Quality Prediction for Component-Based Software Development: Techniques and A Generic Environment

CAI Xia

A Thesis Submitted in Partial Fulfillment
Of the Requirements for the Degree of
Master of Philosophy
In
Computer Science and Engineering

Supervised by:
Prof. Michael R. Lyu

December 2001

Abstract

In this thesis, we address quality assurance issues in component-based software development. First, we propose a quality assurance (QA) model for component-based software development (CBSD), which covers eight main processes in CBSD: component requirement analysis, component development, component certification, component customization, and system architecture design, integration, testing, and maintenance. We propose the Component-based Program Analysis and Reliability Evaluation (ComPARE) environment for evaluation of quality of component-based software systems. ComPARE automates the collection of different metrics, the selection of different prediction models, the formulation of user-defined models, and the validation of the established models according to fault data collected in the development process. Different from other existing tools, ComPARE takes dynamic metrics into account (such as code coverage and performance metrics), integrates them with process metrics and static code metrics for object-oriented programs (such as complexity metrics, coupling and cohesion metrics, inheritance metrics), and provides different models for integrating these metrics to an overall estimation with higher accuracy.

Also, we apply different quality prediction techniques to some real world component-based programs in real world. Based on the analysis, we conclude that the quality prediction models are suitable for component-based software systems.

Acknowledgements

I would like to take this opportunity to express my gratitude to my supervisor, Prof. Michael R. Lyu, for his generous guidance and patience given to me in the past two years. His numerous support and encouragement, as well as his inspiring advice are extremely essential and valuable in my research work. Also, I am grateful for his support and advice for my further study.

Many thanks go to Prof. Kam-Fai Wong, without whose support and valuable advice I would not step into the component-based software development field. Thanks also go to the colleagues of Open Component Foundation, for their helpful sharing and discussions.

I am so grateful to Prof. Ada Fu and Prof. Mei Hwa Chen for their precious time to serve as my thesis examiners.

I would like to thank all the friends I made here. It is their friendship and encouragement that made my study and life happier and easier. I am also grateful for all the staff in the department, whose smiles and hard work provide a pleasant environment for study and research.

Last but not least, I would like to thank my husband Xuetai Zhang, for his unending love, patience and support. Also thanks to our child to be born, who as a gift from God, gives me courage and strength to finish this thesis.

Content

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Component-Based Software Development and Quality Assurance Issues

Modern software systems become more and more large-scale, complex and uneasily controlled, resulting in high development cost, low productivity, unmanageable software quality and high risk to move to new technology [15]. Consequently, there is a growing demand of searching for a new, efficient, and cost-effective software development paradigm.

One of the most promising solutions today is the component-based software development approach. This approach is based on the idea that software systems can be developed by selecting appropriate off-the-shelf components and then assembling them with a well-defined software architecture [12]. This new software development approach is very different from the traditional approach in which software systems can only be implemented from scratch. These commercial off-the-shelf (COTS) components can be developed by different developers using different languages and different platforms. This can be shown in Figure 1.1, where COTS components can be

checked out from a component repository, and assembled into a target software system.

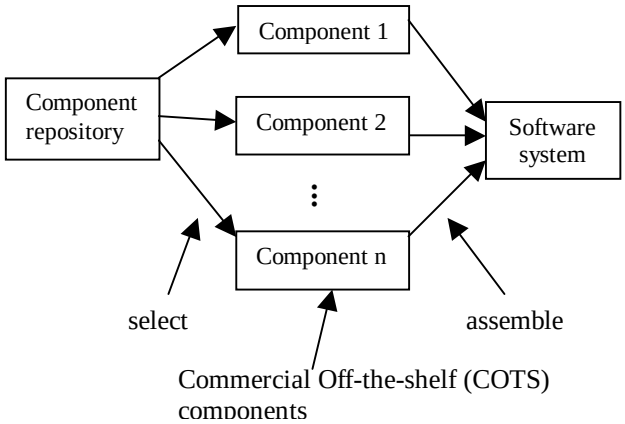


Figure 1.1 Component-based software development

Component-based software development (CBSD) can significantly reduce development cost and time-to-market, and improve maintainability, reliability and overall quality of software systems [13,14]. This approach has raised a tremendous amount of interests both in the research community and in the software industry. The life cycle and software engineering model of CBSD is much different from that of the traditional ones. This is what the Component-Based Software Engineering (CBSE) is focused.

To ensure that a component-based software system can run properly and effectively, the system architecture is the most important factor. According to both research community [2] and industry practice [5], the system architecture of component-based software systems should be layered and modular. This architecture can be seen in Figure 1.2. The top application layer is the application systems

supporting a business. The second layer consists of components engaged in only a specific business or application domain, including components usable in more than a single application. The third layer is cross-business middleware components consisting of common software and interfaces to other established entities. Finally, the lowest layer of system software components includes basic components that interface with the underlying operating systems and hardware.

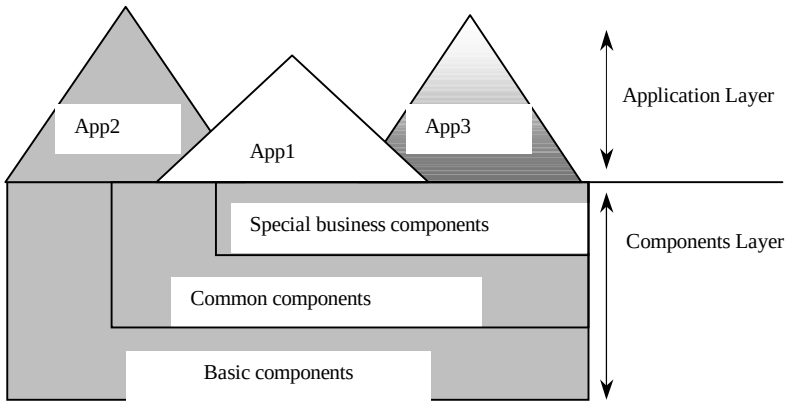


Figure 1.2 System architecture of component-based software systems

Current component technologies have been used to implement different software systems, such as object-oriented distributed component software [23], Web-based enterprise application [13] and embedded software systems [40]. There are also some commercial players involved in the software component revolution, such as BEA, Microsoft, IBM and Sun [7]. A noticeable example is the IBM SanFrancisco project. It provides a reusable distributed object infrastructure and an abundant set of application components to application developers [5].

Up to now, software component technologies are an emerging technology, which

is far from being matured. There is no existing standards or guidelines in this new area, and we do not even have a unified definition of the key item "component". In general, however, a component has three main features: 1) a component is an independent and replaceable part of a system that fulfills a clear function; 2) a component works in the context of a well-defined architecture; and 3) a component communicates with other components by its interfaces [1].

As CBSD is to build software systems using a combination of components including off-the-shelf components, components developed in-house and components developed contractually, the over quality of the final system greatly depends on the quality of the selected components. We need to first measure the quality of a component before we can certify it. Software metrics are designed to measure different attributes of a software system and development process, indicating different levels of quality in the final product [24]. Many metrics such as process metrics, static code metrics and dynamic metrics can be used to predict the quality rating of software components at different development phases [24,26]. For example, code complexity metrics, reliability estimates, or metrics for the degree of code coverage achieved have been suggested. Test thoroughness metric is also introduced to predict a component's ability to hide faults during tests [25].

In order to make use of the results of software metrics, several different techniques have been developed to describe the predictive relationship between software metrics and the classification of the software components into fault-prone and non fault-prone categories [27]. These techniques include discriminant analysis [30], classification

trees [31], pattern recognition [32], Bayesian network [33], case-based reasoning (CBR) [34], and regression tree models [27]. There are also some prototype or tools [36, 37] that use such techniques to automate the procedure of software quality prediction. However, these tools address only one kind of metrics, e.g., process metrics or static code metrics. Besides, they rely on only one prediction technique for the overall software quality assessment.

1.2 Our Main Contributions

From the above, we observe that conventional Software Quality Assurance (SQA) techniques are not applicable to CBSD. In this thesis, we propose an efficient and effective SQA approach for CBSD.

Our research have the following main contributions:

- We propose a QA model for component-based software development. It covers eight main processes in CBSD: component requirement analysis, component development, component certification, component customization, and system architecture design, integration, testing, and maintenance.
- We propose the Component-based Program Analysis and Reliability Evaluation (ComPARE) environment for evaluation of quality of component-based software systems. ComPARE automates the collection of different metrics, the selection of different prediction models, the formulation

of user-defined models, and the validation of the established models according to fault data collected in the development process. Different from other existing tools, ComPARE takes dynamic metrics into account (such as code coverage and performance metrics), integrates them with process metrics and static code metrics for object-oriented programs (such as complexity metrics, coupling and cohesion metrics, inheritance metrics), and provides different models for integrating these metrics to an overall estimation with higher accuracy.

- Also, we apply different quality predicted techniques to some real world component-based programs. From the results, we give some guidelines on current component-based software development.

1.3 Outline of This Thesis

First, we present the technical background and related works of CBSD and SQA in Chapter 2, including the current development frameworks for component based software: e.g., CORBA, COM/DCOM and JavaBeans, and quality assurance issues of CBSD, such as quality prediction techniques based on classification tree, case-based reasoning and Bayesian Network.

Chapter 3 covers the QA model we proposed, which addresses quality management issues in component-based software development process. In Chapter 4, we introduce a generic quality assessment environment called ComPARE to automate the

systematic procedure of quality assessment for CBSD. ComPARE simulates the process of selecting qualified components from a component repository as well as predicting and evaluating the final system based on these components.

Different predicting models have been applied to on some real world CORBA programs. Chapter 5 outlines the results and analyses. Based on the analysis, the advantages and disadvantages of these models are described. Finally we conclude our research work in Chapter 6.

Chapter 2

Technical Background and Related Work

Because our research topic is to investigate whether the conventional Software Quality Assurance (SQA) techniques are applicable to component-based software development (CBSD), we address our survey on current component technologies and QA issues in CBSD. As there are so many un-explored issues about QA of CBSD, we narrow our topic to quality prediction to evaluate and assess the quality of components in the component library.

In this chapter, we survey current development frameworks for CBSD and the features they have as well as some related QA issues. After that, we will introduce some quality prediction techniques that we would address in our research, and existing quality prediction tools that we should learn from.

2.1 Development Framework for Component-based Software

To employ component-based software development, we should know the current development frameworks for this approach. A framework can be defined as a set of constraints on components and their interaction, and a set of benefits that derive from

those constraints [42]. To identify the development framework for component-based software systems, the framework or infrastructure for components should be identified first, as components are the basic units in component-based software systems.

Some approaches, such as Visual Basic Controls (VBX), ActiveX controls, class libraries, and JavaBeans, make it possible for their related languages, such as Visual Basic, C++, Java, and the supporting tools to share and distribute application pieces. But all of these approaches rely on certain underlying services to provide the communication and coordination necessary for the application. The infrastructure of components (sometimes called a *component model*) acts as the "plumbing" that allows communication among components [1]. Among the component infrastructure technologies that have been developed, three have become somewhat standardized: OMG's CORBA, Microsoft's Component Object Model (COM) and Distributed COM (DCOM), and Sun's JavaBeans and Enterprise JavaBeans [7].

2.1.1 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 vendors and object technology user companies [11]. Simply stated, CORBA is a vendor-independent architecture and infrastructure that computer applications use to work together over networks. It manages details of component interoperability, and allows applications to communicate with one another despite of different locations and

designers. The interface is the only way that applications or components communicate with each other. Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network.

The most important part of a CORBA system is the Object Request Broker (ORB). The ORB is the middleware that establishes the client-server relationships between components. Using an ORB, a client can invoke a method on a server object, whose location is completely transparent. The ORB is responsible for intercepting a call and finding an object that can implement the request, pass its parameters, invoke its method, and return the results. The client does not need to know where the object is located, its programming language, its operating system, or any other system aspects that are not related to the interface. In this way, the ORB provides interoperability among applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

CORBA applications are composed of *objects*, individual units of running software that combine functionality and data, and that frequently (but not always) represent something in the real world. Typically, there are many *instances* of an object of a single *type* - for example, an e-commerce website would have many shopping cart object instances, all identical in functionality but differing in that each is assigned to a different customer, and contains data representing the merchandise that its particular

customer has selected. For other types, there may be only one instance. When a legacy application, such as an accounting system, is wrapped in code with CORBA interfaces and opened up to clients on the network, there is usually only one instance.

The IDL interface definition is independent of programming language, but maps to all of the popular programming languages via OMG standards: OMG has standardized mappings from IDL to C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDLscript. This separation of interface from implementation, enabled by OMG IDL, is the essence of CORBA - how it enables interoperability, with all of the transparencies we've claimed. The interface to each object is defined very strictly. In contrast, the implementation of an object - its running code, and its data - is hidden from the rest of the system (that is, encapsulated) behind a boundary that the client may not cross. Clients access objects only through their advertised interface, invoking only those operations that that the object exposes through its IDL interface, with only those parameters (input and output) that are included in the invocation.

In CORBA, every object instance has its own unique *object reference*, an identifying electronic token. Clients use the object references to direct their invocations, identifying to the ORB the exact instance they want to invoke (e.g., ensuring that the books you select go into your own shopping cart, and not into your neighbor's). The client acts as if it is invoking an operation on the object instance, but it is actually invoking on the IDL stub which acts as a proxy. Passing through the stub on the client side, the invocation continues through the ORB (Object Request Broker), and the skeleton on the implementation side, to get to the object where it is executed.

CORBA is widely used in Object-Oriented distributed systems [23] including component-based software systems because it offers a consistent distributed programming and run-time environment over common programming languages, operating systems, and distributed networks.

The OMG has also defined two standards for embedded applications: Minimum CORBA and Real-Time CORBA. Minimum CORBA defines a standard, fully interoperable subset (profile) of CORBA functionality that is appropriate for resource-constraint applications, while Real-Time CORBA extends CORBA so that it can be used to build deterministic applications [28].

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

Introduced in 1993, The Component Object Model (COM) is a software architecture that allows applications to be built from binary software components [9]. COM provides platform-dependent, based on Windows and Windows NT, and language-independent component-based applications.

COM defines how components and their clients interact. This interaction is defined such that the client and the component can connect without the need of any intermediate 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 [20].

COM is the underlying architecture that forms the foundation for higher-level

software services, like those provided by OLE. OLE services span various aspects of commonly needed system functionality, including compound documents, custom controls, interapplication scripting, data transfer, and other software interactions.

It is important to note that COM is a general architecture for component software. While Microsoft is applying COM to address specific areas such as controls, compound documents, automation, data transfer, storage and naming, and others, any developer can take advantage of the structure and foundation that COM provides.

Microsoft® Distributed COM (DCOM) extends the Component Object Model (COM) to support communication among objects on different computers—on a LAN, a WAN, or even the Internet. With DCOM, your application can be distributed at locations that make the most sense to your customer and to the application.

Because DCOM is a seamless evolution of COM, the world's leading component technology, you can take advantage of your existing investment in COM-based applications, components, tools, and knowledge to move into the world of standards-based distributed computing. As you do so, DCOM handles low-level details of network protocols so you can focus on your real business: providing great solutions to your customers.

DCOM is an extension of the Component Object Model (COM). COM defines how components 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. The client calls methods in the component without any overhead whatsoever.

Since DCOM is an inherently secure protocol, it can be used without being encapsulated in a virtual private network: DCOM applications can simply use the cheap, global TCP/IP network. Most companies do not provide direct Internet access to their desktop computers. All but some dedicated server machines are hidden behind a firewall that typically consists of protocol-level (port-based) and application-level (proxy servers) filters.

To summarize, as an extension of the Component Object Model (COM), Distributed COM (DCOM), is a protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner. DCOM is designed for use across multiple network transports, including Internet protocols such as HTTP. When a client and its component reside on different machines, DCOM simply replaces the local interprocess communication with a network protocol. Neither the client nor the component is aware the changes of the physical connections.

2.1.3 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 supports applications of multiple platforms, as well as reusable, client-side and server-side

components [19].

Java platform offers an efficient solution to the portability and security 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, including 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 robust server objects independent of operating systems, Web servers and database management servers.

The JavaBeans API makes it possible to write component software in the Java programming language. Components are self-contained, reusable software units that can be visually composed into composite components, applets, applications, and servlets using visual application builder tools. JavaBean components are known as *Beans*.

Components expose their features (for example, public methods and events) to builder tools for visual manipulation. A Bean's features are exposed because feature names adhere to specific ***design patterns***. A "JavaBeans-enabled" builder tool can then examine the Bean's patterns, discern its features, and expose those features for visual manipulation. A builder tool maintains Beans in a palette or toolbox. You can select a

Bean from the toolbox, drop it into a form, modify it's appearance and behavior, define its interaction with other Beans, and compose it and other Beans into an applet, application, or new Bean. All this can be done without writing a line of code.

Millions of developers around the world have already embraced the Java™ platform. The Java platform has opened up an entirely new world of opportunities for building fully portable network-aware applications. Yet many developers are not yet sure how best to take advantage of the capabilities and benefits the Java platform delivers without sacrificing their existing investment in legacy applications.

The JavaBeans component architecture is a platform-neutral architecture for the Java application environment. It's the ideal choice for developing or assembling network-aware solutions for heterogeneous hardware and operating system environments--within the enterprise or across the Internet.

The JavaBeans component architecture extends "Write Once, Run Anywhere™" capability to reusable component development. In fact, the JavaBeans architecture takes interoperability a major step forward. Based on it, code can theoretically run on every OS and also within any application environment. A beans developer secures a future in the emerging network software market without losing customers that use proprietary platforms, because JavaBeans components interoperate with ActiveX. JavaBeans architecture connects via bridges into other component models such as ActiveX. Software components that use JavaBeans APIs are thus portable to containers including Internet Explorer, Visual Basic, Microsoft Word, Lotus Notes, and others.

The JavaBeans specification defines a set of standard component software APIs for the Java platform. The specification was developed by Sun with a number of leading industry partners and was then refined based on broad general input from developers, customers, and end-users during a public review period.

2.1.4 Comparison among Different Frameworks

Comparison among the development frameworks for component-based software systems above can be found in [1], [13] and [18]. Here we simply summarize these different features in Table 2.1.

	CORBA	EJB	COM/DCOM
Development environment	Underdeveloped	Emerging	Supported by a wide range of strong development environments
Binary interfacing standard	Not binary standards	Based on COM; Java specific	A binary standard for component interaction is the heart of COM
Compatibility & portability	Particularly strong in standardizing language bindings; but not so portable	Portable by Java language specification; but not very compatible.	Not having any concept of source-level standard of standard language binding.
Modification & maintenance	CORBA IDL for defining component interfaces, need extra modification & maintenance	Not involving IDL files, defining interfaces between component and container. Easier modification & maintenance.	Microsoft IDL for defining component interfaces, need extra modification & maintenance
Services provided	A full set of standardized services; lack of implementations	Neither standardized nor implemented	Recently supplemented by a number of key services
Platform dependency	Platform independent	Platform independent	Platform dependent
Language dependency	Language independent	Language dependent	Language independent
Implementation	Strongest for traditional enterprise computing	Strongest on general Web clients.	Strongest on the traditional desktop applications

Table 2.1 Comparison of development frameworks for component-based systems

2.2 Quality Assurance for Component-Based Systems

2.2.1 Traditional Quality Assurance Issues

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 [43], 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:

- 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.

- 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.

- 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.

- 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.

- Testing

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

- 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.

- Tools, Techniques and Methods

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

- 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 [42]:

- 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. [42].

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 [24]. Its emphasis is on the development process and the management responsibilities associated with the process. ISO9000-3 provides guidance on how to apply ISO 9000 standards to software development. The guidance is excellent and has been adopted widely by software community for 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 [43]. 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 [4]. 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.

2.2.2 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 an overall system from scratch, thus the life cycle of component-based software systems is different from that of the traditional software systems. The life cycle of component-based software systems can be summarized as follows [12]: 1) Requirements analysis; 2) Software architecture selection, construction, analysis, and evaluation; 3) Component identification and customization; 4) System integration; 4) System testing; 5) Software maintenance.

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

Quality assurance for component-based software systems should address the life

cycle and its key activities to analyze the components and achieve high quality component-based software systems. QA technologies for component-based software systems are currently premature, as the specific characteristics of component systems differ from those of traditional systems. Although some QA techniques such as reliability analysis model for distributed software systems [21,22] and component-based approach to Software Engineering [10] have been studied, 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 urgent needs.

2.2.3 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 attribute. 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 interface. This provides users with the flexibility to update components while maintaining only the interface and behavior of the components [3].

But as the component development is based on object-oriented programming, there are still objects, methods and classes in a component. So inheritance is also existed between objects inside a component.

2.2.4 Quality Characteristics of Components

As much work is yet to be done for component-based software development, QA technologies for component-based software development has to address the two inseparable parts: 1) How to assess quality of a component? 2) How to assess quality of the whole system based on components? 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, reusability and reliability of components and systems; and tools should be decided to test the existing components and systems.

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

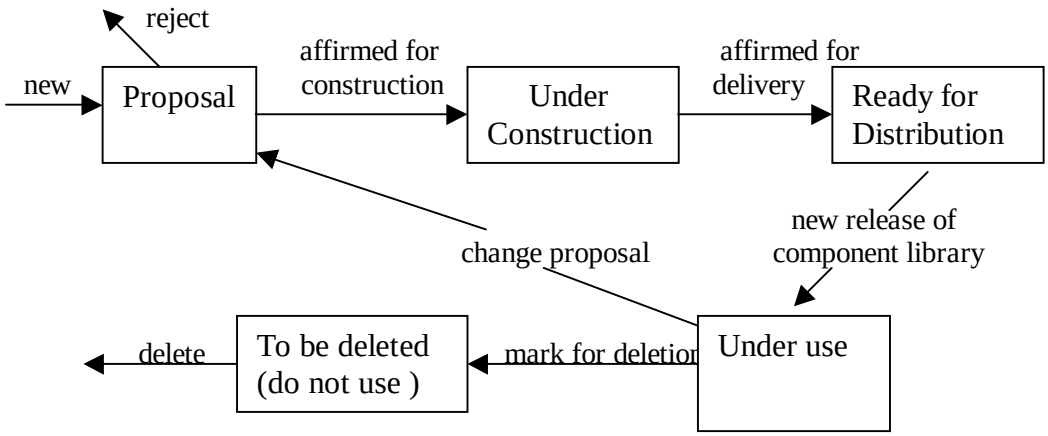


Figure 2.1 The life cycle of a component

A component has a life cycle as illustrated in Figure 2.1. Software metrics have been proposed to measure software complexity and to assure software quality [16,17]. Such metrics are often used to classify components [6]:

1) **Size.** This affects both reuse cost and quality. If it is too small, the benefits will not exceed the cost of managing it. If it is too large, it is hard to have high quality.
2) **Complexity.** This also affects reuse cost and quality. A over trivial component is not profitable to reuse while a over complex component is hard to inherit high quality.
3) **Reuse frequency.** The number of incidences where a component is used is a solid indicator of its usefulness.
4) **Reliability.** The probability of failure-free operations of a component under certain operational scenarios [8].

Based on the characteristics of Java and some widely used commercial off-the-shelf components, common metric suites have been defined, e.g., Metamata Metrics [28] and JProbe Metrics [29] .

Metamata Metrics calculates global complexity and quality metrics statically from Java source code. It helps organize code in a more structured manner, facilitates the QA process [28] and supports the following:

- Most standard object oriented metrics such as object coupling and object cohesion
- Traditional software metrics such as cyclomatic complexity and lines of code
- Applicable to incomplete Java programs or programs with errors, then it

could be used from day one of the development cycle

- Metrics acquisition at any level of granularity (methods, classes...)
- Statistical aggregations (mean, median...)
- JDK 1.1 and JDK 1.2 compatibility.

Table 2.2 is the examples of Metamata Metric98s:

Metric	Measures	Description
Cyclomatic Complexity	Complexity	The amount of decision logic in the code
Lines of Code	Understandability, maintainability	The length of the code; related metrics measure lines of comments, effective lines of code, etc.
Weighted Methods per Class	Complexity, understandability, 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
Coupling Between Objects	Design, reusability, maintainability	The number of other classes to which a class is coupled
Depth of Inheritance Tree	Reusability, testability	The depth of a class within the inheritance hierarchy
Number of Attributes	Complexity, maintainability	The amount of state a class maintains as represented by the number of fields declared in the class

Table 2.2. Examples of Metamata Metrics

JProbe from KL Group has different suites of metrics/tools for different purpose of use [29]. They are designed to help developers build robust, reliable, high-speed business applications in Java. Here is what the JProbe Developer Suite includes:

- *JProbe Profiler and Memory Debugger* - eliminates performance bottlenecks and memory leaks in Java code
- *JProbe Threadalyzer* - detects deadlocks, stalls and race conditions
- *JProbe Coverage* - locates and measures untested Java code.

JProbe Developer Suite paints an intuitive, graphical picture of everything from memory usage to calling relationships, helping the programmer navigate to the root of the problem quickly and easily.

Metamata metrics and Jprobe suites are both used in the QA Lab of Flashline, an industry leader in providing software component products, services and resources that facilitate rapid development of software systems for business applications. We use the result of such metrics in our risk analysis and evaluation tool, which is based on the idea of ARMOR (see section 2.3.2).

2.3 Quality Prediction Techniques

In order to predict the quality of different software components, several techniques have been developed to classify software components according to their reliability [27]. These techniques include discriminant analysis [30], classification trees [31], pattern recognition [32], Bayesian network [33], case-based reasoning (CBR) [34] and regression tree model [37]. Details of some of the prediction techniques are mentioned in section 4.3.

2.3.1 ARMOR: A Software Risk Analysis Tool

As we have mentioned before, there are a lot of metrics and tools to measure and test the quality of a software system. But little of them can integrate the various metrics together and compare the different results of these metrics, so that they can predict the quality as well as the risk of the software.

ARMOR(Analyzer of Reducing Module Operational Risk) is such a tool that is developed by Bell Lab in 1995 [36]. ARMOR can automatically identify the operational risks of software program modules. It takes data directly from project database, failure database, and program development database, establishes risk models according to several risk analysis schemes, determines the risks of software programs, and display various statistical quantities for project management and engineering decisions. The tool can perform the following tasks during project development, testing, and operation: 1) to establish promising risk models for the project under evaluation; 2) to measure the risks of software programs within the project; 3) to identify the source of risks and indicates how to improve software programs to reduce their risk levels; and 4) to determine the validity of risk models from field data.

ARMOR is designed for automating the procedure for the collection of software metrics, the selection of risk models, and the validation of established models. It provided the missing link of both performing sophisticated risk modeling and validate risk models against software failure data by various statistical techniques.

Figure 2.2 shows the high-level architecture for ARMOR.

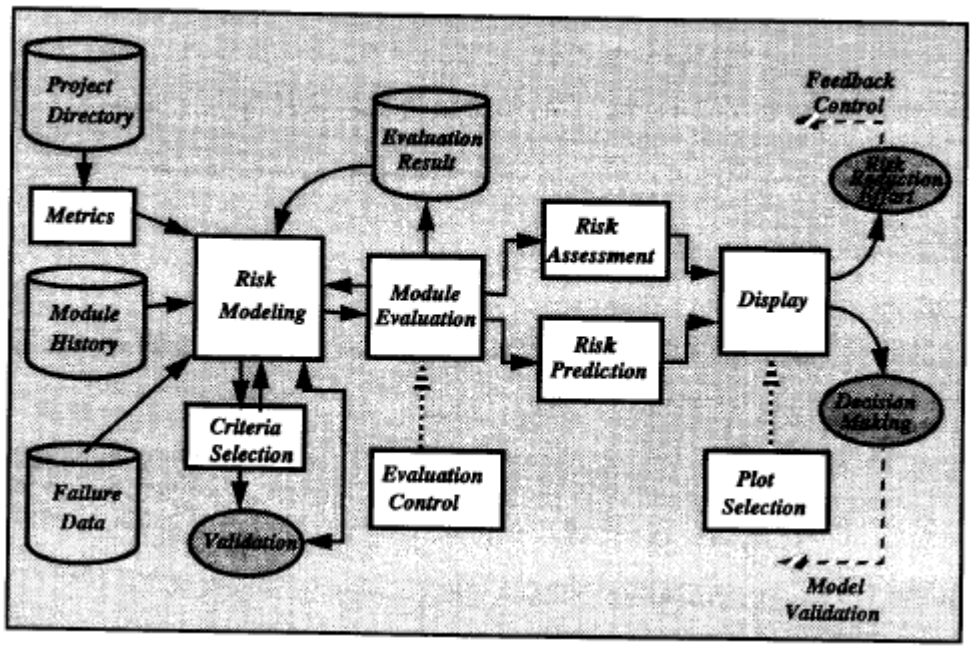


Figure 2.2 High-level architecture for ARMOR

ARMOR can be used:

- To access and compute software data deemed pertinent to software characteristics.
- To compute product metrics automatically whenever possible.
- To evaluate software metrics systematically.
- To perform risk modeling in a user-friendly and user-flexible fashion.
- To display risks of software modules.
- To validate risk models against actual failure data and compare model performance.
- To identify risky modules and to indicate ways for reducing software risks.

Chapter 3
A Quality Assurance Model for CBSD

Many standards and guidelines are used to control the quality activities of software development process, such as ISO9001 and CMM model. In particular, Hong Kong productivity Council has developed the HKSQA model to localize the general SQA models [4]. HKSQA model is a framework of standard practices that a software organization in Hong Kong should follow to produce quality software. The HK Software Quality Assurance Model provides the standard for local software organisations (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.

HKSQA model provides the details of procedures that are required to be followed for each of the seven model practices. These seven practices are:

- Software Project Management: the process of planning, organizing, staffing, monitoring, controlling and leading a software project.

- Software Testing: the process of evaluating a system where the software resides to:
    - confirm that the system satisfies specified requirements;
    - identify and correct defects in the system before implementation.
- Software Outsourcing: the process that involves:
    - Establishing a software outsourcing contract (SOC);
    - Selecting contractor(s) to fulfill the terms of the SOC;
    - Managing contractor(s) in accordance to the terms of the SOC;
    - Reviewing and auditing contractor performance based on results achieved;
    - Accepting the software product and/or service into production when it has been fully tested.
- Software Quality Assurance: a planned and systematic pattern of all actions necessary to provide adequate confidence that the item, product or service conforms to established customer and technical requirements.
- User Requirements Management: the process of discovering, understanding, negotiating, documenting, validating and managing a set of requirements for a computer-based system.
- Post Implementation Support: the process of providing operations and maintenance activities needed to use the software effectively after it has been

delivered.

- Software Change Control: the process of evaluating proposed changes to software configuration items and coordinating the implementation of approved changes to ensure that the integrity of the software remains intact and uncompromised.

In this section, we propose a framework of quality assurance model for the component-based software development paradigm.

Because component-based software systems are developed on an underlying process different from that of the traditional software, their quality assurance model should address both the process of components and the process of the overall system. Figure 3.1 illustrates this view.

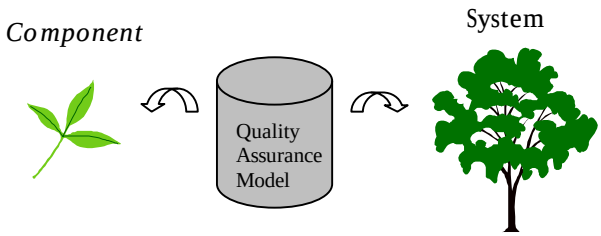


Figure 3.1 Quality assurance model for both components and systems

The main practices relating to components and systems in this model contain the following phases: 1) Component requirement analysis; 2) Component development; 3) Component certification; 4) Component customization; 5) System architecture design; 6) System integration; 7) System testing; and 8) System maintenance.

Details of these phases and their activities are described as follows.

3.1 Component Requirement Analysis

Component requirement analysis is the process of discovering, understanding, documenting, validating and managing the requirements for a component. The objectives of component requirement analysis are to produce complete, consistent and relevant requirements that a component should realize, as well as the programming language, the platform and the interfaces related to the component.

The component requirement process overview diagram is as shown in Figure 3.2. Initiated by the request of users or customers for new development or changes on old system, component requirement analysis consists of four main steps: requirements gathering and definition, requirement analysis, component modeling, and requirement validation. The output of this phase is the current user requirement documentation, which should be transferred to the next component development phase, and the user requirement changes for the system maintenance phase.

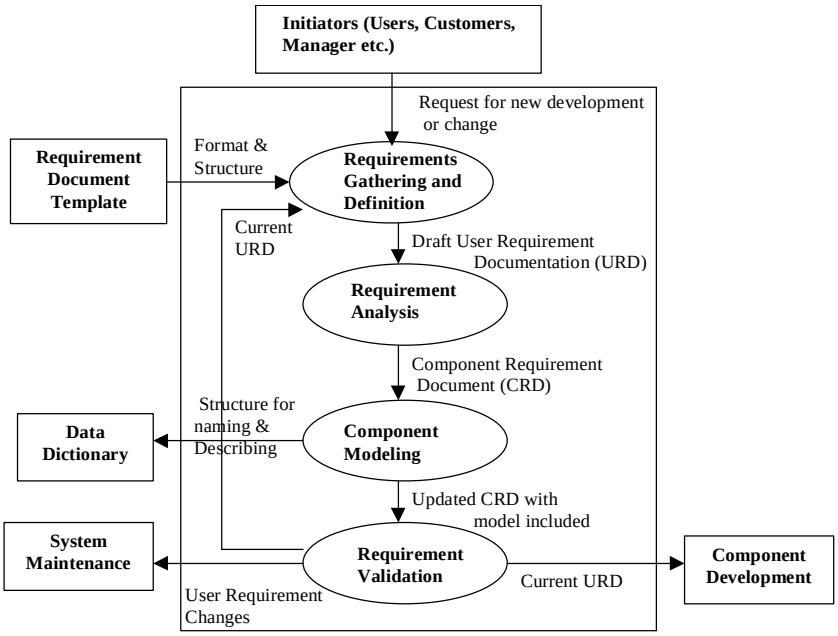


Figure 3.2 Component requirement analysis process overview

3.2 Component Development

Component development is the process of implementing the requirements for a well-functional, high quality component with multiple interfaces. The objectives of component development are the final component products, the interfaces, and development documents. Component development should lead to the final components satisfying the requirements with correct and expected results, well-defined behaviors, and flexible interfaces.

The component development process overview diagram is as shown in Figure 3.3. Component development consists of four procedures: implementation, function testing, reliability testing, and development document. The input to this phase is the component requirement document. The output should be the developed component and its documents, ready for the following phases of component certification and system maintenance, respectively.

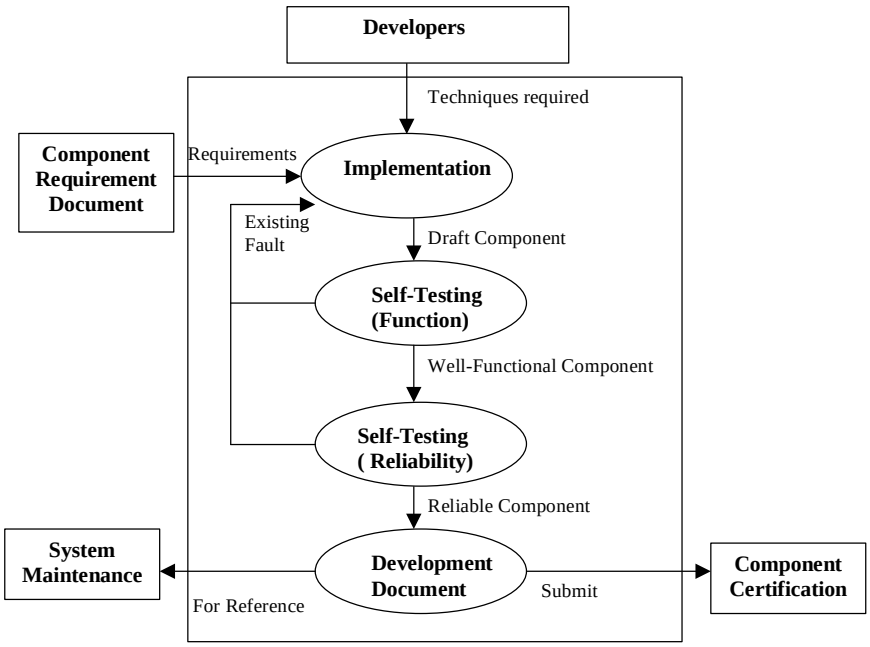


Figure 3.3 Component development process overview

3.3 Component Certification

Component certification is the process that involves: 1) *component outsourcing*: managing a component outsourcing contract and auditing the contractor performance; 2) *component selection*: selecting the right components in accordance to the requirement for both functionality and reliability; and 3) *component testing*: confirm the component satisfies the requirement with acceptable quality and reliability.

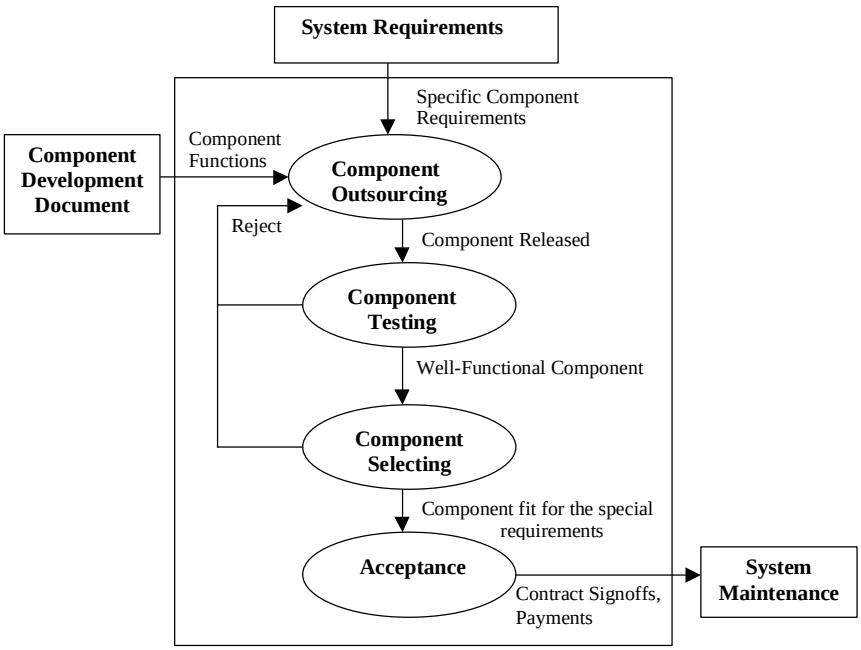


Figure 3.4 Component certification process overview

The objectives of component certification are to outsource, select and test the candidate components and check whether they satisfy the system requirement with high quality and reliability. The governing policies are: 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; and 3) Testing should be in the target environment or a simulated environment. The component certification process

overview diagram is as shown in Figure 3.4. The input to this phase should be component development document, and the output should be testing documentation for system maintenance.

3.4 Component Customization

Component customization is the process that involves 1) modifying the component for the specific requirement; 2) doing necessary changes to run the component on special platform; 3) upgrading the specific component to get a better performance or a higher quality.

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

All components must be customized according to the operational system requirements or the interface requirements with other components in which the components should work. The component customization process overview diagram is as shown in Figure 3.5. The input to component customization is the system requirement, the component requirement, and component development document. The output should be the customized component and document for system integration and system maintenance.

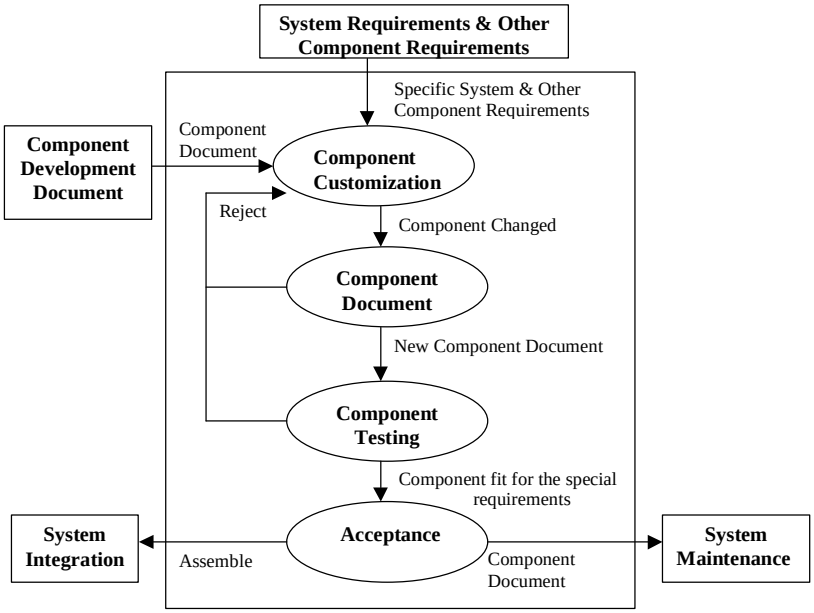


Figure 3.5 Component customization process overview

3.5 System Architecture Design

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

The objectives of system architecture design are to collect the users requirement, identify the system specification, select appropriate system architecture, and determine the implementation details such as platform, programming languages, etc.

System architecture design should address the advantage for selecting a particular architecture from other architectures. The process overview diagram is as shown in

Figure 3.6. This phase consists of system requirement gathering, analysis, system architecture design, and system specification. The output of this phase should be the system specification document for integration, and system requirement for the system testing phase and system maintenance phase.

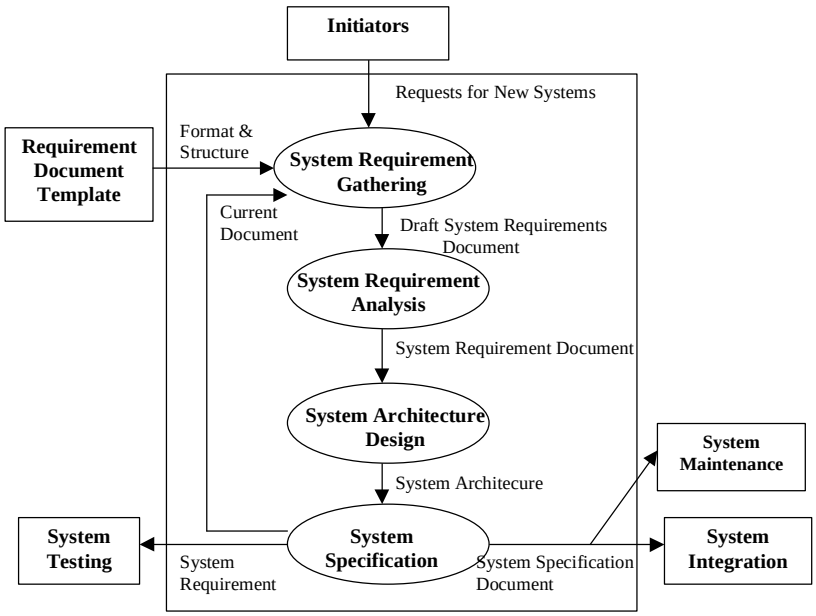


Figure 3.6 System architecture design process overview

3.6 System Integration

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

The objective of system integration is the final system composed by the selected components. The process overview diagram is as shown in Figure 3.7. The input is the system requirement documentation and the specific architecture. There are four steps

in this phase: integration, testing, changing component and re-integration (if necessary). After exiting this phase, we will get the final system ready for the system testing phase, and the document for the system maintenance phase.

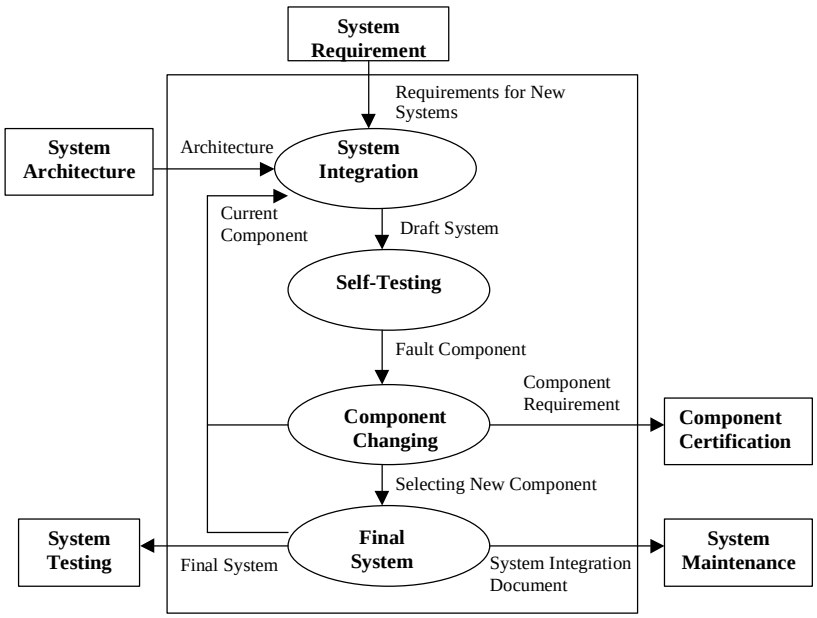


Figure 3.7 System integration process overview

3.7 System Testing

System testing is the process of evaluating a system to: 1) confirm that the system satisfies the specified requirements; 2) identify and correct defects in the system implementation.

The objective of system testing is the final system integrated by components

selected in accordance to the system requirements. System testing should contain function testing and reliability testing. The process overview diagram is as shown in Figure 3.8. This phase consists of selecting testing strategy, system testing, user acceptance testing, and completion activities. The input should be the documents from component development and system integration phases. And the output should be the testing documentation for system maintenance.

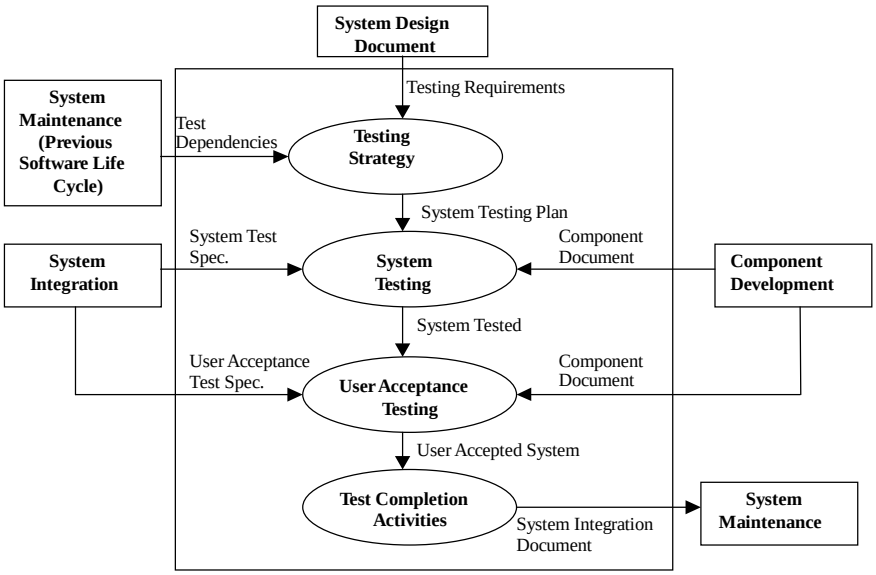


Figure 3.8 System testing process overview

3.8 System Maintenance

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

The objectives of system maintenance are to provide an effective product or service to the end-users while correcting faults, improving software performance or other attributes, and adapting the system to a changed environment.

There shall be a maintenance organization for every software product in the operational use. All changes for the delivered system should be reflected in the related documents. The process overview diagram is as shown in Figure 3.9. According to the outputs from all previous phases as well as request and problem reports from users, system maintenance should be held for determining support strategy and problem management (e.g., identification and approval). As the output of this phase, a new version can be produced for system testing phase for a new life cycle.

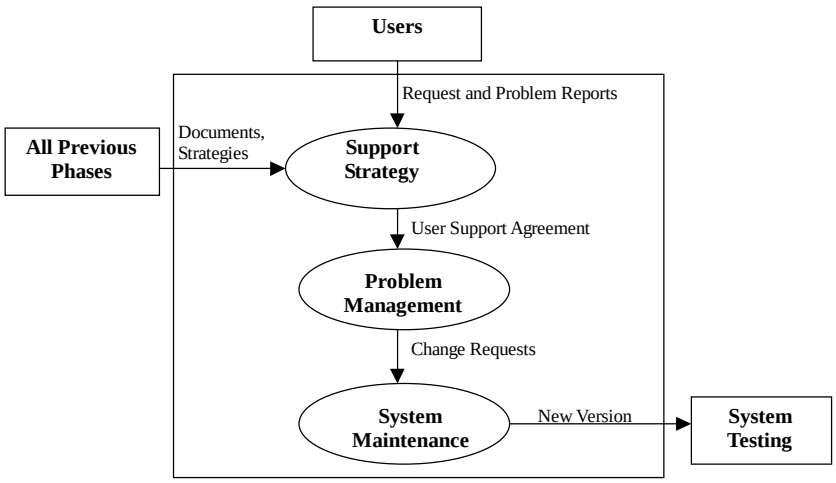


Figure 3.9 System maintenance process overview

Chapter 4
A Generic Quality Assessment Environment: ComPARE

Component-based software development has become a popular methodology in developing modern software systems. It is generally considered that this approach can reduce development cost and time-to-market, and at the same time are built to improve maintainability and reliability. As this approach is to build software systems using a combination of components including off-the-shelf components, components developed in-house and components developed contractually, the over quality of the final system greatly depends on the quality of the selected components.

We need to first measure the quality of a component before we can certify it. Software metrics are designed to measure different attributes of a software system and development process, indicating different levels of quality in the final product [24]. Many metrics such as process metrics, static code metrics and dynamic metrics can be used to predict the quality rating of software components at different development phases [24,27]. For example, code complexity metrics, reliability estimates, or metrics for the degree of code coverage achieved have been suggested. Test thoroughness metric is also introduced to predict a component's ability to hide faults during tests [25].

In order to make use of the results of software metrics, several different techniques have been developed to describe the predictive relationship between software metrics and the classification of the software components into fault-prone and non fault-prone categories [28]. These techniques include discriminant analysis [30], classification trees [31], pattern recognition [32], Bayesian network [33], case-based reasoning (CBR) [34], and regression tree models [27]. There are also some prototype or tools [36, 37] that use such techniques to automate the procedure of software quality prediction. However, these tools address only one kind of metrics, e.g., process metrics or static code metrics. Besides, they rely on only one prediction technique for the overall software quality assessment.

We propose Component-based Program Analysis and Reliability Evaluation (ComPARE) to evaluate the quality of software systems in component-based software development. ComPARE automates the collection of different metrics, the selection of different prediction models, the formulation of user-defined models, and the validation of the established models according to fault data collected in the development process. Different from other existing tools, ComPARE takes dynamic metrics into account (such as code coverage and performance metrics), integrates them with process metrics and more static code metrics for object-oriented programs (such as complexity metrics, coupling and cohesion metrics, inheritance metrics), and provides different models for integrating these metrics to an overall estimation with higher accuracy.

4.1 Objective

A number of commercial tools are available for the measurement of software metrics for object-oriented programs. Also there are off-the-shelf tools for testing or debugging software components. However, few tools can measure the static and dynamic metrics of software systems, perform various quality modeling, and validate such models against actual quality data.

ComPARE aims to provide an environment for quality prediction of software components and assess their reliability in the overall system developed using component-based software development. The overall architecture of ComPARE is shown in Figure 4.1. First of all, various metrics are computed for the candidate components. The users can then weigh the metrics and select the ones deemed important for the quality assessment exercise. After the models have been constructed

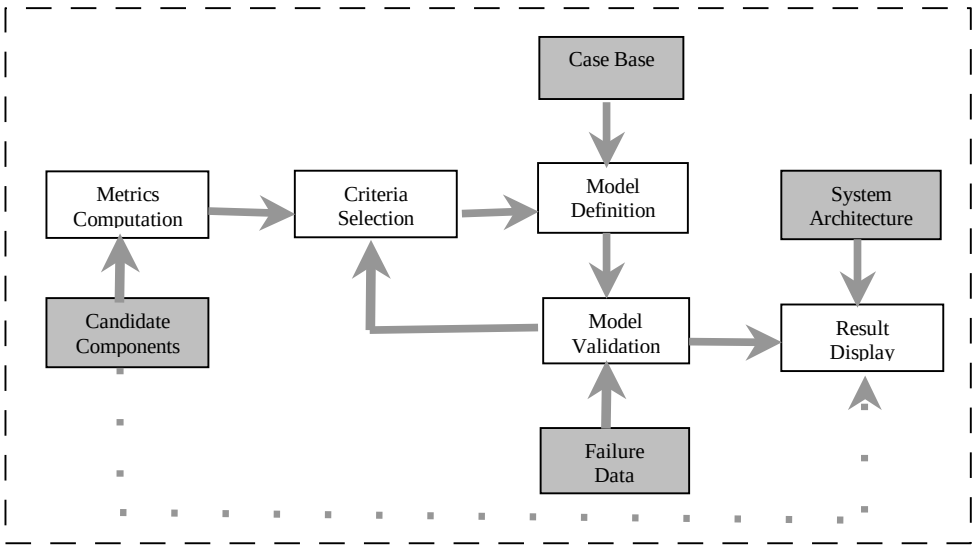


Figure 4.1 Architecture of ComPARE

and executed (e.g., Case Base with CBR), the users can validate the selected models with failure data in real life. If users are not satisfied with the prediction, they can go back to the previous step, re-define the criteria and construct a revised model. Finally, the overall quality prediction can be displayed under the architecture of the candidate system. Results for individual components can also be displayed after all the procedures.

The objectives of ComPARE can be summarized as follows:

1. *To predict overall quality system by using process metrics, static code metrics as well as dynamic metrics.* In addition to complexity metrics, we use process metrics, cohesion metrics, inheritance metrics as well as dynamic metrics (such as code coverage and call graph metrics) as the input to the quality prediction models. Thus the prediction is more accurate as it is based on data from every aspect of the candidate software components.
2. *To integrate several quality prediction models into one environment and compare the prediction result of different models.* ComPARE integrates several existing quality models into one environment. In addition to selecting or defining these different models, user can also compare the prediction results of the models on the candidate component and see how good the predictions are if the failure data of the particular component is available.
3. *To define the quality prediction models interactively.* In ComPARE, there are several quality prediction models that users can select to perform their own

predictions. Moreover, the users can also define their own model and validate their own models through the evaluation procedure.

4. *To display quality of components in different categories.* Once the metrics are computed and the models are selected, the overall quality of the component can be displayed according to the category it belongs to. Program modules with problems can also be identified.

5. *To validate reliability models defined by the user against real failure data (e.g., data obtained from change report).* Using the validation criteria, the result of the selected quality prediction model can be compared with failure data in real life. The user can redefine their models according to the comparison.

6. *To show the source code with potential problems at line-level granularity. ComPARE* can identify the source code with high risk (i.e., the code that is not covered by test cases) at line-level granularity. This can help the users to locate high risk program modules or portions promptly and conveniently.

7. *To adopt commercial tools in assessing software data related to quality attributes.* We adopt Metamata [28] and Jprobe [29] suites to measure different metrics of the candidate components. These two tools, including metrics, audits, debugging, as well as code coverage, memory and deadlock detected, are commercially available in the component-based program testing market.

4.2 Metrics Used in ComPARE

Three different categories of metrics, namely process, static, and dynamic metrics, are computed and collected in CompARE to give overall quality prediction. We have chosen the most useful metrics, which are widely adopted by previous software quality prediction tools from the software engineering research community. The process metrics we select are listed in Table 4.1 [37].

As we perceive Object-Oriented (OO) techniques are essential in the component-based software development approach, we select static code metrics according to the most important features in OO programs: complexity, coupling, inheritance and cohesion. They are listed in Table 4.2 [28,39]. The dynamic metrics encapsulate measurement of the features of components when they are executed. Table 4.3 shows the details description of the dynamic metrics.

This set of process, static, and dynamic metrics can be collected from some commercial tools, e.g., Metamata Suite [28] and Jprobe Testing Suite [29]. We measure and apply these metrics in ComPARE.

Metric	Description
Time	Time spent from the design to the delivery (months)
Effort	The total human resources used (man*month)
Change Report	Number of faults found in the development

Table 4.1 Process Metrics

Abbreviation	Description
Lines of Code (LOC)	Number of lines in the components including the statements, the blank lines of code, the lines of commentary, and the lines consisting only of syntax such as block delimiters.
Cyclomatic Complexity (CC)	A measure of the control flow complexity of a method or constructor. It counts the number of branches in the body of the method, defined by the number of WHILE statements, IF statements, FOR statements, and CASE statements.
Number of Attributes (NA)	Number of fields declared in the class or interface.
Number Of Classes (NOC)	Number of classes or interfaces that are declared. This is usually 1, but nested class declarations will increase this number.
Depth of Inheritance Tree (DIT)	Length of inheritance path between the current class and the base class.
Depth of Interface Extension Tree (DIET)	The path between the current interface and the base interface.
Data Abstraction Coupling (DAC)	Number of reference types that are used in the field declarations of the class or interface.
Fan Out (FANOUT)	Number of reference types that are used in field declarations, formal parameters, return types, throws declarations, and local variables.
Coupling between Objects (CO)	Number of reference types that are used in field declarations, formal parameters, return types, throws declarations, local variables and also types from which field and method selections are made.
Method Calls Input/Output (MCI/MCO)	Number of calls to/from a method. It helps to analyze the coupling between methods.
Lack of Cohesion Of Methods (LCOM)	For each pair of methods in the class, the set of fields each of them accesses is determined. If they have disjoint sets of field accesses then increase the count P by one. If they share at least one field access then increase Q by one. After considering each pair of methods, LCOM = (P > Q) ? (P - Q) : 0

Table 4.2 Static Code Metrics

Metric	Description
Test Case Coverage	The coverage of the source code when executing the given test cases. It may help to design effective test cases.
Call Graph metrics	The relationships between the methods, including method time (the amount of time the method spent in execution), method object count (the number of objects created during the method execution) and number of calls (how many times each method is called in you application).
Heap metrics	Number of live instances of a particular class/package, and the memory used by each live instance.

Table 4.3 Dynamic Metrics

4.2.1 Metamata Metrics

Metamata Metrics [28] evaluates the quality of software by analyzing the program source and quantifying various kinds of complexity. Complexity is a common source of problems and defects in software. High complexity makes it more difficult and costly to develop, understand, maintain, extend, test and debug a program. Some of the benefits of using metrics for complexity analysis are:

- It provides feedback into the design and implementation phases of the project to help engineers identify and remove unnecessary complexity.
- It improves the allocation of testing effort by leveraging the connection between complexity and errors, and focusing testing on the more error-prone parts of the code.
- Optimizing testing resources leads to lower testing costs, as well as a reduced

release cycle.

- Over time, metrics information collected over several projects can lead to quality control guidelines for measuring *good* software, and can thus improve the overall software development process.

Metamata has a catalog of 13 metrics which are based on standard literature from the quality assurance community and have been accepted as a necessary base of metrics by this same community. Metamata Metrics calculates global complexity and quality metrics statically from Java source code, helps organize code in a more structured manner and facilitates the QA process It has the following features:

- Most standard object oriented metrics such as object coupling and object cohesion
- Traditional software metrics such as cyclomatic complexity and lines of code
- Can be used on incomplete Java programs or programs with errors - and consequently, can be used from day one of the development cycle
- Obtain metrics at any level of granularity (methods, classes...)
- Performs statistical aggregations (mean, median...)
- Works with both JDK 1.1 and JDK 1.2

One consequence of this is that Metamata Metrics will calculate a value for a metric when given the source for a class that is different from the value that it calculates when given the corresponding class file generated for it by a Java compiler.

The current list of metrics that have equivalent definitions for both Java source and class files: Depth of inheritance tree, Number of attributes, Number of local methods, Weighted methods per class, Data abstraction coupling and Number of classes.

The current list of metrics that are either not available for class files, or can produce different values for source and class files is: Cyclomatic complexity, Lines of code, Number of remote methods, Response for class, Fan out, Coupling between objects and Lack of cohesion of methods.

4.2.2 JProbe Metrics

The JProbe from KL Group has different suites of metrics/tools for different purpose of use [29]. They are designed to help developers build robust, reliable, high-speed business applications in Java. Here is what the JProbe Developer Suite includes:

- *JProbe Profiler and Memory Debugger* - eliminates performance bottlenecks and memory leaks in your Java code
- *JProbe Threadalyzer* - detects deadlocks, stalls and race conditions
- *JProbe Coverage* - locates and measures untested Java code.

JProbe Developer Suite paints an intuitive, graphical picture of everything from memory usage to calling relationships, helping the programmer navigate to the root of the problem quickly and easily. Figure 5.2 is an example of Jprobe coverage window, stating the untested Java code including untested lines of code and methods.

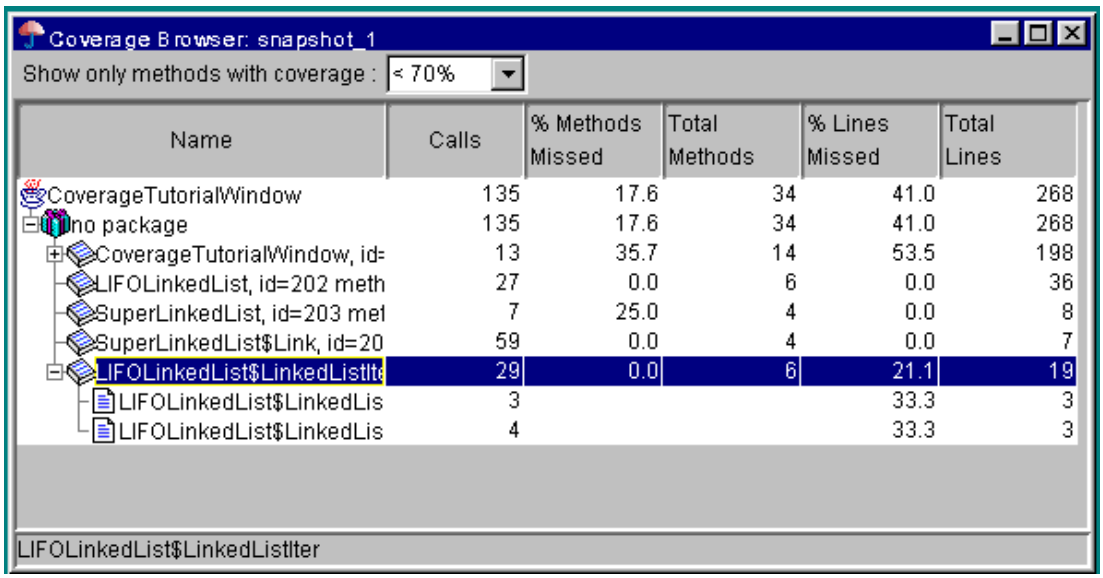


Figure 4.2 Example of a JProbe coverage browser window

4.2.3 Application of Metamata and Jprobe Metrics

Metamata metrics and Jprobe suites are both used in the QA Lab of Flashline, an industry leader in providing software component products, services and resources that facilitate rapid development of software systems for business. We use the result of such metrics applications in our risk analysis and evaluation tool: ComPARE.

Figure 4.3, Table 4.4 and 4.5 are sample reports in the QA Lab of Flashline [44] when testing the EJB components using the commercial tools mentioned above.

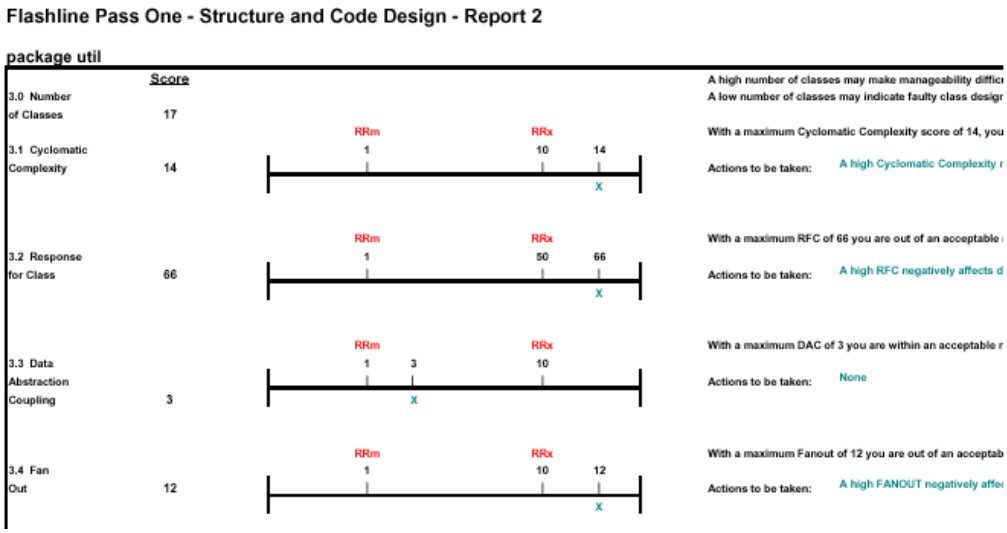


Figure 4.3 Flashline QA analysis report on structure and code design

Tests	Applicability	Actions to be taken	Value
P2.1 Performance Metrics (Method time, Object Count, Number of calls)	Identifies excessive memory usage by certain parts (methods, classes) of the application. Checks coding efficiency.	Avoid excessive object creation and excessive method calling	1. Performance 2. Reusability
P2.2 Method detail	Identifies which lines of codes are responsible for excessive memory usage or object creation	Indentify and correct the methods that are responsible for excessive memory usage	1. Performance 2. Maintainability 3. Reusability
P2.3 Method memory utilization	Maps the memory utilization of all methods. Visually portrays the methods that are using memory most heavily as having a relatively darker color than those which are more lean.	Audit those methods identified as using excessive memory for correct logic and structure	1. Performance 2. Maintainability
P2.4 Heap Usage	Dynamically portrays, through a series of "snapshots" the amount of memory available to the JVM. This identifies at what point in the program execution cycle there is a memory leak.	Audit those classes and methods that are creating the memory leaks.	1. Performance 2. Maintainability 3. Reliability
P2.5 Identify untested and unused lines of code	Scans code for those lines that have not been tested due to unfulfilled testing conditions and for code that is packaged in classes that are rarely called.	Design testing methodologies that exercise 100% of the code.	1. Reliability 2. Maintainability
P2.6 Thread interaction monitor: deadlock prediction and avoidance	If the program is taking too much time and memory for no apparent reason, thread conflict might be the root cause. This test looks for possible thread interaction sequences that present a danger of deadlock, racing situation, or starvation.	Walk through the logic carefully looking out for potential thread conflict.	1. Performance 2. Maintainability

Table 4.4 Flashline QA report on dynamic metrics

Features	Applicabiliy	Actions to be taken	Value
P1.1 Depth of inheritance hierarchy	When code hierarchy is too deep, it's difficult to understand, predict behavior and (potentially) debug	Determine, if it's possible to reduce the depth of inheritance hierarchy	1. Maintainability 2. Reusability
P1.2. Data abstraction coupling	Counts the number of types that are used in the field declarations. Too many reference types make reuse/coupling/decoupling more difficult	Determine the necessity of coupling	1. Reusability 2. Maintainability
P1.3. Number of attributes	A high number of attributes may lead to inefficient memory utilization and may reflect poor product design. A low number of attributes per class can also indicate poor design, for example, unnecessary levels of inheritance	Perform attribute usage walkthrough to determine necessity of attributes	1. Maintainability 2. Reusability
P1.4. Number of methods (simple, by categories, weighted)	A high number of methods per class indicate that the class design has been partitioned incorrectly. A low number of attributes per class can also indicate poor design, for example, unnecessary levels of inheritance	Perform attribute usage walkthrough to determine necessity of methods. Check the class cohesion (M12)	1. Maintainability 2. Reusability
P1.5. Number of classes	A system with high number of classes has potentially more interactions between objects. This reduces comprehensibility of the system that in turn makes it harder to test, debug and maintain. A low number of classes may indicate that that the class design has been partitioned incorrectly	If number of classes is too high, check for high P1.1. If number of classes is too low, check for high P1.12, P1.2, and P1.11.	1. Maintainability
P1.6. Cyclomatic complexity	Methods with a high cyclomatic complexity tend to be more difficult to understand and maintain	If cyclomatic complexity is too high, try to split complex methods into several simpler ones.	1. Maintainability
P1.7. Lines of code	A high number of lines of code per class or per method can reduce maintainability	If a method has a high number of lines of code, check for high P1.6 and act accordingly. If a class has a high number of lines of code, check for high P1.12.	1. Maintainability
P1.8. Number of remote methods	Counts the number of invocations of methods that doesn't belong to class, its superclass, its subclasses or interfaces the class implements. High number of remote methods can be an indication of high coupling between classes.	If the number of remote methods is high, check for high P1.2, P1.10, and P1.12.	1. Maintainability 2. Reusability
P1.9. Response for class	Counts the sum of number methods, defined in the class and number of remote methods	If the number is high, check separately for high P1.4 and P1.8	1. Maintainability 2. Reusability
P1.10. Fan out	Counts the number of reference types used in: • field declarations • formal parameters and return types • *throws* declarations • local variables	If the number is high, check for high P1.2 and P1.11	1. Maintainability 2. Reusability
P1.11. Coupling between objects.	A high coupling reduces modularity of the class and makes reuse more difficult.	If coupling is high, check for high P1.2, P1.5 and P1.12.	1. Maintainability 2. Reusability
P1.12. Lack of class cohesion	The cohesion of a class is the degree to which its methods are related to each other. If a class exhibits low method cohesion, it indicates that the design of the class has probably been partitioned incorrectly, and could benefit by being split into more classes with individually higher cohesion	Split class if necessary	1. Reusability 2. Maintenability

Table 4.5 Flashline QA report on code metrics

4.3 Models Definition

In order to predict the quality of different software components, several techniques have been developed to classify software components according to their reliability [27]. These techniques include discriminant analysis [30], classification trees [31], pattern recognition [32], Bayesian network [33], case-based reasoning (CBR) [34] and regression tree model [37]. In ComPARE, we integrate five types of models to evaluate the quality of the software components for an overall component-based system evaluation. Users can customize these models and compare the prediction results from different tailor-made models.

4.3.1 Summation Model

This model gives a prediction by simply adding all the metrics selected and weighted by a user. The user can validate the result by real failure data, and then benchmark the result. Later when new components are included, the user can predict their quality according to their differences from the benchmarks. The concept of summation model can be summarized as the following:

$$Q = \sum_{i=1}^{n} a_i m_i \qquad (1)$$

where m_i is the value of one particular metric, a_i is its corresponding weighting factor, n is the number of metrics, and Q is the overall quality mark.

4.3.2 Product Model

Similar to the summation model, the product model multiplies all the metrics selected and weighted by the user and the resulting value indicates the level of quality of a given component. Similarly, the user can validate the result by real failure data, and then determine the benchmark for later usage. The concept of product model is shown as the following:

$$Q = \prod_{i=1}^{n} mi \qquad (2)$$

where m_i is the value of one particular metric, n is the number of metrics, and Q is the overall quality mark. Note that m_is are normalized as a value which is close to 1, so that none of them will dominate the result.

4.3.3 Classification Tree Model

Classification tree model [31] is to classify the candidate components into different quality categories by constructing a tree structure. All the candidate components are leaves in the tree. Each node of the tree represents a metric (or a composed metric calculated by other metrics) of a certain value. All the children of the left sub tree of the node represent those components whose value of the same metric is smaller than the value of the node, while all the children of the right sub tree of the node are those components whose value of the same metric is equal to or larger than the value of the node.

The tree modeling approach [27] is a goal oriented statistical technique which consists of recursive partitioning of the variable space using binary splits. The dependent variable or the response variable (usually denoted by y) in this context consists of the number of faults in a software module and the set of classication, predictor or independent variables (usually denoted by x) consists of the various software complexity metrics for the module. The algorithm attempts to partition the predictor variable space into homogeneous regions such that within each region the distribution of the response variable conditional to the predictor variables f(yjx), is independent of the predictor variables (x).

At each step, the tree-construction algorithm searches through all possible binary splits of all the predictor variables until the overall deviance, i.e., the sum of the deviances for each subset is minimized. The algorithm then begins the search again for the next binary split, reconsidering all the variables until the next binary split is made, and so on. Thus the tree-construction method uses a one-step look ahead, i.e., it chooses the next split by minimizing the deviance for that split, without making an effort to optimize the performance of the entire tree which is an NP-complete problem.

Intuitively, the algorithm uses a set of learning data to construct a regression tree which is used as a predicting device. Each terminal node in the tree represents a partition or a subset of the data that is homogeneous with respect to the dependent variable. The predicted value of the dependent variable is the average of all the observations in the node. In the present context, the tree-modeling procedure attempts to identify the modules with the same number of errors, and thus have the same degree

of fault-proneness.

In ComPARE, a user can define the metrics and their value at each node from the root to the leaves. Once the tree is constructed, a candidate component can be directly classified by following the threshold of each node in the tree until it reaches a leaf node. The user can validate and evaluate the final tree model after its definition. Below is an example of the outcome of a tree model. At each node of the tree there are metrics and values, and the leaves represent the components with certain number of predicted faults in the classification result.

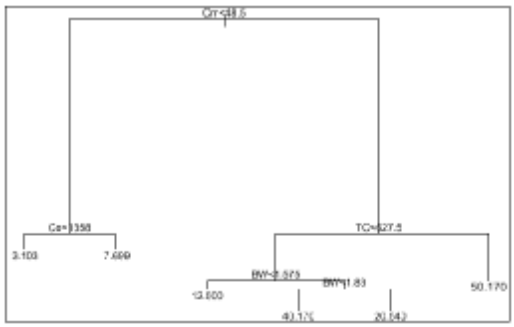


Figure 4.4 An example of classification tree model

4.3.4 Case-Based Reasoning Model

Case-based reasoning (CBR) has been proposed for predicting the quality of software components [34]. A CBR classifier uses previous "similar" cases as the basis for predicting the quality of a software component.. Previous cases are stored in a case base. Similarity is defined in terms of a set of metrics. The major conjecture behind this model is that the candidate component that has a similar structure to the components in the case base will inherit a similar quality level.

CBR method has a number of advantages. Most notable one is that the detailed characterization of the similar cases can help to interpret the automatic classification results. In principle, as well, a CBR classifier would provide a straight forward approach for dealing with missing values. However, in the context of quality prediction using product metrics, there are rarely missing values.

When evaluating the predictive performance of a CBR classifier, one first constructs a case base of previous components where the source code/dynamic metrics and the quality are known. A different data set is then used as the test set. This can be achieved in a number of different ways, such as a holdout sample, cross-validation, bootstrapping, multiple releases, or random subsets.

A CBR classifier can be instantiated in different ways by varying its parameters. But according to the previous research, there is no significant difference in prediction validity when using any combination of parameters in CBR. So we adopt the simplest CBR classifier modeling with Euclidean distance, z-score standardization [34], but no weighting scheme. Finally, we select the single, nearest neighbor for the prediction purpose.

4.3.5 Bayesian Network Model

Bayesian networks (also known as Bayesian Belief Networks, BBN) is a graphical network that represents probabilistic relationships among variables [33]. BBNs enable reasoning under uncertainty. The framework of Bayesian networks offers a compact,

intuitive, and efficient graphical representation of dependence relations between entities of a problem domain. The graphical structure reflects properties of the problem domain directly, which provides a tangible visual representation as well as a sound mathematical basis in Bayesian probability [35]. The foundation of Bayesian networks is the following theorem known as Bayes' Theorem:

$$P(H|E,c) = \frac{P(H|c)P(E|H,c)}{P(E|c)} \qquad (3)$$

where H, E, c are independent events and P the probability of such event under certain circumstances.

With BBNs, it is possible to integrate expert beliefs about the dependencies between different variables and to propagate consistently the impact of evidence on the probabilities of uncertain outcomes, such as "unknown component quality". Details of the BBN model for quality prediction can be found in [33]. Users can also define their own BBN models in ComPARE and compare the results with other models.

4.4 Operations in ComPARE

As a generic quality assessment environment for component-based software system, ComPARE suggests eight major functional areas: File Operations, Selecting Metrics, Selecting Criteria, Model Selection and Definition, Model Validation,

Display Result, Windows Switch, and Help System. The details of some key functions are described in the following sections.

4.4.1 Selecting Metrics

User can select the metrics they want to collect for the opened component-based system. There are three categories of metrics available: process metrics, static metrics and dynamic metrics. The details of these metrics have shown in section 4.2.

4.4.2 Selecting and Weighing Criteria

After computing the different metrics, users need to select and weigh the criteria on these metrics before using them in the reliability modeling. Each metric can be selected or omitted, and if selected, be marked with the weight between 0 and 100%. Such information will be used as input parameter later in the quality prediction models.

4.4.3 Model Selection and Definition

The Model operations allow users to select or define the model they would like to perform in the evaluation. The users should give the probability of each item related to the overall quality of the candidate component.

4.4.4 Model Validation

Model validation allows comparison between different models and with respect to actual software failure data. It facilitates the users to compare the different results based on chosen subset of the software failure data under certain validation criteria. The comparisons between different models in their predictive capability are summarized in a summary table. Model Validation operations are activated only when the software failure data are available.

4.5 ComPARE Prototype

Under the framework that we have described, we prototyped a specific version of ComPARE which targets software components developed by the Java language. Java is one of the most popular languages used in off-the-shelf components development today, and can run three standard architectures for component-based software development: i.e., CORBA, COM/DCOM and JavaBeans/EJB.

Figure 4.4 and Figure 4.5 show screen dumps of the described ComPARE prototype tool. Using ComPARE, computation of various metrics for software components and application of quality prediction models are straightforward. Users also have flexible choices in selecting and defining different models. The combination of simple operations and a variety of quality models makes it easy for users to identify an appropriate prediction model for a given component-based system.

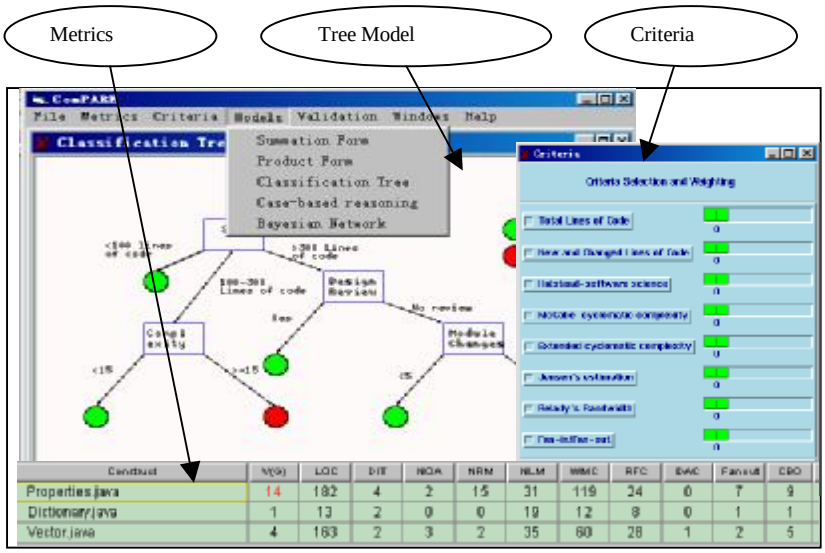


Figure 4.4 GUI of ComPARE for metrics, criteria and tree model

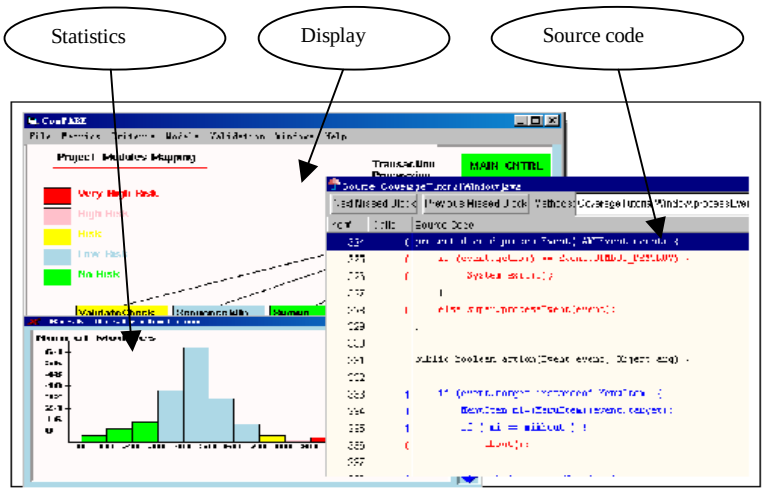


Figure 4.5 GUI of ComPARE for prediction display, risky source code and result statistics

Chapter 5

Experiments and Discussions

In ComPARE, we propose to provide a systematic procedure for predicting the quality of software components and assess their reliability in the overall system developed using component-based software development. ComPARE integrates several quality prediction models into one environment and compare the prediction result of different models in case that the failure data of the particular component is available.

Also, ComPARE adopts commercial tools in accessing software data related to quality attributes. We adopt Metamata and Jprobe suites (see section 4.2) to measure the different metrics for the candidate components, as well as CART and Hugin sytems (see section 5.3) as the classification tree and Bayesian Network model to predict the quality of the given components. These tools are widely adopted in the component-based program certification and quality prediction markets.

In this chapter, we use these classification tree and BBN models to predict and evaluate the relationship between the number of faults and software metrics of some CORBA programs. All the programs are designed according to the same specification,

the programming teams can choose their own programming languages. The test cases are designed to access the functionalities of the final programs according to the specification. The details of the test can be found in [45]. Here we apply the programs to our prediction models.

5.1 Data Description

In the fall of 1998 we engaged 19 programming teams to design, implement, test and demonstrate a Soccer Team Management System using CORBA, which is a project of a class for the students majored in computer science. The duration of the project was 4 weeks. The programming teams (2-3 students for each team) participating in this project were required to independently design and develop a distributed system, which allows multiple clients to access a Soccer Team Management Server for 10 different operations. The teams were free to choose different CORBA vendors (Visibroker or Iona Orbix), using different programming languages (Java or C++) for the client or server programs. These programs have to pass an acceptance test, in which programs were subjected to two types of test cases for each of the 10 operations: one for normal operation and the other for operations which would raise exceptions. The total number of test cases used in this experiment are 57.

Among these 19 programs 12 chose to use Visibroker, while 7 chose to use Iona Orbix. For the 12 Visibroker programs, 9 used Java for both client and server

implementation, 2 used C++ for both client and server implementation, and 1 used Java as its client and C++ as its server.

The metrics collected and the test results for the different program versions are shown in Table 5.1.

Team	TLOC	CLOC	SLOC	CClass	CMethod	SClass	SMethod	Fail	Maybe	R	R1
P2	1129	613	516	3	15	5	26	7	6	0.77	0.88
P3	1874	1023	851	3	23	5	62	3	6	0.84	095
P4	1309	409	900	3	12	1	23	3	12	0.74	0.95
P5	2843	1344	1499	4	26	1	25	2	1	0.95	0.96
P6	1315	420	895	3	3	1	39	13	10	0.60	0.77
P7	2674	1827	847	3	17	5	35	3	14	0.70	0.95
P8	1520	734	786	3	24	4	30	1	6	0.88	0.98
P9	2121	1181	940	4	22	3	43	4	2	0.89	0.93
P10	1352	498	854	3	12	5	41	2	2	0.93	0.96
P11	563	190	373	3	12	3	20	6	3	0.84	0.89
P12	5695	4641	1054	14	166	5	32	1	4	0.91	0.98
P13	2602	1587	1015	3	27	3	32	17	19	0.37	0.70
P14	1994	873	1121	4	12	5	39	4	6	0.82	0.93
P15	714	348	366	4	11	4	33	2	5	0.88	0.96
P16	1676	925	751	3	3	23	44	30	0	0.47	0.47
P17	1288	933	355	6	25	5	35	3	3	0.89	0.95
P18	1731	814	917	3	12	3	20	4	9	0.77	0.93
P19	1900	930	970	3	3	2	20	35	1	0.37	0.39

Table 5.1 General Metrics of Different Teams

The meaning of the metrics and testing results are listed below:

- Total Lines of Code (TLOC): the total length of whole program, including lines of codes in client program and server program;
- Client LOC (CLOC): lines of codes in client program;
- Server LOC (SLOC): lines of codes in server program;

- Client Class (CClass): number of classes in client program;
- Client Method (CMethod): number of methods in client program;
- Server Class (SClass): number of classes in server program;
- Server Method (SMethod): number of methods in server program;
- Fail: the number of test cases that the program failed on
- Maybe: the number of test cases, which were designed to raise exceptions, and failed to work as the client side of the program forbid it. In this situation, we were not sure whether the server was designed properly to raise the expected exceptions. Thus we put down "maybe" as the result.
- R: pass rate, defined by $R_j = \frac{P_j}{C}$, where C is the total number of test cases applied to the programs (i.e., 57); P_j is the number of "Pass" cases for program j, $P_j = C - Fail - Maybe$.
- R1: pass rate 2, defined by $R1_j = \frac{P_j + M_j}{C}$, where C is the total number of test cases applied to the programs (i.e., 57); P_j is the number of "Pass" cases for program j, $P_j = C - Fail - Maybe$; M_j is the number of "Maybe" cases for program j.

5.2 Experiment Procedures

In order to evaluate the quality of these CORBA programs, we applied the test cases to the programs and assessed their quality and reliability based on the test results. We describe our experiment procedures below.

First of all, we collected the different metrics of all the programs. Metamata and JProbe Suite were used for this purpose.

We designed test cases for these CORBA programs according to the specification. We used black box testing method, i.e., testing was on system functions only. Each operation defined in the system specification was tested one by one. We defined some test cases for each operation. The test cases were selected in 2 categories: normal cases and cases that caused exceptions in the system. For each operation in the system, at least 1 normal test case was conducted in the testing. In the other cases, all the exceptions were covered. But in order to reduce the work load, we tried to use as few test cases as possible so long as all the exceptions have been catered for.

We used the test results as indicator of quality. We applied different quality prediction models: i.e., classification tree model and Bayesian Network model to the metrics and test results. We then validate the prediction results of these models against the test results.

We divided the programs into two groups: training data and test set, and adopted cross evaluation. This was done after or during the prediction process according to the prediction models.

After applying the metrics to the different models, we analyzed the accuracy of

their predicting results and identified their advantages and disadvantages. Also, based on the results, we adjusted the coefficients and weights of different metrics in the final models.

5.3 Modeling Methodology

We adopted two quality prediction models in our experiment: classification tree model and Bayesian Belief Network. Respectively, two commercial tools CART and Hugin Explorer tool were used.

5.3.1 Classification Tree Modeling

CART is an acronym for Classification and Regression Trees, a decision-tree procedure introduced in 1984. Salford Systems' CART [41] is the only decision tree system based on the original CART code and included enhancements. The CART methodology solves a number of performance, accuracy, and operational problems that still plague many current decision-tree methods. CART's innovations include:

- solving the "how big to grow the tree" problem;
- using strictly two-way (binary) splitting;
- incorporating automatic testing and tree validation, and;
- providing a completely new method for handling missing values.

The CART methodology is technically known as binary recursive partitioning. The process is binary because parent nodes are always split into exactly two child nodes and recursive because the process can be repeated by treating each child node as a parent. The key elements of a CART analysis are a set of rules for:

- splitting each node in a tree;
- deciding when a tree is complete; and
- assigning each terminal node to a class outcome (or predicted value for regression)

Splitting Rules

To split a node into two child nodes, CART always asks questions that have a "yes" or "no" answer. For example, the questions might be: is age <= 55? Or is credit score <= 600?

How do we come up with candidate splitting rules? CART's method is to look at all possible splits for all variables included in the analysis. For example, consider a data set with 215 cases and 19 variables. CART considers up to 215 times 19 splits for a total of 4085 possible splits. Any problem will have a finite number of candidate splits and CART will conduct a brute force search through them all.

Choosing a Split

CART's next activity is to rank order each splitting rule on the basis of a quality-of-split criterion. The default criterion used in CART is the GINI rule, essentially a measure of how well the splitting rule separates the classes contained in the parent node.

Besides Gini, CART includes six other single-variable splitting criteria - Symgini, twoing, ordered twoing and class probability for classification trees, and least squares and least absolute deviation for regression trees - and one multi-variable splitting criteria, the linear combinations method. The default Gini method typically performs best, but, given specific circumstances, other methods can generate more accurate models. CART's unique "twoing" procedure, for example, is tuned for classification problems with many classes, such as modeling which of 170 products would be chosen by a given consumer.

Stopping Rules and Class Assignment

Once a best split is found, CART repeats the search process for each child node, continuing recursively until further splitting is impossible or stopped. Splitting is impossible if only one case remains in a particular node or if all the cases in that node are exact copies of each other (on predictor variables). CART also allows splitting to be stopped for several other reasons, including that a node has too few cases. (The default for this lower limit is 10 cases, but may be set higher or lower to suit a

particular analysis).

Once a terminal node is found we must decide how to classify all cases falling within it. One simple criterion is the plurality rule: the group with the greatest representation determines the class assignment. CART goes a step further: because each node has the potential for being a terminal node, a class assignment is made for every node whether it is terminal or not. The rules of class assignment can be modified from simple plurality to account for the costs of making a mistake in classification and to adjust for over- or under-sampling from certain classes.

Pruning Trees

Instead of attempting to decide whether a given node is terminal or not, CART proceeds by growing trees until it is not possible to grow them any further. Once CART has generated what we call a maximal tree, it examines smaller trees obtained by pruning away branches of the maximal tree. The reason CART does not stop in the middle of the tree-growing process is that there might still be important information to be discovered by drilling down several more levels.

Testing

Once the maximal tree is grown and a set of sub-trees are derived from it, CART determines the best tree by testing for error rates or costs. With sufficient data, the simplest method is to divide the sample into learning and test sub-samples. The

learning sample is used to grow an overly-large tree. The test sample is then used to estimate the rate at which cases are misclassified (possibly adjusted by misclassification costs). The misclassification error rate is calculated for the largest tree and also for every sub-tree. The best sub-tree is the one with the lowest or near-lowest cost, which may be a relatively small tree.

Some studies will not have sufficient data to allow a good-sized separate test sample. The tree-growing methodology is data intensive, requiring many more cases than classical regression. When data are in short supply, CART employs the computer-intensive technique of cross validation.

Cross Validation

CART uses two test procedures to select the "optimal" tree, which is the tree with the lowest overall misclassification cost, thus the highest accuracy. Both test disciplines, one for small datasets and one for large, are entirely automated, and they ensure the optimal tree model will accurately classify existing data and predict results. For smaller datasets and cases when an analyst does not wish to set aside a portion of the data for test purposes, CART automatically employs cross-validation. For large datasets, CART automatically selects test data or uses pre-defined test records or test files to self-validate results.

Cross validation is used if data are insufficient for a separate test sample. In such cases, CART grows a maximal tree on the entire learning sample. This is the tree that

will be pruned back. CART then proceeds by dividing the learning sample into 10 roughly-equal parts, each containing a similar distribution for the dependent variable. CART takes the first 9 parts of the data, constructs the largest possible tree, and uses the remaining 1/10 of the data to obtain initial estimates of the error rate of selected sub-trees. The same process is then repeated (growing the largest possible tree) on another 9/10 of the data while using a different 1/10 part as the test sample. The process continues until each part of the data has been held in reserve one time as a test sample. The results of the 10 mini-test samples are then combined to form error rates for trees of each possible size; these error rates are applied to the tree based on the entire learning sample.

5.3.2 Bayesian Belief Network Modeling

The HUGIN System is a tool enabling one to construct model based decision support systems in domains characterized by inherent uncertainty. The models supported are *Bayesian belief networks* and their extension *influence diagrams*. The HUGIN System allows the user to define both discrete nodes and to some extent continuous nodes in the models.

Bayesian networks are often used to model domains that are characterized by inherent uncertainty. This uncertainty can be due to imperfect understanding of the domain, incomplete knowledge of the state of the domain at the time where a given task is to be performed, randomness in the mechanisms governing the behaviour of the

domain, or a combination of these.)

Formally, a Bayesian belief network can be defined as follows: A Bayesian belief network is a directed acyclic graph with the following properties:

- Each node represents a random variable.
- Each node representing a variable A with parent nodes representing variables B_1, B_2,..., B_n is assigned a conditional probability table (cpt):

$$P(A|B_1, B_2, \ldots, B_n)$$

The nodes represent random variables, and the edges represent probabilistic dependencies between variables. These dependences are quantified through a set of conditional probability tables (CPTs): Each variable is assigned a CPT of the variable given its parents. For variables without parents, this is an unconditional (also called a marginal) distribution.

Inference in a Bayesian network means computing the conditional probability for some variables given information (evidence) on other variables. This is easy when all available evidence is on variables that are ancestors of the variable(s) of interest. But when evidence is available on a descendant of the variable(s) of interest, we have to perform inference against the direction of the edges. To this end, we employ Bayes' Theorem:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

An influence diagram is a belief network augmented with decisions and utilities

(the random variables of an influence diagram are often called chance variables). Edges into decision nodes indicate time precedence: an edge from a random variable to a decision variable indicates that the value of the random variable is known when the decision will be taken, and an edge from one decision variable to another indicates the chronological ordering of the corresponding decisions. The network must be acyclic, and there must exist a directed path that contains all decision nodes in the network.

We have developed a prototype BBN to show the potential of one of the quality prediction models: BBN, and illustrated their useful properties using real metrics data from the project. The quality prediction BBN example is shown in Figure 5.1. The node probability is determined by the metrics and the testing data, see Table 5.1. Figure 5.1 also shows the execution of the BBN model using the Hugin Explorer tool [35].

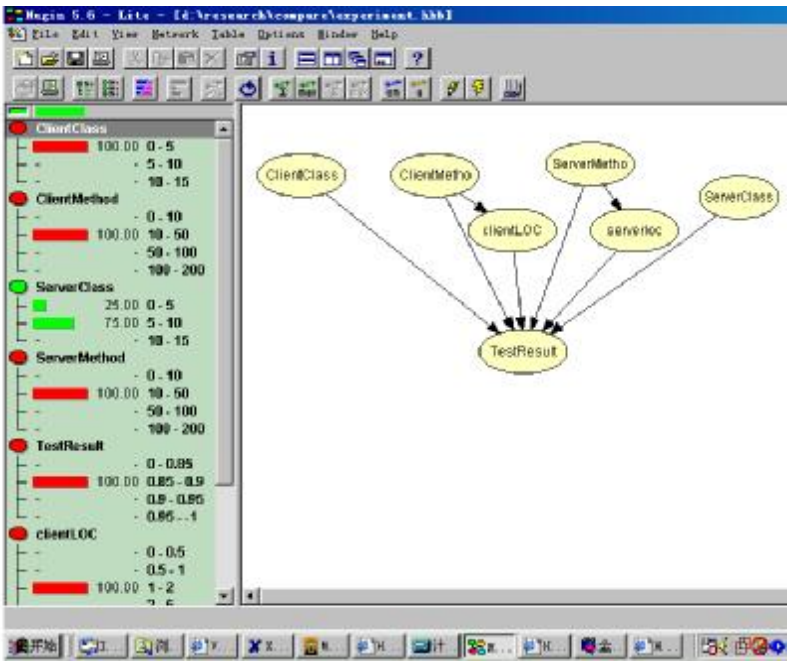


Figure 5.1 The quality prediction BBN model and execution demonstration.

5.4 Experiment Results

5.3.1 Classification Tree Results Using CART

We apply the metrics and testing results in Table 5.1 to the CART tool, and get the classification tree results of predicting the quality variable "Fail". Table 5.2 is the option setting when we construct the tree modeling. The tree constructed is shown as Figure 5.2, and the relative importance of each metric is listed in Table 5.3.

The detailed information and the report of running CART can be found in Appendix A.

```
Construction Rule           Least Absolute Deviation
Estimation Method           Exploratory - Resubstitution
Tree Selection              0.000 se rule
Linear Combinations         No

Initial value of the complexity parameter             = 0.000
Minimum size below which node will not be split       = 2
Node size above which sub-sampling will be used       = 18
Maximum number of surrogates used for missing values  = 1
Number of surrogate splits printed                    = 1
Number of competing splits printed                    = 5
Maximum number of trees printed in the tree sequence  = 10
Max. number of cases allowed in the learning sample   = 18
Maximum number of cases allowed in the test sample    = 0
Max # of nonterminal nodes in the largest tree grown  = 38
 (Actual # of nonterminal nodes in largest tree grown = 10)
Max. no. of categorical splits including surrogates   = 1
Max. number of linear combination splits in a tree    = 0
     (Actual number cat. + linear combination splits  = 0)
Maximum depth of largest tree grown                   = 13
     (Actual depth of largest tree grown              = 7)
Maximum size of memory available                     = 9000000
     (Actual size of memory used in run               = 5356)
```

Table 5.2 Option Setting when constructing the classification tree

```
                 Relative     Number Of   Minimum
Metrics         Importance    Categories  Category
-------------------------------------------------
CMETHOD           100.000
TLOC               45.161
SCLASS             43.548
CLOC               33.871
SLOC                4.839
SMETHOD             0.000
CCLASS              0.000

N of the learning sample =       18
```

Table 5.3 Variable importance in classification tree

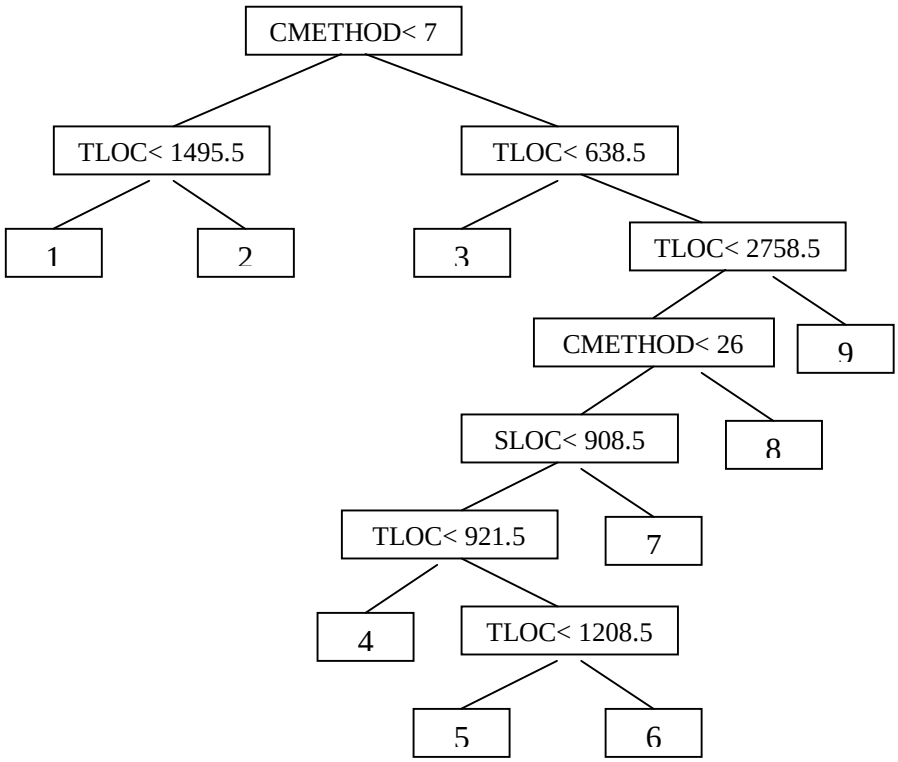


Figure 5.2 Classification tree structure

Parent Node	Wgt Count	Count	Median	MeanAbsDev	Complexity
1	1.00	1	13.000	0.000	17.000
2	2.00	2	35.000	2.500	17.000
3	1.00	1	6.000	0.000	6.333
4	1.00	1	2.000	0.000	2.500
5	1.00	1	7.000	0.000	4.000
6	6.00	6	3.000	0.500	4.000
7	3.00	3	4.000	0.000	3.000
8	1.00	1	17.000	0.000	14.000
9	2.00	2	2.000	0.500	8.000

Table 5.4 Terminal node information in classification tree

From Figure 5.2, we can see that the 18 learning samples are classified into 9 groups (terminal nodes), whose information are listed in Table 5.4. The most important vector was the number of methods in the client program (CMethod), and the next three

most important vectors were TLOC, SCLASS and CLOC. From the node information, we can observe that the most non fault-prone nodes are those programs with 638.5<TLOC<921.5 and 7<CMETHOD<26 and SLOC<908.5, or CEMTHOD>7 and TLOC<638.5. The relationship between classification results and three main metrics was analyzed and listed in Table 5.5.

Terminal Node	Mean Faults	CMethod	TLOC	SLOC
4	2	7~26	638.5~921.5	<=908.5
9	2	>7	<=638.5	-
6	3	7~26	1208.5~2758.5	<=908.5
7	4	7~26	638.5~921.5	>908.5
3	6	>7	<=638.5	-
5	7	7~26	638.5~921.5	<=908.5
1	13	<=7	<=1495.5	-
8	17	>26	638.5~921.5	-
2	35	<=7	>1495.5	-

Table 5.5 Relationship between classification results and 3 main metrics

5.3.2 BBN Results Using Hugin

We constructed an influence diagram for the CORBA programs according to the metrics and testing results collected in the testing procedure, as shown in Figure 5.3. The “TestResult” here is the variable “Fail” in Table 5.1. The reason why we chose the simplest diagram here to let each metric influences the testing result directly is that, we assume that each of these metrics has its own impacts on the testing result, even if there are some redundancy or interaction between these metrics. We would not omit any important relationships using such diagram, and it should be a good starting point

for our analysis.

Once the influence diagram is constructed, we input the probability of metrics and testing results collected in our test procedures, as shown in Figure 5.4.

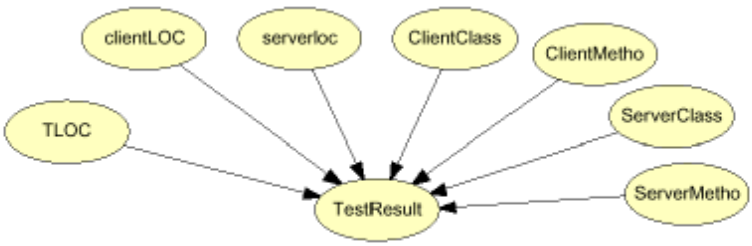


Figure 5.3 The Influence Diagram of the BBN model

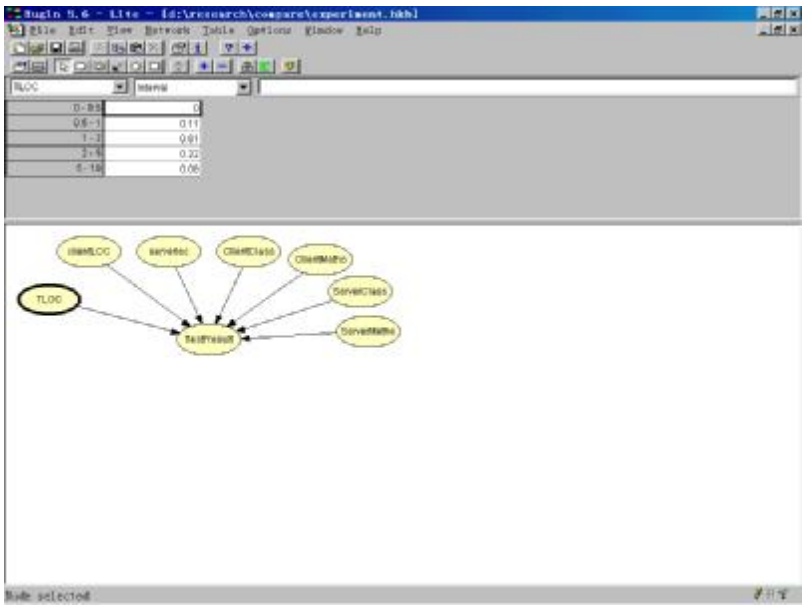
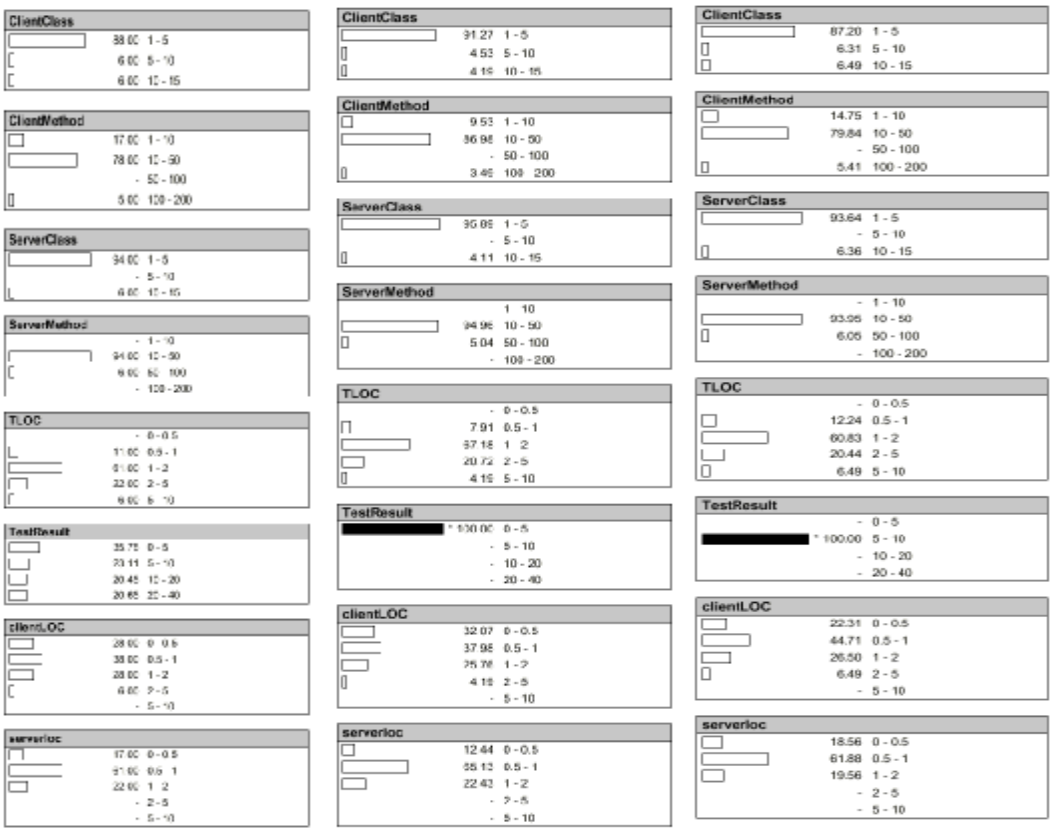


Figure 5.4 The probability description of nodes in BBN model

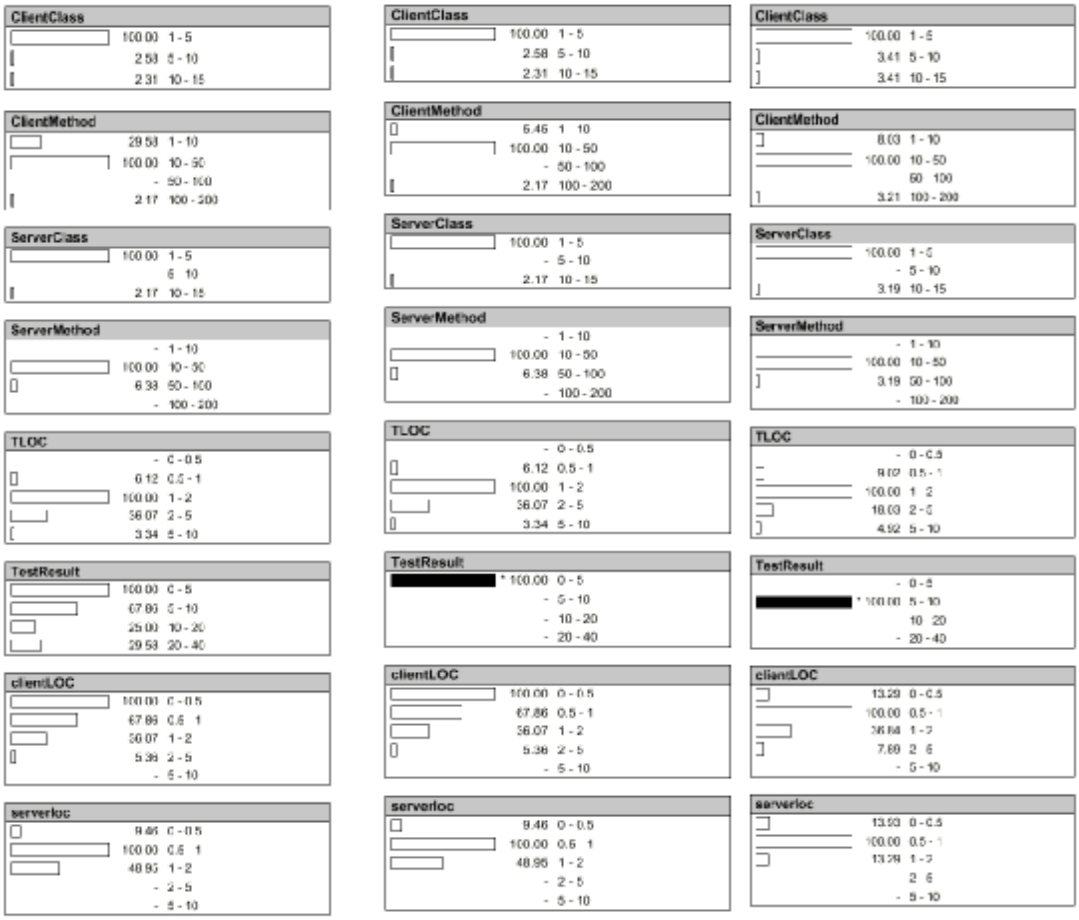


(a) (b) (c)

Figure 5.5 The different probability distribution of metrics according to the quality indicator (sum propagation)

The running result of Hugin tool are shown in Figure 5.5 and Figure 5.6, where (a) is the original probability distribution of different metrics and testing results; (b) is the probability distribution of the metrics when the number of faults is less than 5; (c) is the probability distribution of the metrics when the number of faults is between 5 and 10. Figure 5.5 is the results of summation propagation, and Figure 5.6 is the results of max propagation.

The sum propagation shows the true probability of state of nodes with the total summation 1. For the max propagation, if a state of a node belongs to the most



(a) (b) (c)

Figure 5.6 The different probability distribution of metrics according to the quality indicator (max propagation)

probable configuration it is given the value 100. All other states are given the relative value of the probability of the most probable configuration they found in comparison to the most probable configuration. That is, assume a node N has two states a and b, and b belongs to the most probable configuration of the entire BBN which has the probability 0.002. Then, b is given the value 100. Now, assume that the most probable configuration which a belongs to has probability 0.0012. Then, a is given the value 60.

Using max propagation instead of sum propagation, we can find the probability of

the most likely combination of states under the assumption that the entered evidence holds. In each node, a state having the value 100.00 belongs to a most likely combination of states.

From the Figure 5.6(b), we can find the best combination of the metrics with respect to the corresponding testing results, as listed in Table 5.6. For test result between 0 and 5, the ranges of CMethod, TLOC and SLOC are very close to the results of classification tree in Table 5.5.

TestResult	CCLASS	CMethod	SCLASS	SMethod	TLOC	CLOC	SLOC
0-5	1-5	10-50	1-5	10-50	1-2K	0-0.5K	0.5-1K
5-10	1-5	10-50	1-5	10-50	1-2L	0.5-1K	0.5-1K

Table 5.6 Relationship between test result and metrics in BBN

5.5 Comparison and Discussion

In our experiment, we used some real CORBA programs as the testing data and applied them to two quality prediction models: classification tree model and Bayesian Belief Network model. We adopted two commercial tools: CART and Hugin systems to implement the two models accordingly. From the experimental results listed above, we compared the quality prediction ability of the two models.

First, classification tree model predicts the quality of a program by constructing a tree model according to the collected metrics. If the learning sample is large enough, the prediction result of classification tree would be very accurate. It means that we

could predict the quality of a program by its metrics accurately according to the classification tree model.

However, the disadvantage of classification tree modeling is that it needs large learning data and more data descriptions. In our case, the classification tree result will be more accurate if we had used more programs for learning, and more metrics could be collected to describe the features of various aspects for the given programs.

As BBN constructs the influence diagram of the dependency relationship of the metrics and testing result, it can predict the range of testing results by giving the combination of different metrics. Also, it can suggest the best combination of metrics, which is more clear in BBN than in classification tree modeling, if we want to reduce the testing result to a specific range.

The obvious disadvantage of BBN model is that user should know the dependent relationship very well in his specific domain before he can construct a correct influence diagram and get the prediction result. But this kind of expert acknowledge is usually not available before the prediction results.

In our experiment, as the testing data is restricted, only 18 programs were used to construct the models and validate the prediction. To make the comparison more accurate and fair, we will adopt more programs as test data in our future work. Also, if we could collect data from real systems based on components, we could apply these models to the components as well as the whole systems to get the relationship of their qualities.

Chapter 6

Conclusion

The scale of modern software systems are getting increasingly large and complex. They are not easy to control, resulting in high development cost, low productivity, unmanageable software quality and high risk to move to new technology. Consequently, there is a growing demand of searching for a new, efficient, and cost-effective software development paradigm.

One of the most promising solutions today is the component-based software development (CBSD) approach. This approach is based on the idea that software systems can be developed by selecting appropriate off-the-shelf components and then assembling them with a well-defined software architecture. As CBSD is to build software systems using a combination of components including off-the-shelf components, components developed in-house and components developed contractually, the over quality of the final system greatly depends on the quality of the selected components. We need to first measure the quality of a component before we could certify it. Software metrics are designed to measure different attributes of a

software system and development process, indicating different levels of quality in the final product .

In order to make use of the results of software metrics, several different techniques have been developed to describe the predictive relationship between software metrics and the classification of the software components into fault-prone and non fault-prone categories. These techniques include discriminant analysis, classification trees, pattern recognition, Bayesian network, case-based reasoning (CBR), and regression tree models.

From our observations, conventional Software Quality Assurance (SQA) techniques are not applicable to CBSD due to its special features. For this reason, we investigate the most efficient and effective quality assurance approach suitable to CBSD in our research.

First, we propose a QA model for component-based software development, which covers eight main processes in CBSD: component requirement analysis, component development, component certification, component customization, and system architecture design, integration, testing, and maintenance.

We also propose the Component-based Program Analysis and Reliability Evaluation (ComPARE) environment to evaluate the quality of software systems in component-based programming technology. ComPARE automates the collection of different metrics, the selection of different prediction models, the formulation of user-defined models, and the validation of the established models according to fault

data collected in the development process. Different from other existing tools, ComPARE takes dynamic metrics into account (such as code coverage and performance metrics), integrates them with process metrics and more static code metrics for object-oriented programs (such as complexity metrics, coupling and cohesion metrics, inheritance metrics), and provides different models for integrating these metrics to an overall estimation with higher accuracy.

Finally, we apply different quality predicted techniques on some component-based programs in real world. From the analysis of these predicted results, we also have some discussions on the quality prediction models, which is capable to apply to component-based software systems.

However, as the testing data is restricted in our experiment, only 18 programs are used to construct the models and validate the prediction. To make the comparison more accurate and fair, we will adopt more programs as test data in our future work. Also, in case that we can collect real systems based on components, we can apply these models the components as well as the whole systems to get the relationship of their qualities. We can also consider to adopt other existing quality prediction models to these component-based software system in order to give most appropriate models applicable to CBSD.

Appendix A

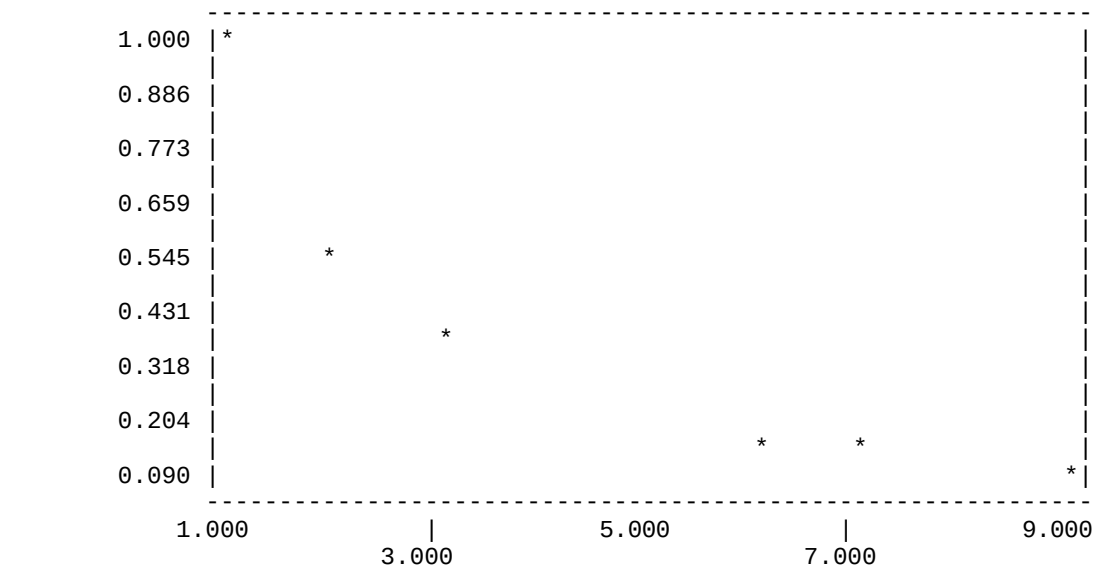
Classification Tree Report of CART

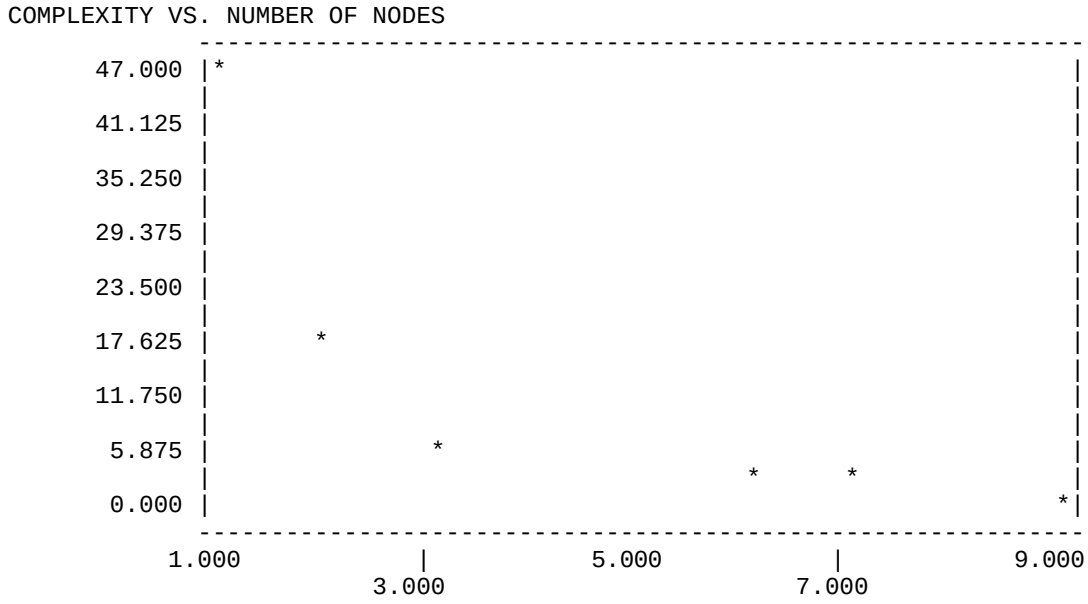
```
CART VERSION 4.0.0.20

 Case weights not supported for LAD rule.

 RECORDS READ: 19
 RECORDS DELETED, DEPENDENT VARIABLE MISSING: 1
 RECORDS WRITTEN IN LEARNING SAMPLE: 18

<a name="12"></a>


 LEARNING SAMPLE VARIABLE STATISTICS
 ===================================

 VARIABLE                  LEARN
 ------------------------------
     TLOC     MEAN|    1905.556
                SD|    1132.905
                 N|      18.000
               SUM|   34300.000
     CLOC     MEAN|    1071.667
                SD|     989.644
                 N|      18.000
               SUM|   19290.000
     SLOC     MEAN|     833.889
                SD|     289.280
                 N|      18.000
               SUM|   15010.000
   CCLASS     MEAN|       4.000
                SD|       2.612
                 N|      18.000
               SUM|      72.000
  CMETHOD     MEAN|      23.611
                SD|      36.398
                 N|      18.000
               SUM|     425.000
   SCLASS     MEAN|       4.611
                SD|       4.828
                 N|      18.000
               SUM|      83.000
  SMETHOD     MEAN|      33.278
                SD|      10.670
                 N|      18.000
               SUM|     599.000
     FAIL     MEAN|       7.778
                SD|       9.932
                 N|      18.000
               SUM|     140.000
 ------------------------------
```

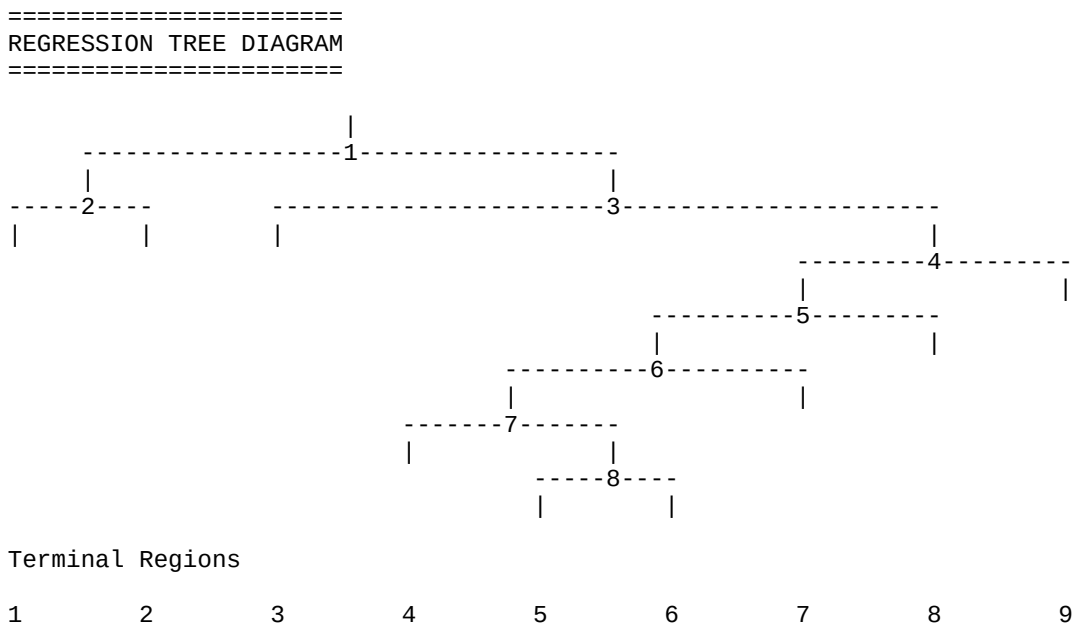
```
 CURRENT MEMORY REQUIREMENTS
     TOTAL:          4555.     DATA:           144  ANALYSIS:          4411.
 AVAILABLE:        9000000.   SURPLUS:        8995445.

 BUILD PREPROCESSOR CPU TIME: 00:00:00.16

 THE DATA ARE BEING READ ...

 18 Observations in the learning sample.
 FILE: D:\RESEARCH\ComPARE\test-data\test-data.XLS[xls7]

 CART IS RUNNING.


 EXPLORATORY BUILD CPU TIME: 00:00:00.05

 Tree constructed with complexity parameter = 0.000

<a name="13"></a>
 =============
 TREE SEQUENCE
 =============
 Dependent variable: FAIL
```

Tree	Terminal Nodes	Resubstitution Relative Error	Complexity Parameter	Relative Complexity	Rho Squared
1	9	0.090	0.000	0.000	0.910
2	7	0.140	2.500	0.025	0.860
3	6	0.170	3.000	0.030	0.830
4	3	0.360	6.333	0.063	0.640
5	2	0.530	17.000	0.170	0.470
6	1	1.000	47.000	0.470	0.000

```
 Initial median = 4.000
 Initial mean absolute deviation = 5.556


 RESUBSTITUTION RELATIVE ERROR VS. NUMBER OF NODES
```







Terminal Regions

1 2 3 4 5 6 7 8 9

NODE INFORMATION

```
                    ************************************
                    *          Node 1: CMETHOD         *
                    *              N: 18               *
                    ************************************

     ******************************     ******************************
     *           Node 2             *     *           Node 3             *
     *            N: 3              *     *            N: 15             *
     ******************************     ******************************

Node 1 was split on CMETHOD
A case goes left if CMETHOD <= 7.000
Improvement =  2.611          Complexity Threshold =  47.000

  Node     Cases    Wgt Count       Median   MeanAbsDev
     1        18        18.00        4.000        5.556
     2         3         3.00       30.000        7.333
     3        15        15.00        3.000        2.067

     Surrogate          Split                          Assoc.     Improve.
   1 SCLASS    r       14.000                           0.333        1.500

     Competitor         Split                                    Improve.
   1 SCLASS            14.000                                       1.500
   2 SMETHOD           21.500                                       0.278
   3 TLOC            2638.000                                       0.222
   4 SLOC             768.500                                       0.167
   5 CCLASS            10.000                                       0.167



                    ************************************
                    *           Node 2: TLOC           *
                    *              N: 3                *
                    ************************************

     ==============================     ==============================
     =       Terminal Node 1      =     =       Terminal Node 2      =
     =            N: 1            =     =            N: 2            =
     ==============================     ==============================

Node 2 was split on TLOC
A case goes left if TLOC <= 1495.500
Improvement =  0.944          Complexity Threshold =  17.000

  Node     Cases    Wgt Count       Median   MeanAbsDev
     2         3         3.00       30.000        7.333
    -1         1         1.00       13.000            .
    -2         2         2.00       35.000        2.500

     Surrogate          Split                          Assoc.     Improve.
   1 CLOC      s      672.500                           1.000        0.944

     Competitor         Split                                    Improve.
   1 CLOC             672.500                                       0.944
   2 SCLASS             1.500                                       0.944
   3 SLOC             932.500                                       0.278
   4 SMETHOD           29.500                                       0.278
```

```
                    ************************************
                    *           Node 3: TLOC           *
                    *              N: 15               *
                    ************************************

     ================================     ******************************
     =       Terminal Node 3        =     *            Node 4            *
     =            N: 1              =     *             N: 14            *
     ================================     ******************************

Node 3 was split on TLOC
A case goes left if TLOC <= 638.500
Improvement =  0.167          Complexity Threshold =  6.333

  Node     Cases    Wgt Count       Median   MeanAbsDev
     3        15        15.00        3.000        2.067
    -3         1         1.00        6.000            .
     4        14        14.00        3.000        2.000

     Surrogate          Split                          Assoc.     Improve.
   1 CLOC      s      269.000                           1.000        0.167

     Competitor         Split                                     Improve.
   1 CLOC             269.000                                        0.167
   2 SLOC             908.500                                        0.111
   3 CCLASS            10.000                                        0.111
   4 CMETHOD           96.500                                        0.111
   5 SCLASS             3.500                                        0.111



                    ************************************
                    *           Node 4: TLOC           *
                    *              N: 14               *
                    ************************************

     ******************************     ================================
     *            Node 5            *     =       Terminal Node 9        =
     *             N: 12            *     =            N: 2              =
     ******************************     ================================

Node 4 was split on TLOC
A case goes left if TLOC <= 2758.500
Improvement =  0.111          Complexity Threshold =  8.000

  Node     Cases    Wgt Count       Median   MeanAbsDev
     4        14        14.00        3.000        2.000
     5        12        12.00        3.000        2.083
    -9         2         2.00        2.000        0.500

     Surrogate          Split                          Assoc.     Improve.
   1 CMETHOD   s       25.500                           0.500        0.056

     Competitor         Split                                     Improve.
   1 CLOC            3234.000                                       0.111
   2 SLOC             908.500                                       0.111
```

```
   3 CCLASS           10.000                                          0.111
   4 CMETHOD          96.500                                          0.111
   5 SCLASS            3.500                                          0.056
```

```
                    ************************************
                    *          Node 5: CMETHOD         *
                    *               N: 12              *
                    ************************************

     ******************************     ================================
     *           Node 6             *     =       Terminal Node 8       =
     *            N: 11             *     =            N: 1             =
     ******************************     ================================
```

```
Node 5 was split on CMETHOD
A case goes left if CMETHOD <= 26.000
Improvement =  0.778          Complexity Threshold =  14.000

  Node     Cases    Wgt Count       Median   MeanAbsDev
     5        12        12.00        3.000        2.083
     6        11        11.00        3.000        1.000
    -8         1         1.00       17.000         .

     Competitor         Split                                      Improve.
   1 SLOC             908.500                                         0.222
   2 TLOC            1625.500                                         0.111
   3 CLOC             555.500                                         0.111
   4 SCLASS             3.500                                         0.111
   5 SMETHOD           21.500                                         0.056
```

```
                    ************************************
                    *           Node 6: SLOC           *
                    *               N: 11              *
                    ************************************

     ******************************     ================================
     *           Node 7             *     =       Terminal Node 7       =
     *            N: 8              *     =            N: 3             =
     ******************************     ================================
```

```
Node 6 was split on SLOC
A case goes left if SLOC <= 908.500
Improvement =  0.167          Complexity Threshold =  3.000

  Node     Cases    Wgt Count       Median   MeanAbsDev
     6        11        11.00        3.000        1.000
     7         8         8.00        3.000        1.000
    -7         3         3.00        4.000         .

     Surrogate          Split                          Assoc.     Improve.
   1 TLOC      s     1625.500                           0.333        0.056

     Competitor         Split                                      Improve.
   1 TLOC             921.500                                         0.056
```

```
   2 CLOC             378.500                                          0.056
   3 CMETHOD           11.500                                          0.056
   4 SCLASS             3.500                                          0.056
   5 SMETHOD           21.500                                          0.056



                    ************************************
                    *           Node 7: TLOC           *
                    *               N: 8               *
                    ************************************

     ================================     ******************************
     =       Terminal Node 4        =     *            Node 8            *
     =            N: 1              =     *             N: 7             *
     ================================     ******************************

 Node 7 was split on TLOC
 A case goes left if TLOC <= 921.500
 Improvement =  0.056          Complexity Threshold =  2.500

   Node     Cases    Wgt Count       Median   MeanAbsDev
      7         8         8.00        3.000        1.000
     -4         1         1.00        2.000         .
      8         7         7.00        3.000        1.000

     Surrogate          Split                          Assoc.     Improve.
   1 CLOC      s      378.500                           1.000        0.056

     Competitor         Split                                     Improve.
   1 CLOC             378.500                                        0.056
   2 CMETHOD           11.500                                        0.056
   3 SCLASS             4.500                                        0.056



                    ************************************
                    *           Node 8: TLOC           *
                    *               N: 7               *
                    ************************************

     ================================     ================================
     =       Terminal Node 5        =     =       Terminal Node 6        =
     =            N: 1              =     =            N: 6              =
     ================================     ================================

 Node 8 was split on TLOC
 A case goes left if TLOC <= 1208.500
 Improvement =  0.222          Complexity Threshold =  4.000

   Node     Cases    Wgt Count       Median   MeanAbsDev
      8         7         7.00        3.000        1.000
     -5         1         1.00        7.000         .
     -6         6         6.00        3.000        0.500


<a name="15"></a>
```

```
 =========================
 TERMINAL NODE INFORMATION
 =========================


Parent
  Node    Wgt Count   Count      Median   MeanAbsDev               Complexity

-------------------------------------------------------------------------------
     1        1.00       1      13.000        0.000                   17.000
     2        2.00       2      35.000        2.500                   17.000
     3        1.00       1       6.000        0.000                    6.333
     4        1.00       1       2.000        0.000                    2.500
     5        1.00       1       7.000        0.000                    4.000
     6        6.00       6       3.000        0.500                    4.000
     7        3.00       3       4.000        0.000                    3.000
     8        1.00       1      17.000        0.000                   14.000
     9        2.00       2       2.000        0.500                    8.000


<a name="16"></a>
 ===================
 VARIABLE IMPORTANCE
 ===================

                 Relative     Number Of   Minimum
                Importance    Categories  Category
 ------------------------------------------------
 CMETHOD          100.000
 TLOC              45.161
 SCLASS            43.548
 CLOC              33.871
 SLOC               4.839
 SMETHOD            0.000
 CCLASS             0.000

 N of the learning sample =       18

<a name="17"></a>
 ===============
 OPTION SETTINGS
 ===============

 Construction Rule           Least Absolute Deviation
 Estimation Method           Exploratory - Resubstitution
 Tree Selection              0.000 se rule
 Linear Combinations         No

 Initial value of the complexity parameter             = 0.000
 Minimum size below which node will not be split       = 2
 Node size above which sub-sampling will be used       = 18
 Maximum number of surrogates used for missing values  = 1
 Number of surrogate splits printed                    = 1
 Number of competing splits printed                    = 5
 Maximum number of trees printed in the tree sequence  = 10
 Max. number of cases allowed in the learning sample   = 18
 Maximum number of cases allowed in the test sample    = 0
 Max # of nonterminal nodes in the largest tree grown  = 38
  (Actual # of nonterminal nodes in largest tree grown = 10)
 Max. no. of categorical splits including surrogates   = 1
```

```
Max. number of linear combination splits in a tree    = 0
     (Actual number cat. + linear combination splits  = 0)
Maximum depth of largest tree grown                   = 13
     (Actual depth of largest tree grown              = 7)
Maximum size of memory available                      = 9000000
     (Actual size of memory used in run               = 5356)


TOTAL CPU TIME: 00:00:00.22
```

Appendix B

Publication List

1. “Component-Based Software Engineering: Technologies, Development Frameworks, and Quality Assurance,” **Xia Cai**, M.R.Lyu, K.F.Wong and R. Ko, Proceedings of Seventh Asia-Pacific Software Engineering Conference (APSEC 2000), Singapore, Dec. 2000, pp.372-379.

2. “ComPARE: A Generic Quality Assessment Environment for Component-Based Software Systems,” **Xia Cai**, M.R.Lyu, K.F.Wong and M.Wong, Proceedings of The 2001 International Symposium on Information Systems and Engineering (ISE'2001), Las Vegas, USA, Jun. 2001, pp. 348-354.

3. “Component-based Embedded Software Engineering: Development Framework, Quality Assurance and A Generic Assessment Environment", **Xia Cai**, M.R.Lyu and K.F.Wong, Accepted by the Special Issue of International Journal of Software Engineering & Knowledge Engineering (IJSEKE) on Embedded Software Engineering, Apr. 2002.

Bibliography

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

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

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

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

[5] IBM: http://www4.ibm.com/software/ad/sanfrancisco, Mar, 2000.

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

[7] W. Kozaczynski, G. Booch, “Component-based software Engineering,” IEEE Software Volume: 155, Sept.-Oct. 1998, pp. 34–36.

[8] M.R.Lyu (ed.), Handbook of Software Reliability Engineering, McGraw-Hill, New York, 1996.

[9] Microsoft: http://www.microsoft.com/com, Nov. 2001.

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

[11] OMG: http://www.omg.org/corba, Nov. 2001.

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

[13] 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.

[14] 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.

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

pp. 398.

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

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

[18] C.Szyperski, "Component Software: Beyond Object-Oriented Programming," Addison-Wesley, New York, 1998.

[19] SUN http://developer.java.sun.com/developer,Mar. 2000

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

[21] 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.

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

[23] S.S.Yau, B. Xia, “Object-Oriented Distributed Component Software

Development based on CORBA," Proceedings of COMPSAC'98, 1998, pp. 246-251.

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

[25] J.Voas and J.Payne, "Dependability Certification of Software Components," *The Journal of Systems and Software,* 52, pp.165-172, 2000,

[26] N. E. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Transactions on Software Engineering,* SE-26(8), pp.797–814, Aug. 2000.

[27] S.S.Gokhale and M.R.Lyu, "Regression Tree Modeling for the Prediction of Software Quality," *Proceedings of Third ISSAT International Conference on Reliability and Quality in Design,* Anaheim, California, March 1997.

[28] Metamata: http://www.metamata.com/metrics.html, Nov. 2001.

[29] Jprobe: http://www.sitraka.com/software/jprobe/, Nov. 2001.

[30] J.Munson and T.Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering,* SE-18(5), May 1992.

[31] A. A. Porter and R. W. Selby, "Empirically Guided Software Development Using Metric-Based Classification Trees," *IEEE Software,* pp. 46-53, Mar.1990.

[32] L.C.Briand, V.R.Basili and C.Hetmanski, "Developing Interpretable Models for Optimized Set Reduction for Identifying High-Risk Software Components,"

IEEE Transactions on Software Engineering, SE-19(11), pp.1028-1034, Nov.1993.

[33] N.E.Fenton and M.Neil, "A Critique of Software Defect Prediction Models," *IEEE Transactions on Software Engineering*, SE-25(5), pp.675-689, Oct. 1999.

[34] K.E.Emam, S.Benlarbi, N.Goel and S.N.Rai, "Comparing Case-Based Reasoning Classifiers for Predicting High Risk Software Components," *The Journal of systems and Software*, 55, pp.301-320, 2001.

[35] http://www.hugin.com, Nov. 2001.

[36] M.R.Lyu, J.S.Yu, E.Keramidas and S.R.Dalal, "ARMOR: Analyzer for Reducing Module Operational Risk," *Proceedings of Twenty-Fifth International Symposium on Fault-Tolerant Computing* (FTCS-25), pp.137-142, 1995.

[37] A.A.Keshlaf and K.Hashim, "A Model and Prototype Tool to Manage Software Risks," *Proceedings of First Asia-Pacific Conference on Quality Software*, pp.297-305, 2000.

[38] J.F.Patenaude, E.Merlo, M.Dagenais and B.Lague, "Extending Software Quality Assessment Techniques to Java Systems," *Proceedings of Seventh International Workshop on Program Comprehension*, pp.49-56, 1999.

[39] T.Systa, Y.Ping and H.Muller, "Analyzing Java Software by Combining Metrics and Program Visualization," *Proceedings of Fourth European Software*

Maintenance and Reengineering, pp.199 –208, 2000.

[40] A.Rhodes,, "Component-Based Development for Embedded Systems," Proceedings of Systems Conference, No.313.

[41] Salford Systems: http://www.salford-systems.com, Nov. 2001.

[42] D.J.Smith, "Achieving Quality Software (Third Edition)," Chapman & Hall, 1995

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

[44] Flashline: http://www.flashline.com, Nov. 2001

[45] G. Xing and M.R. Lyu, "Testing, Reliability, and Interoperability Issues in the CORBA Programming Paradigm," Proceedings of 1999 Asia-Pacific Software Engineering Conference (APSEC'99), pp. 530-536, 1999.