritance, instead, component interfaces act as the "contract" between the component and the application, the application has no view inside the component beyond the exposed i nterface. This provides users with the flexibility to update components while maintaining only the interface and behavior of the components [Herz00].

A component has a life cycle as illustrated in Figure 5 .1. Some metrics used to identifying components include [Jaco92]:

- **Size.** This affects both reuse cost and quality. If it is too small, thebenefits will not exceed the cost of managing it. If it is too large, it is hard to have high quality.
- **Complexity.** This also affects reuse cost and quality. A too trivial component is not profitable to reuse and with a too complex component itis hard to have high quality.
- **Reuse frequency.** The number of places where a component is used is of course important too.

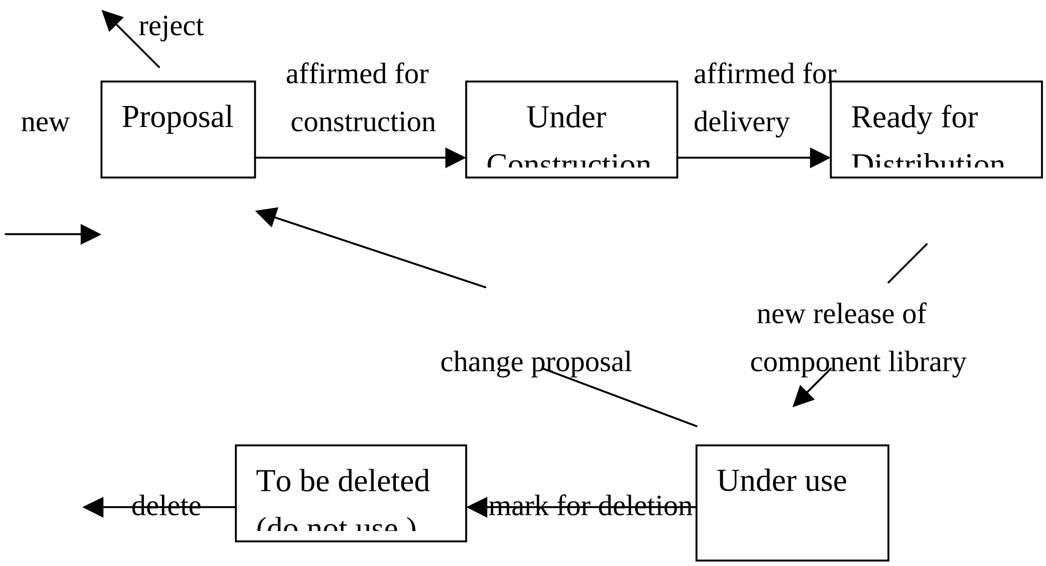


Figure 5.1 The life cycle of a component

5.3 Open problems about QA for component-based software

Although some QA technique such as reliability analysis model for distributed software systems [Yaco99a] [Yaco99b], and component -based approach to Software Engineering [Ning94] have been reached, there is still no clear and well -defined standards or guidelines for component-based software systems.

As many work has to be done to component -based software development, quality assurance technologies for component -based software development has to address the two inseparable parts: 1) How to certify quality of a component? 2) How to certify quality of software systems based on components [Pour98]? To answer the questions, models should be promoted to define the overall quality control of components and systems; metrics should be found to measure the size, complexity and reliability of components and systems; tools should be decided to test the existing components and systems.

5.4 Quality Characteristics of Components

To evaluate a component, we must determine how to certify the quality of components. The quality characteristics of components are the foundation to guarantee the quality of components, and thus the foundation to guarantee the quality of whole component-based software systems. Here are some recommended characteristics of quality of components.

■ **Functionality**

-- The degree to which the component implements all required capabilities.
-- Contains all references and required items.
-- The degree to which a component is free from faults in its specification, design, and implementation;
-- The degree to which a component is free from faults in its specification, design, and implementation;

■ **Interface**

-- The completeness of the input/output of a component

-- The flexibility of the interface to add/decrease some parameters

- **Userability**

-- The number of users of a component.
-- The sum of the lengths of time when used.

- **Testability**

-- Equipped with test cases, test plans and test report.
-- The ability of exception handling.

- **Modifiability (Maintainability)**

--The ease with which a component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.
-- The ease with which software can be maintained, for example, enhanced, adapted, or corrected to satisfy specified requirements.
-- Modifiable with minimal impact.

- **Documentation**

-- Contains all documents necessary.

- **Fault Tolerance (Reliability)**

-- The ability of a component tolerates wrong inputs.

5.5 A Draft Quality Assurance Model for Component-Based Software Systems

Because of the different process of component-based software from traditional software, the quality assurance model should address both the process of components and the total systems. Figure 5.2 illustrates this view.

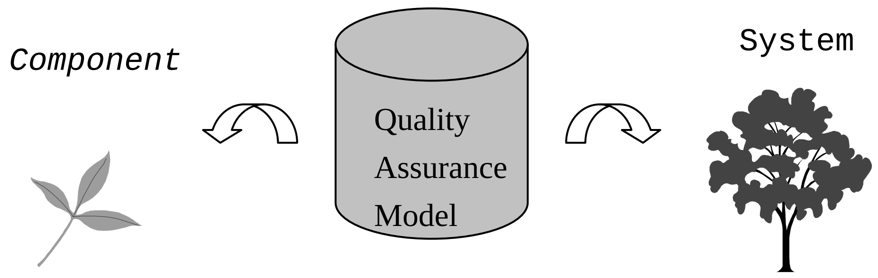


Figure 5.2 Quality Assurance Model for both components and systems

The main practices related to components and systems should contain:

1) *Component requirement analysis*

   the process of discovering, understanding, documenting, validating and managing the requirements for a component.

2) *Component development*

   the process of transferring the requirements to a well-functional component with multiple interfaces.

3) *Component certification*

   the process that involves:

   - component outsourcing: managing a component outsourcing contract and auditing the contractor performance;
   - component selecting: selecting the right components in accordance to the requirement;
   - component testing: confirm the component satisfies the requirement with acceptable quality and reliability;

4) *Component customization*

   the process that involves:

   - modifying the component for the specific requirement;
   - doing necessary changes to run the component on specific flatform;
   - upgrading the specific component to get a better performance or a higher quality;

5) *System architecture design*

   the process of evaluating, selecting and creating software architecture of a component-based system.

6) *System integration*

   the process of assembling components selected into a whole system.

7) *System testing*

   the process of evaluating a system to:

   - confirm that the system satisfies specified requirements;

- identify and correct defects in the system before implementation.

8) *System maintenance*

   the process of providing operations and maintenance activities needed to use the software effectively after it has been delivered.

Practice overview is listed below. For consistency, each practice is described under the heading of Definition, Objectives, Governing Policy and Process Overview Diagram.

5.5.1 Component Requirement Analysis

5.5.1.1 Definition

Component requirement analysis is the process of discovering, understanding, documenting, validating and managing the requirements for a component.

5.5.1.2 Objectives

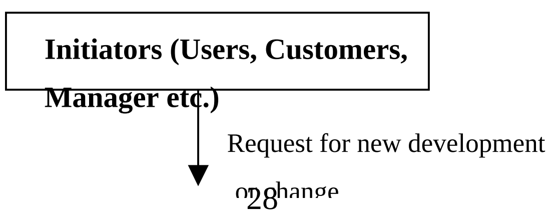
The objectives of component requirement analysis are to produce complete, consistent and relevant requirements that a component should realize.

5.5.1.3 Governing Policy

Component requirement analysis should contain complete and clear requirements that a component should realize, as well as the programming language, the platform and the interfaces related to the component.

5.5.1.4 Process Overview Diagram

See Figure 5.3.



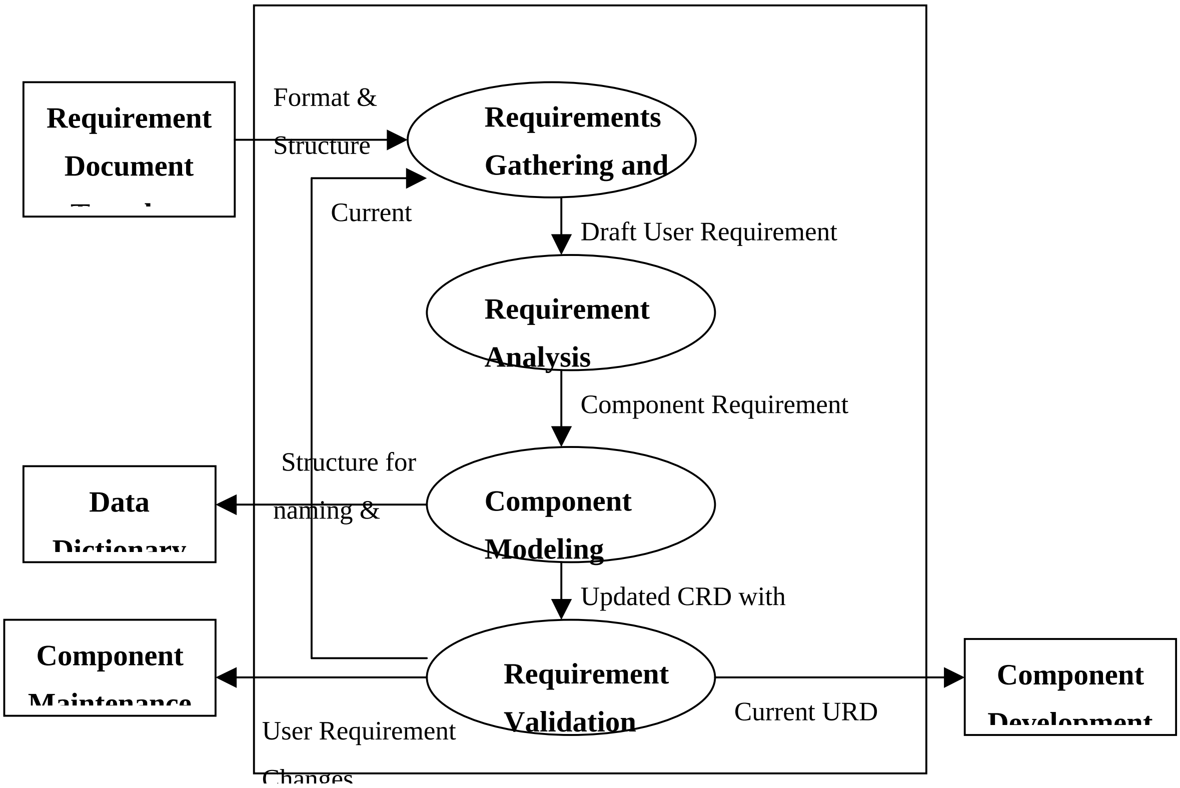


Figure 5.3 Component Requirement Analysis Process overview

5.5.2 Component Development

5.5.2.1 Definition

Component development is the process of implementing the requirements to a well-functional, high-quality component with multiple interfaces.

5.5.2.2 Objectives

The objectives of component development are the final component product, the interfaces and development documents.

5.5.2.3 Governing Policy

Component development should lead to the final component satisfying the requirements with correct and expected result, well-defined and flexible interfaces.

5.5.2.4 Process Overview Diagram

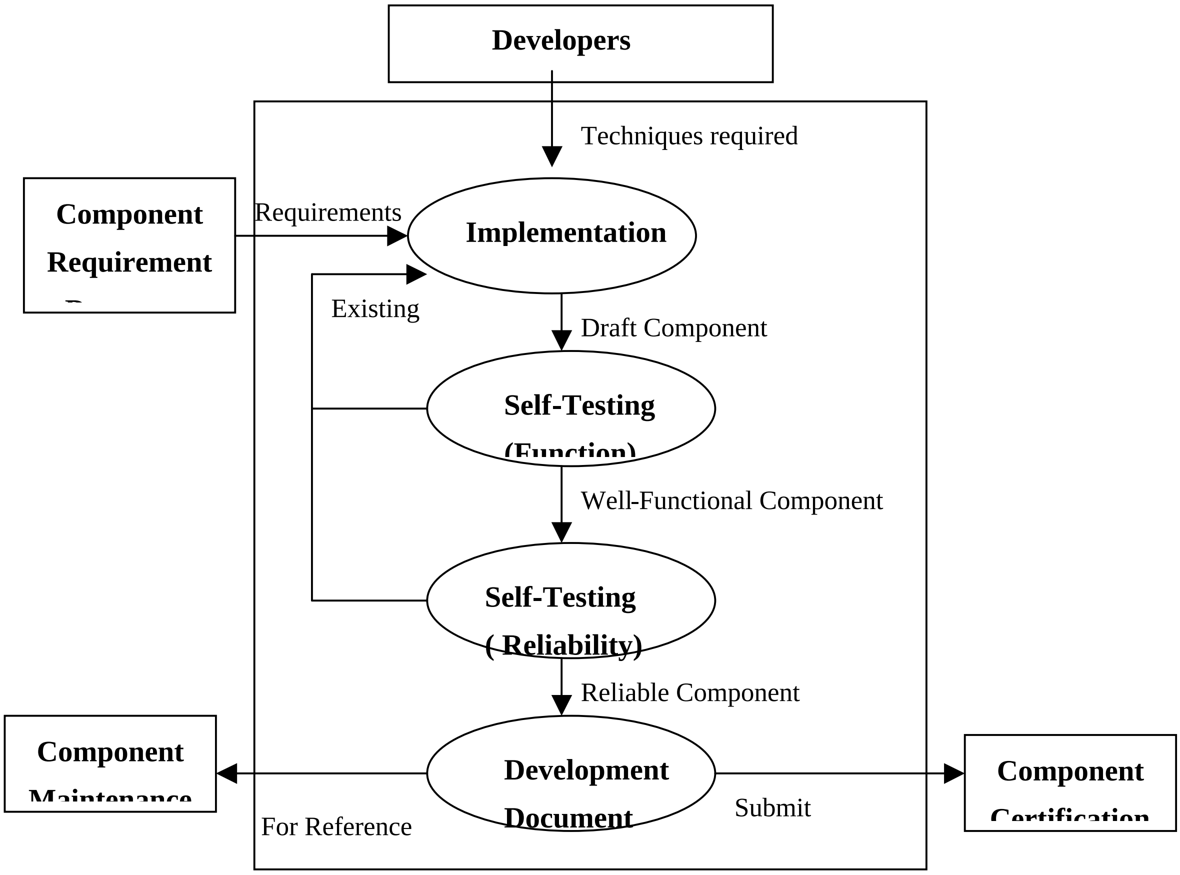


Figure 5.4 Component Development Process overview

5.5.3 Component Certification

5.5.3.1 Definition

Component certification is the process that involves:

- <u>component outsourcing</u>: managing a component outsourcing contract and auditing the contractor performance;
- <u>component selecting</u>: selecting the right components in accordance to the requirement for both function and reliability;
- <u>component testing</u>: confirm the component satisfies the requirement with acceptable quality and reliability;

5.5.3.2 Objectives

The objectives of component certification are to outsourcing, selecting and testing the candidate components to check whether they satisfy the system requirement and achieve the high quality and reliability.

5.5.3.3 Governing Policy

1. Component outsourcing should be charged by a Software Contract Manager;
2. All candidate components should be tested to be free from all known defects;
3. Testing should be in the target or simulated environment.

5.5.3.4 Process Overview Diagram

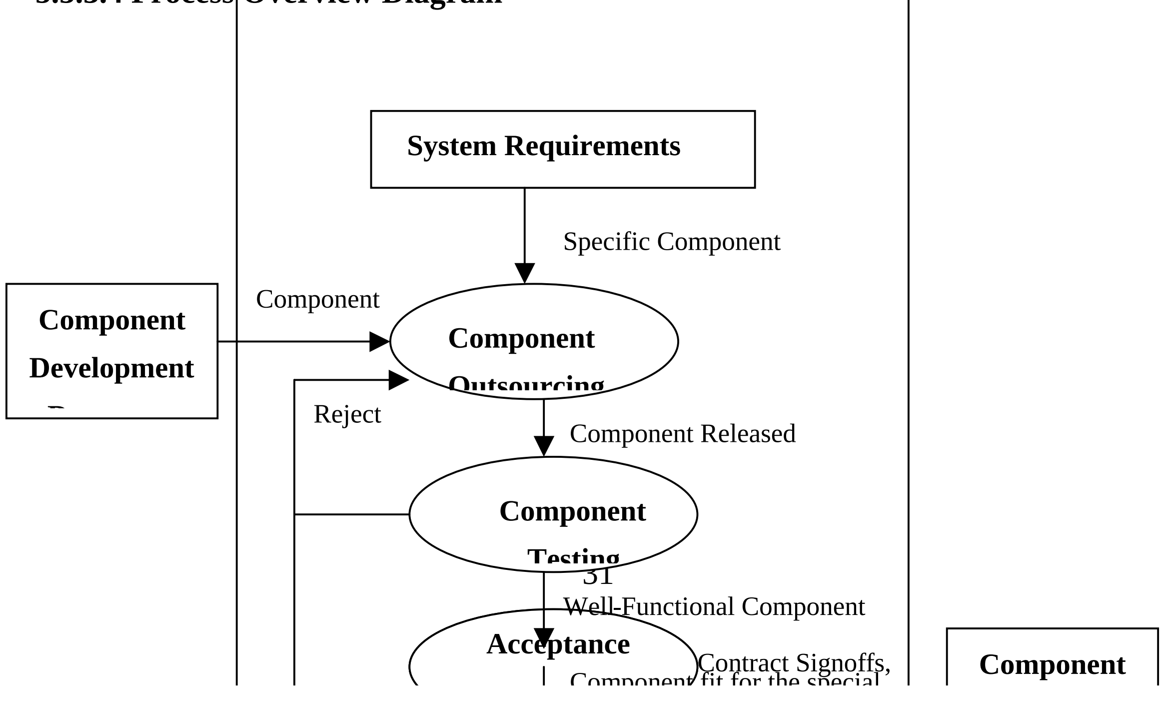


Figure 5.5 Component Certification Process overview

5.5.4 Component Customization

5.5.4.1 Definition

Component customization is the process that involves:

- modifying the component for the specific requirement;
- doing necessary changes to run the component on special flatform;
- upgrading the specific component to get a better performance or a higher quality;

5.5.4.2 Objectives

The objectives of component customization are to making necessary changes to a developed component so that it can be used in specific environment or cooperate with other components well.

5.5.4.3 Governing Policy

All components must be customized according to the system requirements on environment or the requirements of other components with which the components should work.

5.5.4.4 Process Overview Diagram

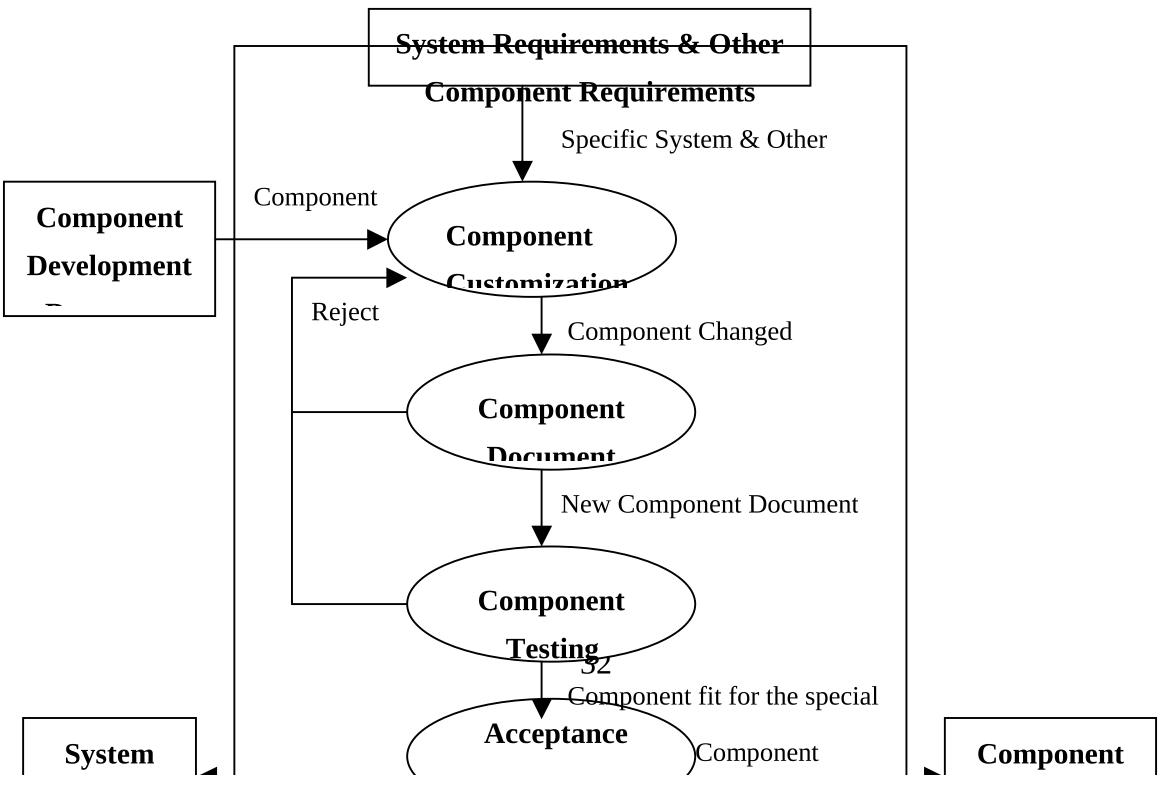


Figure 5.6 Component Customization Process overview

5.5.5 System Architecture Design

5.5.5.1 Definition

System architecture design is the process of evaluating, selecting and creating software architecture of a component-based system.

5.5.5.2 Objectives

The objectives of s ystem architecture design are to collecting the users requirement, identifying the system specification, selecting appropriate system architecture, and determining the implementation details such as platform, programming language and etc.

5.5.5.3 Governing Policy

System architecture design should address the advantage for the selecting architecture from other architectures.

5.5.5.4 Process Overview Diagram

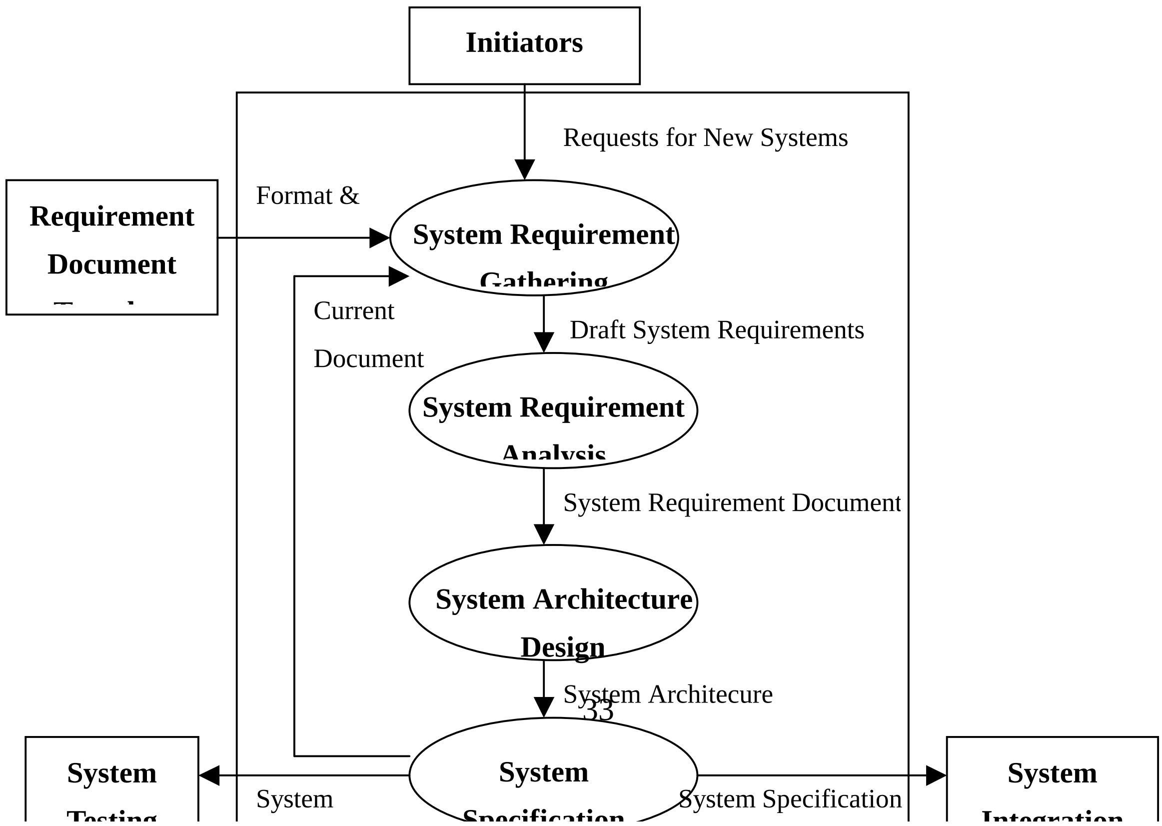


Figure 5.7 System Architecture Design Process overview

5.5.6 System Integration

5.5.6.1 Definition

System integration is the process of assembling components selected into a whole system under the designed system architecture.

5.5.6.2 Objectives

The objective of system integration is the final system composed by the components selected.

5.5.6.3 Governing Policy

System integration should

5.5.6.4 Process Overview Diagram

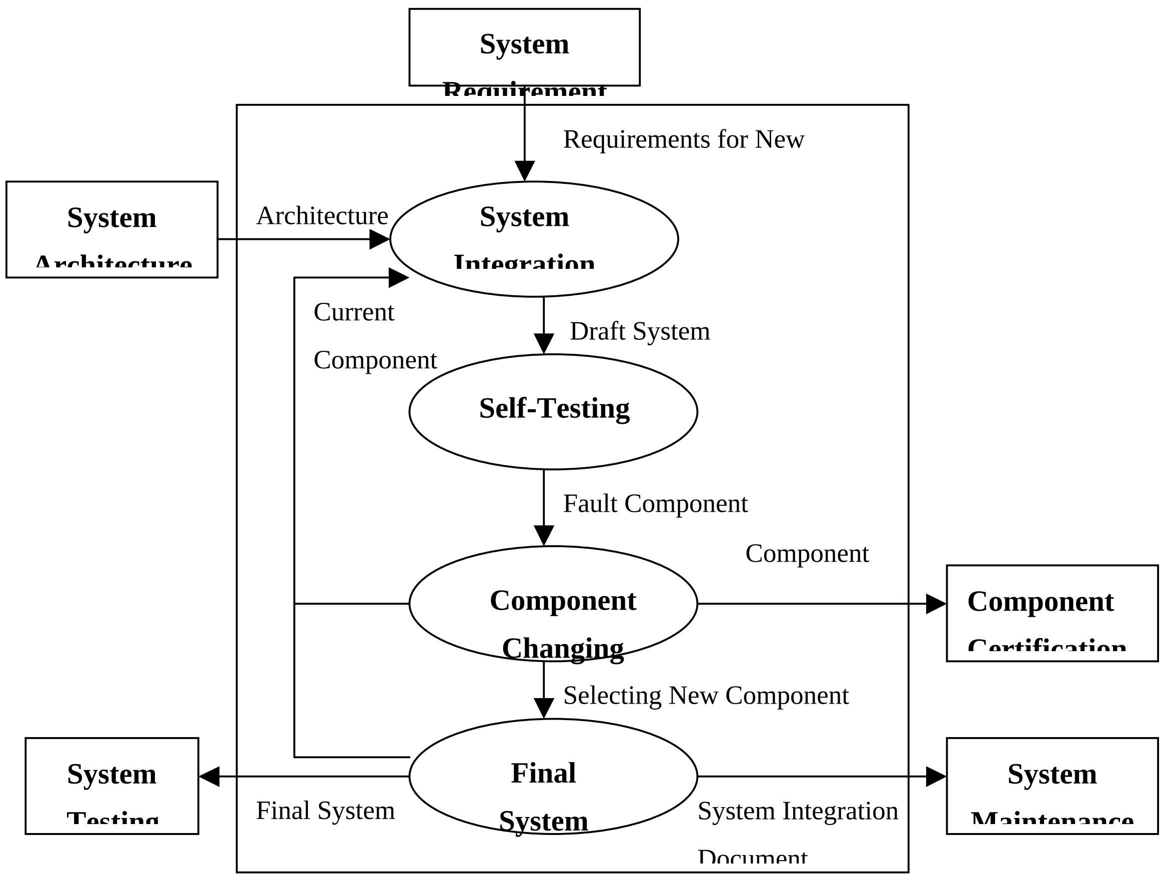


Figure 5.8 System Integration Process overview

5.5.7 System Testing

5.5.7.1 Definition

System testing is the process of evaluating a system to:

- confirm that the system satisfies specified requirements;
- identify and correct defects in the system before implementation.

5.5.7.2 Objectives

The objective of system testing is the final system integrated by components selected in accordance to the system requirements.

5.5.7.3 Governing Policy

System testing should contain function testing and reliability testing.

5.5.7.4 Process Overview Diagram

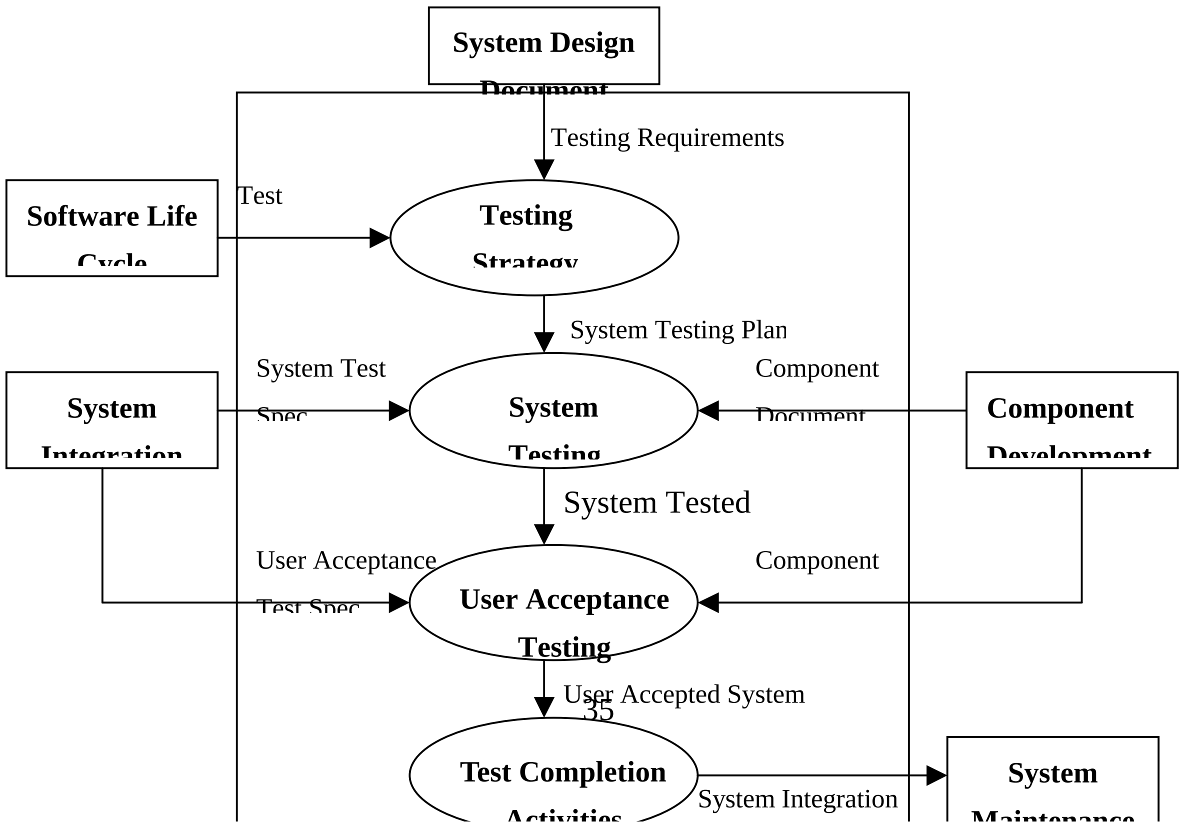


Figure 5.9 System Testing Process overview

5.5.8 System Maintenance

5.5.8.1 Definition

System maintenance is the process of providing service and maintenance activities needed to use the software effectively after it has been delivered.

5.5.8.2 Objectives

The objectives of system maintenance are to provide an effective product or service to the end-users while correcting faults, improve performance or other attributes or adapt to a changed environment to keep the software usable and useful after it has been delivered.

5.5.8.3 Governing Policy

There shall be a maintenance organization for every software product in operational use. All changes about the system delivered should be reflected in the related documents.

5.5.8.4 Process Overview Diagram

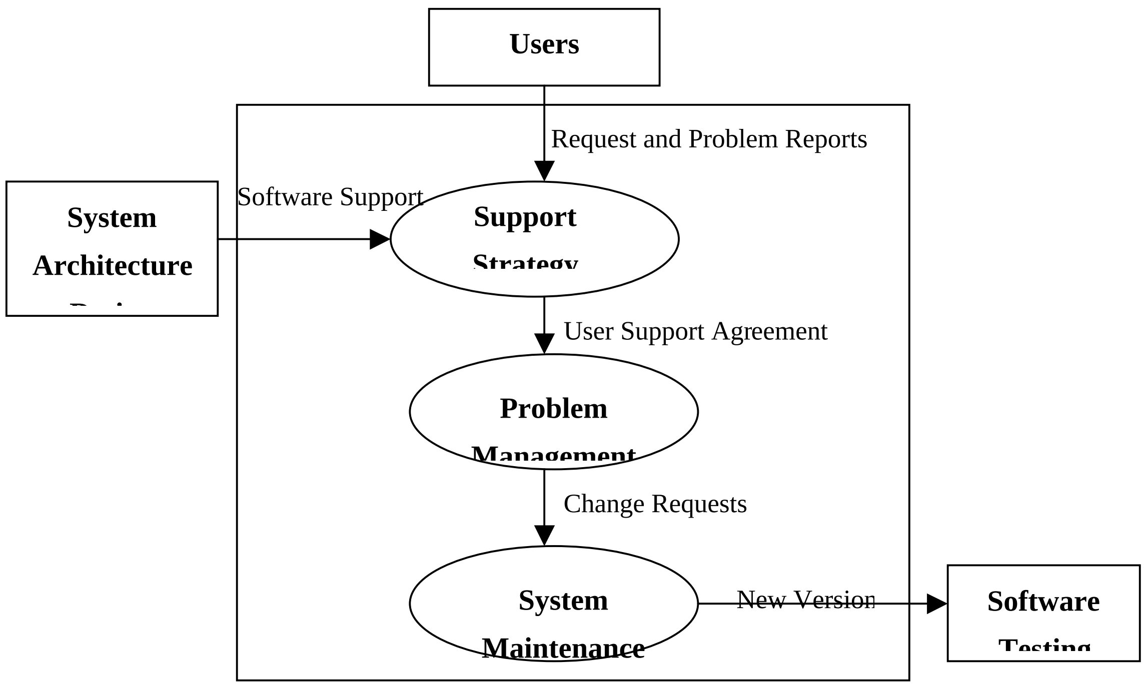


Figure 5.10 System Maintenance Process overview

6. Conclusion and Future Work

Component-Base Software Development is a new promising software development approach, which has potential to reduce significantly development cost and time-to-market, and improve maintainability, reliability and overall quality of application. Because this approach is developing systems by selecting off-the-shelf components and assembling them with an appropriate software architecture, it is much different with the traditional ones. Quality Assurance is very important for component-based software systems, especially when the components come from different developers.

In this paper, a survey is done on current component-based software technologies and the features they have. The survey is also about Quality Assurance for both traditional approach and object-oriented technology. At last, I propose some features and a simple draft of Quality Assurance Model for component-based software development.

My future work is to complement the draft Quality Assurance model so that it can actually guide the practices of component-based software development; and to find out whether there are some testing tools and metrics available to test software components under certain component technology.

7. References

[Adle95] R.M.Adler, "Emerging Standards for Component Software," Computer, Volume: 28 3, March 1995 , pp. 68 –77.

[Brow98] A.W.Brown, K.C. Wallnau, "The Current State of CBSE," IEEE Software Volume: 15 5, Sept.-Oct. 1998, pp. 37 – 46.

[Grah97] I.Graham, B.Henderson-Sellers, H.Younessi, "The OPEN Process Specification," Addison-Wesley, 1997.

[Gris97] M. L. Griss, "Software Reuse Architecture, Process, and Organization for Business Success," Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering, 1997, pp. 86-98.

[Hend98] B.Henderson-Sellers, A.J.H.Simons, H.Youness, "The OPEN Toolbox of Techniques," Addison-Wesley, 1998.

[Hend99] B.Henderson-Sellers, "OO software process improvement with metrics," Proceedings of Sixth International Software Metrics Symposium, 1999, pp. 2 -8.

[Herz00] P.Herzum, O.Slims, "Business Component Factory - A Comprehensive Overview of Component-Based Development for the Enterprise," OMG Press, 2000.

[HKPC00] Hong Kong Productivity Council, http://www.hkpc.org/itd/servic11.htm, April, 2000

[IBM00] IBM: http://www-4.ibm.com/software/ad/sanfrancisco, Mar. 2000

[Jaco92] I.Jacobson, M. Christerson, P.Jonsson, G. Overgaard, "Object-Oriented Software Engineering: A Use Case Driven Approach," Addison-Wesley Publishing Company, 1992

[Koza98] W. Kozaczynski, G. Booch, "Component-Based Software Engineering," IEEE Software Volume: 15 5 , Sept.-Oct. 1998 , pp. 34 –36.

[Kung98] D.C.Kung,, H.Pei, Y.Toyoshima, C.Chen, J.Gao, "Object-oriented software testing-some research and development," Proceedings of Third IEEE International High-Assurance Systems Engineering Symposium, 1998, pp. 158 -165.

[Lam97] W. Lam, A.J.Vickers, "Managing the Risks of Component-Based Software Engineering," Proceedings Fifth International Symposium on Assessment of Software Tools and Technologies, 1997, pp. 123–132.

[Lo98] B.W.N. Lo, Haifeng Shi, "A preliminary testability model for object-oriented software," Proceedings of International Conference for Software Engineering: Education & Practice, 1998, pp. 330 -337.

[Meta00] Metamata: http://www.metamata.com/products/metrics_top.html, Mar,2000.

[Micr00] Microsoft: http://www.microsoft.com/isapi, Mar, 2000

[Ning94] J.Q. Ning, K. Miriyala, W. Kozaczynski,, "An Architecture-Driven, Business-Specific, and Component-Based Approach to Software Engineering," Proceedings Third International Conference on Software Reuse: Advances in Software Reusability, 1994, pp. 84-93

[Ning97] J. Q. Ning, "Component-Based Software Engineering (CBSE)," Proceedings Fifth International Symposium on Assessment of Software Tools and Technologies, 1997, pp. 143-148.

[OMG00] OMG: http://www.omg.org/corba/whatiscorba.html, Mar. 2000

[Pour98] G. Pour, "Component-Based Software Development Approach: New Opportunities and Challenges," Proceedings Technology of Object-Oriented Languages, 1998. TOOLS 26., pp. 375-383.

[Pour99a] G. Pour, "Enterprise JavaBeans, JavaBeans & XML Expanding the Possibilities for Web-Based Enterprise Application Development," Proceedings Technology of Object-Oriented Languages and Systems, 1999, TOOLS 31, pp.282-291.

[Pour99b] G.Pour, M. Griss, J. Favaro, "Making the Transition to Component-Based Enterprise Software Development: Overcoming the Obstacles – Patterns for Success," Proceedings of Technology of Object-Oriented Languages and systems, 1999, pp.419 – 419.

[Pour99c] G.Pour, "Software Component Technologies: JavaBeans and ActiveX," Proceedings of Technology of Object-Oriented Languages and systems, 1999, pp. 398 – 398.

[Raja92a] C. Rajaraman, M.R. Lyu,"Reliability and Maintainability Related Software Coupling Metrics in C++ Programs," Proceedings 3rd IEEE International Symposium on Software ReliabilityEngineering (ISSRE'92), 1992, pp. 303-311.

[Raja92b] C. Rajaraman, M.R. Lyu, "Some Coupling Measures for C++ Programs," Proceedings TOOLS USA 92 Conference, August 1992, pp. 225-234.

[Sand94] J.Sanders, E.Curran, "Software Quality: A Framework for Success in Software Development and Support," Addison-Wesley Publishing Company, 1994

[Schm94] C.H.Schmauch, "ISO9000 for Software Developers,", ASQC Quality Press, 1994

[Shih97] T.K. Shih, C.M.Chung, C.C.Wang, W.C Pai,Decomposition of inheritance hierarchy DAGs for object-oriented software metrics," Proceedings of International Conference and Workshop on Engineering of Computer-Based Systems, 1997, pp. 238 -245.

[Smit95] D.J.Smith, “Achieving Quality Software(Third Edition),” Chapman & Hall, 1995

[SUN00] SUN : http://developer.java.sun.com/developer/technicalArticles/Bean, Mar. 2000

[Tran97] V. Tran, D.B. Liu, B.Hummel, “Component-Based Systems Development: Challenges and Lessons Learned, Software,” Proceedings, Eighth IEEE International Workshop on Technology and Engineering Practice [incorporating Computer Aided Software Engineering], 1997, pp. 452 – 462.

[Wang97] Y.M.Wang, O.P.Damani, W.J. Lee, “Reliability and Availability Issues in Distributed Component Ojbect Model (DCOM),” Fourth International Workshop on Community Networking Proceedings, 1997, pp. 59 –63.

[Yaco99a] S.M. Yacoub, B. Cukic, H.H. Ammar, “A Component-Based Approach to Reliability Analysis of Distributed Systems,” Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems, 1999, pp. 158 –167.

[Yaco99b] S.M.Yacoub, B. Cukic, H.H.Ammar, “A Scenario-Based Reliability Analysis of Component-Based Software,” Proceedings 10th International Symposium on Software Reliability Engineering, 1999, pp. 22 –31.

[Yau98] S.S.Yau, B. Xia, “Object-Oriented Distributed Component Software Development based on CORBA,” Proceedings of COMPSAC’98. The Twenty-Second Annual International, 1998. pp. 246-251.