A Generic Environment for COTS Testing and Quality Prediction

Xia Cai[1], Michael R. Lyu[1], and Kam-Fai Wong[2]

Dept. of Computer Science and Engineering[1],Dept. of Computer Science and Engineering[2],The Chinese University of Hong Kong,Shatin, N.T., Hong Kong, China

{xcai,lyu}@cse.cuhk.edu.hk,kfwong@se.cuhk.edu.hk

Summary. In this chapter, we first survey current component technologies and discuss the features they inherit. Quality assurance (QA) characteristics of component systems, and the life cycle of component-based software development (CBSD) are also addressed. Based on the characteristics of the life cycle, we propose a QA model for CBSD. The model covers the eight main processes in component-based software systems (CBS) development. A Component-based Program Analysis and Reliability Evaluation (ComPARE) environment is established for evaluation and prediction of quality of components. ComPARE provides a systematic procedure for predicting the quality of software components and assessing the reliability of the final system developed using CBSD. Using different quality prediction techniques, ComPARE has been applied to a number of component-based programs. The prediction results and the effectiveness of the quality prediction models for CBSD were outlined in this paper.

1.1 Introduction

Based on the component-based software development (CBSD) approach[1], software systems are developed using a well-defined software architecture and off-the-shelf components (COTS) as building bricks [2]. It is different from the traditional approach in which software systems are implemented from scratch. Commercial off-the-shelf (COTS) components are developed by different developers using different languages and different platforms [3]. Typically, COTS components are available from a component repository; users select the appropriate ones and integrate them to establish the target software system (see Figure1.1).

In general, a component has three main features: 1) it is an independent and replaceable part of a system that fulfills a clear function; 2) it works in the context of a well-defined architecture; and 3) it communicates with

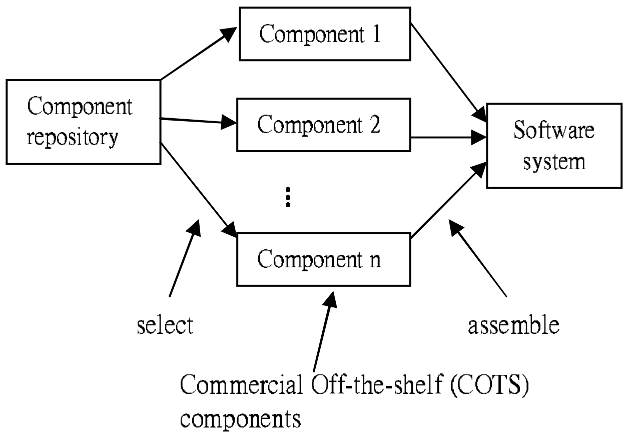


Fig. 1.1. Component-based software development

other components by its interfaces [4]. Current component technologies have been used to implement different software systems, such as object-oriented distributed component software [5] and Web-based enterprise applications [6].

The system architecture of a component-based software system is layered and modular [7, 8, 9], see Figure 2.

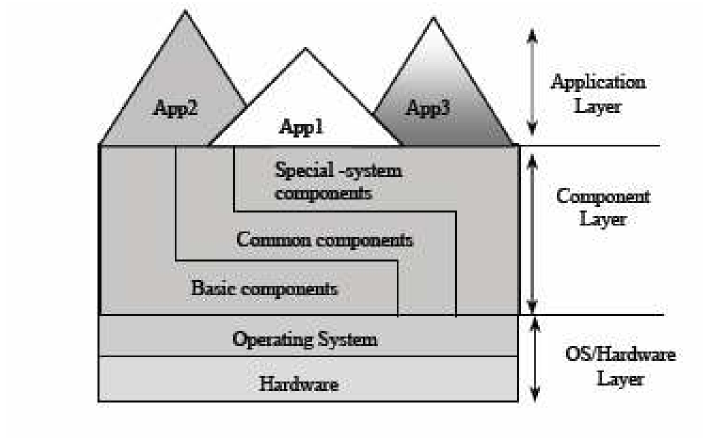


Fig. 1.2. System architecture of component-based software systems

The top application layer entails information systems designed for various applications. The second layer consists of components for a specific system or application domains. Components in this layer are applicable to more than one single application. The third layer comprises of cross-system middleware components which includes common software and interfaces to other established entities. The fourth layer of system software components includes basic components that interface with the underlying operating systems and hosting hardware. Finally, the lowest two layers involves the operating and hardware systems.

A CBSD-based software system is composed of one or more components, which may be procured from off-the-shelf, produced in-house or developed by contracts. The overall quality of the final system depends heavily on the quality of the components involved. One needs to be able to assess the quality of a component to reduce the risk of development. Software metrics are designed to measure different attributes of a software system and the development process, and are used to evaluate the quality of the final product [10]. Process metrics (e.g., reliability estimates) [11], static code metrics (e.g., code complexity) [12] and dynamic metrics (e.g., test thoroughness) [13] are widely used to predict the quality of software components at different development phases [10, 14].

Several techniques are used to model the predictive relationship between different software metrics and for component classification, i.e., classifying software components into fault-prone and non fault-prone categories [15]. These techniques include discriminant analysis [16], classification trees [17], pattern recognition [18], Bayesian network [19], case-based reasoning (CBR) [20], and regression tree models [15]. There are also prototypes and tools [11, 21], which use such techniques to automate software quality prediction. However, these tools employ only one metric, e.g., process metrics or static code metrics. Furthermore, they rely on only one prediction technique for overall software quality assessment.

The objective of this chapter is to evaluate individual quality of off-the-shelf components and overall quality of software systems. We integrate different prediction techniques and different software metric categories to form a single environment, and investigate their effectivenss on quality prediction of components and CBS.

The rest of this chapter is organized as follows: we first give an overview of the state-of-the-art CBSD techniques in Section 2, and highlight the quality assurance (QA) issues behind them in Section 3. Section 4 proposes a QA model which is designed for quality management in CBSD process. In Section 5, we propose ComPARE, a generic quality assessment environment for CBSD. It facilitates quality evaluation of individual components as well as the target systems. Different prediction models have been applied to real world CORBA programs. In Section 6, the pros and cons of these prediction models are analyzed. Finally, Section 7 concludes this chapter.

1.2 A Development Framework for Component-Based Software Systems

A framework can be defined as a set of constraints on components and their interactions, and a set of benefits that derive from those constraints [22]. To identify the development framework for component-based software, the framework or infrastructure for components should be identified first, as components are the basic units in the component-based software systems.

Visual Basic Controls (VBX), ActiveX controls, class libraries, JavaBeans, etc., make it possible for their corresponding programming languages, i.e. Visual Basic, C++, Java, and the supporting tools to share and distribute application fragments. But all these approaches rely on certain underlying services to provide communication and coordination. The infrastructure of components (sometimes called a component model) acts as the "plumbing" that allows communication among components [4]. Among the component infrastructure technologies that have been developed, there are three de facto industrial standards: OMG's CORBA, Microsoft's Component Object Model (COM) and Distributed COM (DCOM), and Sun's JavaBeans and Enterprise JavaBeans [23].

1.2.1 Common Object Request Broker Architecture (CORBA)

CORBA is an open standard for interoperability. It is defined and supported by the Object Management Group (OMG), an organization of over 400 software vendor and object technology user companies [24]. CORBA manages details of component interoperability, and allows applications to communicate with one another despite of different locations and designs. Interface is the only way, which applications or components communicate.

The most important part of a CORBA system is the Object Request Broker (ORB). ORB is the middleware that establishes client-server relationship between components. Using an ORB, a client can invoke a method on a server object, whose location is completely transparent. ORB is responsible for intercepting a call and finding an object, which 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, which are not related to the interface. In this way, ORB supports interoperability among applications on different machines in heterogeneous distributed environments and can seamlessly interconnect multiple object systems.

CORBA is widely used in Object-Oriented distributed systems [5] including component-based software systems for it offers a consistent distributed programming and run-time environment over common programming languages, operating systems, and distributed networks.

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

Component Object Model (COM) is a general architecture for component software [25]. It supports platform-dependent, based on Windows and Windows NT, and language-independent component-based applications.

COM defines how components and their clients interact. As such, a client and a component can be connected without the support of an intermediate system component. In particular, COM provides a binary standard, which components and their clients must follow to ensure dynamic interoperability. This enables on-line software update and cross-language software reuse [26].

Distributed COM (DCOM) is an extension of the Component Object Model (COM). It is a protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner. DCOM supports multiple network protocols, 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 of changes in the physical connections.

1.2.3 Sun Microsystems's JavaBeans and Enterprise JavaBeans

Sun's Java-based component model consists of two parts: the JavaBeans for client-side component development and the Enterprise JavaBeans (EJB) for the server-side component development. The JavaBeans component architecture supports multiple platforms, as well as reusable, client-side and server-side components [27].

Java platform offers an efficient solution to the portability and security problems through the use of portable Java bytecode and the concept of trusted and untrusted Java applets. Java provides a universal integration and enabling technology for enterprise application integration (EAI). The technology enables 1) interoperation across multivendor servers; 2) propagation of transaction and security contexts; 3) multilingual clients; and 4) supporting ActiveX via DCOM/CORBA bridges.

JavaBeans and EJB extend the native strength of Java including portability and security to component-based development. The portability, security, and reliability nature of Java are well suited for developing robust server objects independent of operating systems, Web servers and database management servers.

1.2.4 Comparison among Different Architectures

Comparison bwteen the development technologies for component-based software systems can be found in [4, 28, 29]. Table 1.1 summarizes their different features.

Table 1.1. Comparison of development technologies for component-based software systems

	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 and 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 and maintenance	CORBA IDL for defining component interfaces, need extra modification and maintenance	Not involving IDL files, defining interfaces between component and container. Easier modification and maintenance.	Microsoft IDL for defining component interfaces, need extra modification and 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 in general Web clients.	Strongest in traditional desktop applications

1.3 Quality Assurance for Component-Based Software Systems

1.3.1 The Development Life Cycle of Component-Based Software Systems

A component-based software system (CBS) is developed by assembling different components rather than programming from scratch. Thus the life cycle of a component-based software system is different from that of a traditional software system. The cycle can be summarized as follows [2]: 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 CBS defines a system in terms of computational components and interactions among components. The focus is on composing and assembling components. Composition and assembly mostly take place separately, and even independently. Component identification, customization and integration are crucial activities in the development life cycle of CBS. It includes two main parts: 1) evaluation of candidate COTS based on the functional and quality requirements provided by the user; and 2) customization of suitable candidate COTS prior to integration. Integration involves communication and coordination among the selected components.

Quality assurance (QA) for CBS targets every stage of the development life cycle. QA technologies for CBS are currently premature as specific characteristics of component systems are not accounted for. Although some QA techniques such as reliability analysis model for distributed software systems [30, 31] and component-based approach to Software Engineering [32] have been studied, there is still no clear and well-defined standards or guidelines for CBS. The identification of the QA characteristics, along with the models, tools and metrics, are all under urgent needs.

1.3.2 Quality Characteristics of Components

QA technologies for component-based software development has to cater for two inseparable parts: 1) How to ensure the quality of a component? 2) How to ensure the quality of the target component-based software system? To answer these questions, models should be defined for quality control of individual components and the target CBS; metrics should be defined to measure the size, complexity, reusability and reliability of individual components and the target CBS; and tools should be designed to evaluate existing components and CBS.

To evaluate a component, we must determine how to assess the quality of the component [33, 34]. Here we propose a list of component features for the assessment: 1) Functionality; 2) Interface; 3) Usability; 4) Testability; 5) Maintainability; 6) Reliability.

Software metrics can be proposed to measure software complexity [35, 36]. Such metrics are often used to classify components [37]. They include
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 ensure high quality.
2) Complexity. This also affects reuse cost and quality. A component which is too-trivial is not worthwhile to modularize. But on the other hand, a component which is too complex is hard to ensure high quality.
3) Reuse frequency. The number of times and different domains where a component has been used previously is a solid indicator of its usefulness.
4) Reliability. The probability of failure-free operations of a component under certain operational scenarios [38].

1.4 A Quality Assurance Model for Component-Based Software Systems

Since component-based software systems are developed on an underlying process different from that of traditional software, their quality assurance model should address both the process of componentization and the process of the overall system development. Figure 1.3 illustrates this view.

Many standards and guidelines are used to control the quality activities of traditional software development process, such as ISO9001 and CMM model

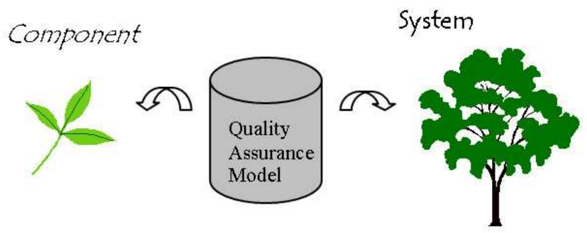


Fig. 1.3. Quality assurance model for both components and systems

[39]. In particular, Hong Kong productivity Council has developed the HK-SQA model to localize the general SQA models [40]. In this section, we propose a quality assurance model for component-based software development.

In our model, the main practices relating to components and software systems 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.

1.4.1 Component Requirement Analysis

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

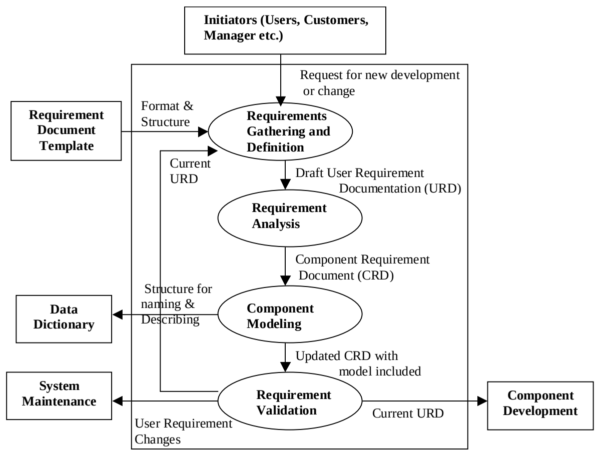

Initiators (Users, Customers, Manager etc.)
Request for new development or change
Requirement Document Template
Format & Structure
Requirements Gathering and Definition
Current URD
Draft User Requirement Documentation (URD)
Requirement Analysis
Component Requirement Document (CRD)
Data Dictionary
Structure for naming & Describing
Component Modeling
Updated CRD with model included
System Maintenance
Requirement Validation
Current URD
Component Development
User Requirement Changes

Fig. 1.4. Component requirement analysis process overview

The component requirement process overview diagram is as shown in Figure 1.4. Initiated by the users or customers for a new development or changes to an 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, the user requirement changes for the system maintenance phase, and data dictionary for all the latter phases.

1.4.2 Component Development

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

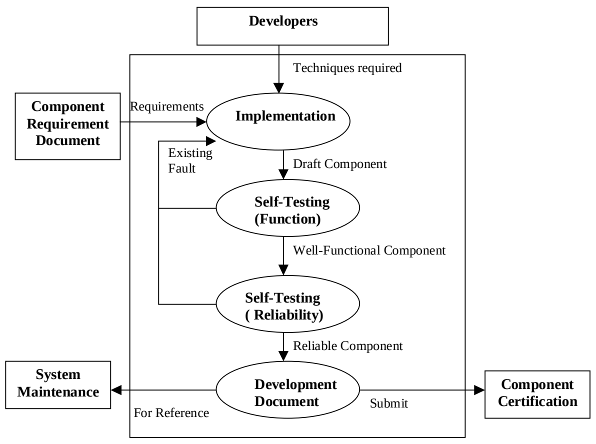


Fig. 1.5. Component development process overview

The component development process overview diagram is shown in Figure 1.5. Component development consists of four procedures: implementation, function testing, reliability testing, and development documentation. 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.

1.4.3 Component Certification

Component certification is the process, which 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 requirements for both functionality and reliability; and 3) component testing: confirm that the component satisfies the requirement with acceptable quality and reliability.

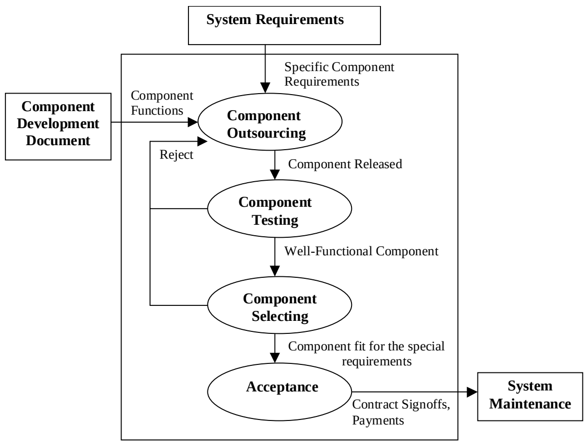

System Requirements
Specific Component Requirements
Component Development Document
Component Functions
Component Outsourcing
Reject
Component Released
Component Testing
Well-Functional Component
Component Selecting
Component fit for the special requirements
Acceptance
Contract Signoffs, Payments
System Maintenance

Fig. 1.6. 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 in a simulated environment. The component certification process overview diagram is shown in Figure 1.6. The input to this phase is the component development documents, and the output is the testing documentation for system maintenance.

1.4.4 Component Customization

Component customization is the process which involves 1) modifying the component for specific requirements; 2) making necessary changes to the component for running on the local platforms; 3) upgrading the specific component to get a better performance or a higher quality. The objective of component customization is to make necessary changes to a developed component so that it can be used in a specific environment or cooperate with other components well.

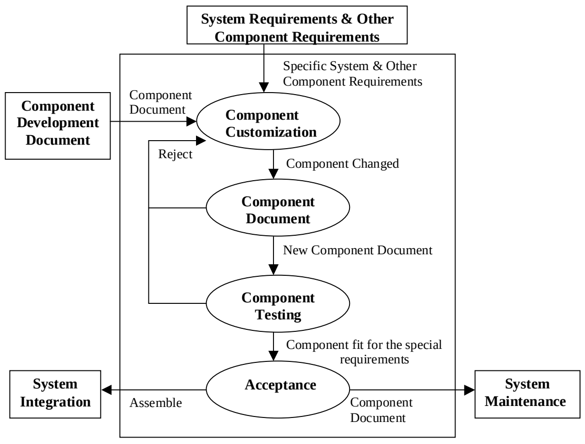


Fig. 1.7. Component customization process overview

All components must be customized according to the operational system requirements or the interface requirements. The component customization process overview diagram is shown in Figure 1.7. The input to component customization are the system requirements, the component requirements, and component development documents. The output are the customized components, and documents for system integration and system maintenance.

1.4.5 System Architecture Design

System architecture design is the process of evaluating, selecting and creating software architecture of a component-based software system. The objectives of system architecture design are to collect the users requirements, determine the system specification, select appropriate system architecture, and determine the implementation details such as platform, programming languages, etc.

System architecture design should compare the pros and cons of different system architectures and select the one suitable for the target CBS. The process overview diagram is shown in Figure 1.8. This phase consists of system requirement gathering, analysis, system architecture design, and system specification. The output of this phase comprises of the system specification document for system integration, and the system requirements for the system testing and system maintenance phases.

1.4.6 System Integration

System integration is the process of properly assembling the components selected to produce the target CBS under the designed system architecture. The process overview diagram is shown in Figure 1.9. The input are the system requirement documentation and the specific architecture. There are four steps

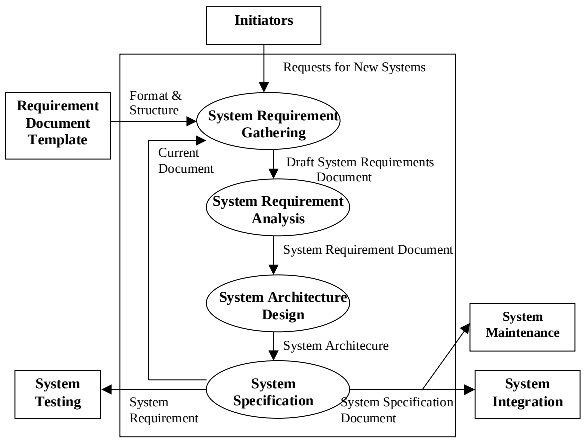

Initiators
Requests for New Systems
Requirement Document Template
Format & Structure
System Requirement Gathering
Current Document
Draft System Requirements Document
System Requirement Analysis
System Requirement Document
System Architecture Design
System Architecure
System Maintenance
System Testing
System Requirement
System Specification
System Specification Document
System Integration

Fig. 1.8. System architecture design process overview

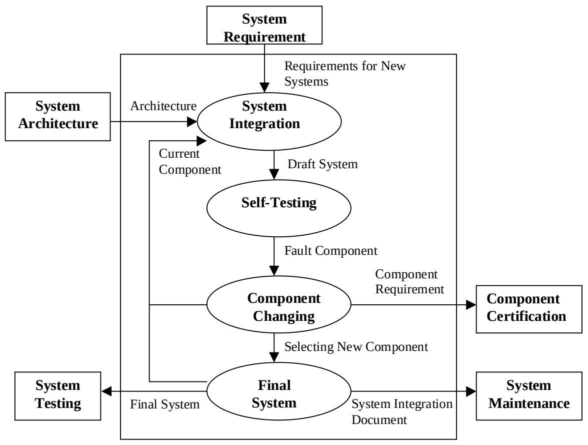


Fig. 1.9. System integration process overview

in this phase: integration, testing, changing component and re-integration (if necessary). At the end of this phase, the final target system will be ready for system testing, and the appropriate document for the system maintenance phase.

1.4.7 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. System testing includes function testing and reliability testing. The process overview diagram is shown in Figure 1.10. This phase consists of selecting testing strategy, system testing, user acceptance testing, and completion activities. The

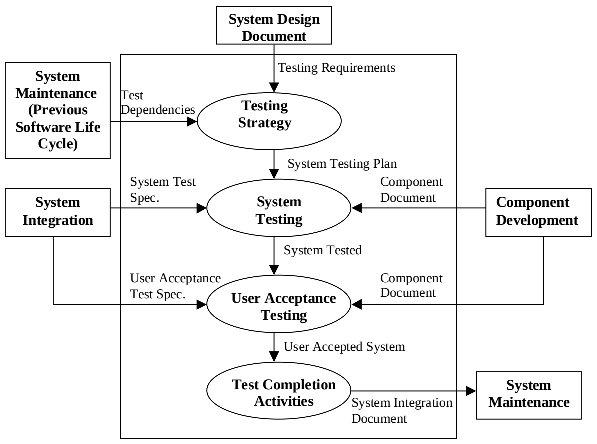


Fig. 1.10. System testing process overview

input comprises of the documents from the component development and system integration phases. And the output includes the testing documentation for system maintenance. Note that this procedure must cater for the interaction testing between multiple components, which includes coordination issues, deadlocks, etc.

1.4.8 System Maintenance

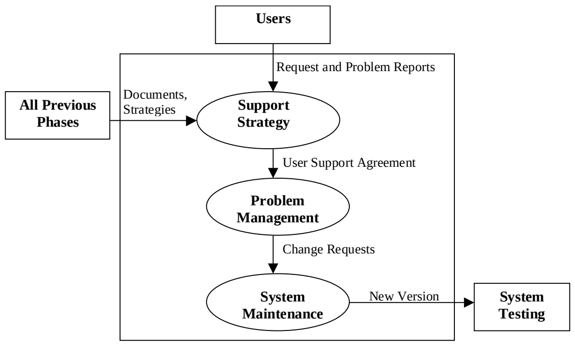


Fig. 1.11. System maintenance process overview

System maintenance is the process of providing service and maintenance activities required 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 repairing faults, improving software performance or other attributes, and adapting the system to a changed environment.

A maintenance organization should be available for every CBS product. All changes for the delivered system should be reflected in the related documents. The process overview diagram is shown in Figure 1.11. According to the outputs from all previous phases as well as requests and problem reports from users, system maintenance should be performed to determine the setup support and problem management (e.g., identification and approval) strategies. This phase produces a new version of the CBS, which may be subjected to further system testing.

1.5 A Generic Quality Assessment Environment for Component-Based Systems - ComPARE

We propose a 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 faulty data collected in the development process. Different from other existing tools [21], ComPARE takes dynamic metrics into account (such as code coverage and performance metrics), integrates them with process metrics and other static code metrics (such as complexity metrics, coupling and cohesion metrics, inheritance metrics) which are adopted from object-oriented software engineering, and provides different estimation models for overall system assessment.

1.5.1 Overall Architecture

A number of commercial tools are available for the measurement of software metrics for object-oriented programming. Also there are off-the-shelf tools for testing and debugging of software components [41]. 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 the reliability of the overall system based on them. The overall architecture of ComPARE is shown in Figure 1.12. First of all, various metrics are computed for the candidate components, then the users can select and weigh the metrics deemed important to quality assessment. After the models have been constructed and executed (e.g., "case base" is used in BBN model), the users can validate the selected models with previous failure data collections. If the users are not satisfied with the prediction result, 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 based on the architecture of the candidate system. Results from individual components can also be displayed for sensitivity analysis and system redesign.

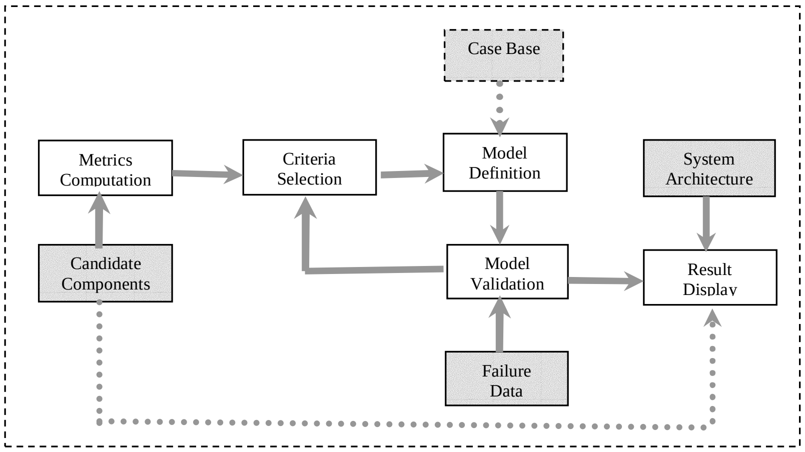


Fig. 1.12. Architecture of ComPARE

The objectives of ComPARE are summarized as follows:

1. To predict the overall quality 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, the users 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, the user can select from several quality prediction models and select the one most suitable for this prediction task. Moreover, the user can also define their own models and validate them in the evaluation stage.

4. To classify components using different quality categories. Once the metrics are computed and the models 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., 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 in the environment) at line-level granularity. This can help the users locate high risk program modules or portions promptly and conveniently.

7. To adopt commercial tools in accessing software data related to quality attributes. We adopt Metamata [42] and Jprobe [43] suites to measure the different metrics for the candidate components. These two tools, involving metrics, audits, debugging, as well as code coverage, memory and deadlock detected, are commercially available.

1.5.2 Metrics Used in ComPARE

Table 1.2. Process Metrics

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

Three different categories of metrics, namely process, static, and dynamic, are analyzed in CompARE to give the overall quality prediction. We have chosen proven metrics, i.e., those that are widely adopted by previous software quality prediction tools in the software engineering research community [44, 45]. The process metrics we selected are listed in Table 1.2 [11]. Since we perceive that Object-Oriented (OO) techniques are essential in component-based software development, we select static code metrics according to the most important features in OO programs, i.e., complexity, coupling, inheritance and cohesion. They are listed in Table 1.3 [12, 13, 42, 46]. The dynamic metrics measuring component features when they are executed. Table 1.4 shows the detailed description of the dynamic metrics.

Sets of process, static, and dynamic metrics can be collected from commercial tools, e.g., Metamata Suite [42] and Jprobe Testing Suite [43]. We adopt these metrics in ComPARE.

1.5.3 Models Definition

In order to predict the quality of software systems, several techniques have been developed to classify software components according to their reliability [15]. These techniques include discriminant analysis [16], classification trees [17], pattern recognition [18], Bayesian network [19], case-based reasoning (CBR) [20], and regression tree model [11].

Up to now, there is no good quality prediction models for CBS. Here we set some evaluation criteria for good quality prediction models [47]: 1)Useful quantities: the model can make predictions of quantities reflecting software

Table 1.3. Static Code Metrics

Abbreviation	**Description**
Lines of Code FF(LOC)	Number of lines in the components including statements, blank lines, lines of commentary, and 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 Attri-butes (NA)	Number of fields declared in the class or interface.
Number Of Classes (NOC)	Number of classes or interfaces, which 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, which are used in the field declarations of the class or interface.
Fan Out (FANOUT)	Number of reference types, which are used in field declarations, formal parameters, return types, throws declarations, and local variables.
Coupling between Objects (CO)	Number of reference types, which 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 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 then increase the count P by one. If they share at least one field then increase Q by one. After considering each pair of methods, $$\begin{aligned} LCOM &= (P - Q) \quad if \quad P > Q \\ &= 0 \qquad \text{otherwise} \end{aligned}$$

Table 1.4. Dynamic Metrics

Metric	**Description**
Test Case Coverage	The coverage of the source code when the given test cases are executed.
Call Graph metrics	Statistics about a method, 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.

quality. 2)Prediction accuracy: the model can make predictions of quality which can be accurately observed later. 3)Ease of measuring parameters: the parameters in the model are easily measured or simulated. 4)Quality of assumptions: the assumptions should be reasonable, rather than too narrow or limited. 5)Applicability: the model should be widely used in various projects or experiments. 6)Simplicity: the model should not be too hard to implement or realize.

In ComPARE, we combine existing quality prediction models according to the above criteria. Initially, one employs an existing prediction model, e.g., classification tree model or BBN model, customizes it and compares the prediction results with different tailor-made models. In particular, we have investigated the following prediction models and studied their applicability to ComPARE in our research.

Summation Model

This model gives a prediction by simply adding all the metrics selected and weighted by the 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 is formulated as follows:

$$Q = \sum_{i=1}^{n} \alpha_i m_i \tag{1.1}$$

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

Product Model

Similar to the summation model, the product model multiplies all the metrics selected and weighted by the user. 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 follows:

$$Q = \prod_{i=1}^{n} m_i \tag{1.2}$$

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_i's are normalized to a value close to 1, so that no single metric can dominate the result.

Classification Tree Model

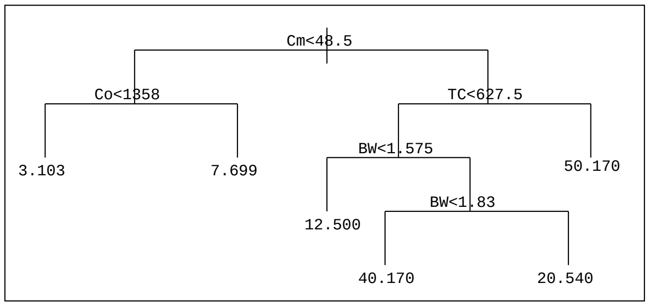


Fig. 1.13. An example of the classification tree model

Classification tree model [17] classifies candidate components into different quality categories by constructing a tree structure. All candidate components (with certain failure rate) form the leaves of the tree. Each node of the tree represents a metric (or a composed metric calculated by other metrics) of a certain value. All children of the left sub tree of a node represent those components whose value of the same metric is smaller than the value of the node. Similarly, all children of the right sub-tree of a node are those components whose value of the same metric is equal to or larger than the value of the node. Figure 1.13 gives an example of the classification tree model.

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. Again, the user can validate and evaluate the final tree model after its definition. Figure 1.13 is an example of the outcome of a tree model, where Cm(number of comments), Co(code characters), Tc(total line of code) and BW(Belady's bandwith metric) are sample metrics [15]. 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.

Case-Based Reasoning Model

Case-based reasoning (CBR) has been proposed for predicting quality of software components [20]. A CBR classifier uses previous ”similar” cases as the basis for prediction. 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 a candidate component which has a similar structure as a component in the case base will be assigned to a similar quality level.

A CBR classifier can be instantiated in different ways by varying its parameters. But according to previous research, there is no significant difference in prediction validity when using any combination of parameters in CBR. For this reason, we adopt the simplest CBR classifier modeling with Euclidean distance, z-score standardization [20], and without weighting scheme. Finally, we select the single, nearest neighbor for prediction.

Bayesian Network Model

Bayesian networks (also known as Bayesian Belief Networks, BBN) is a graphical network that represents probabilistic relationships among variables [19]. BBNs enable reasoning under uncertainty. Besides, 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, and provides a tangible visual representation as well as a sound mathematical basis in Bayesian probability [48]. The foundation of Bayesian networks is based on the following theorem, which is known as the Bayes' Lemma:

$$(H|E,c) = \frac{P(H|c)P(E|H,c)}{P(E|c)} \tag{1.3}$$

where H, E, c are independent events. P is 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 [19]. Users can also define their own BBN models in ComPARE and compare the results with other models.

1.5.4 Operations in ComPARE

ComPARE suggests eight functions: File Operations, Metrics Selection, Criteria Selection and Weighting, Model Selection and Definition, Model Validation, Display Result, Windows Switch, and Help. The details of some of these key functions are described in the following:

Metrics Selection

Users can select the metrics they want to collect for the component-based software systems. Three categories of metrics are available: process metrics, static metrics and dynamic metrics. The details of these metrics are shown in Section 1.5.2.

Criteria Selection and Weighting

After computing different metrics, the users will select and weigh the criteria associated to these metrics before using them. Each metric can be assigned a weight between 0 and 1.

Model Selection and Definition

This operation allows the users to select or define the model they would like to use in the evaluation. The users are required to provide the probability of each metric, which affects the quality of the candidate component.

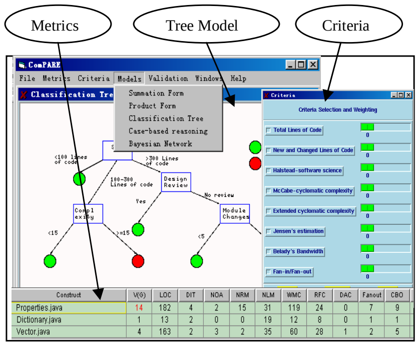
Model Validation

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

1.5.5 Prototype

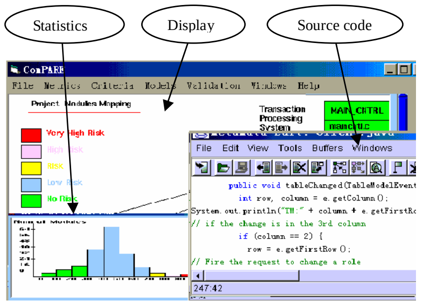
We have developed a ComPARE prototype for QA of Java-based components and CBS. Java is one of the most popular languages used in off-the-shelf components development today. It is a common language binding the three standard architecture of component-based software development: namely CORBA, DCOM and Java/RMI.

Figure 1.14 and Figure 1.15 show screen dumps of the ComPARE prototype. The computation of various metrics for software components and application of quality prediction models can be seen as a straightforward process. 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 the users to identify an appropriate prediction model for a given component-based software system.



Construct	V(G)	LOC	DIT	NOA	NRM	NLM	WMC	RFC	DAC	Fanout	CBO
Properties.java	14	182	4	2	15	31	119	24	0	7	9
Dictionary.java	1	13	2	0	0	19	12	8	0	1	1
Vector.java	4	163	2	3	2	35	60	28	1	2	5

Fig. 1.14. GUI of ComPARE for metrics, criteria and tree model



```
public void tableChanged(TableModelEvent
  int row, column = e.getColumn();
System.out.println("TM:" + column + e.getFirstRo
// if the change is in the 3rd column
  if (column == 2) {
    row = e.getFirstRow();
// Fire the request to change a role
```

Fig. 1.15. GUI of ComPARE for prediction display, risky source code and result statistics

1.6 Experiment and Discussion

1.6.1 Objective

ComPARE provides a systematic procedure for predicting the quality of software components and for assessing their reliability in the final target system. As there is no existing QA model for CBS, ComPARE adopts existing quality prediction models.

In this section, we investigate the effectiveness of different existing quality prediction models and their applicability to CBS. In our experiment, we use the five models selected in Section 1.5.3 to predict and evaluate the relationship between the number of faults and software metrics of some CORBA programs obtained in a component-based software engineering experiment. In this experiment, all programs were designed according to the same specification. The programming teams could choose their own programming languages. The test cases were designed to assess the functionalities of the final programs according to the specification. The details of the testing and evaluation of this experiment is shown in [49]. We applied the selected prediction models to the final CORBA programs and investigate how well they behave. This information is useful for the users in determining which is the appropriate quality prediction models.

1.6.2 Data Description and Experiment Procedure

In the fall of 1998 we engaged 19 programming teams to design, implement, test and demonstrate a Soccer Team Management System using CORBA. This was a class project for 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. The system should allow 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 had 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. In total, 57 test cases were used in the experiment.

Among these 19 programs 12 chose to use Visibroker, and 7 Iona Orbix. For the 12 Visibroker programs, 9 used Java and 2 used C++ for both client and server implementation, and 1 used Java and C++ for client and server respectively. Because Team 1 did not pass the acceptance test, we will not include it in our evaluations. The metrics collected and the test results for the 18 different program versions are shown in Table 1.5. 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 the client and server programs;
- Client LOC (CLOC): lines of codes in the client program;
- Server LOC (SLOC): lines of codes in the server program;
- Client Class (CClass): number of classes in the client program;
- Client Method (CMethod): number of methods in the client program;
- Server Class (SClass): number of classes in the server program;
- Server Method (SMethod): number of methods in the 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.

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 [42] and JProbe Suite [43] 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

Table 1.5. General Metrics of Different Teams

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

cases and cases that caused exceptions in the system. For each operation in the system, at least 1 normal test case was conducted in 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 validated the prediction results of these models against the test results. We divided the programs into two groups: training data and testing 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.

1.6.3 Experiment Results

Summation Model

The Summation model gives a prediction by simply adding all the metrics selected and weighted by the user. For simplicity, we gives equal weighting factor for all the metrics, e.g., all metrics = 1. Also we normalize the values of metrics by using the ratio of the actual value to the maximum value of that particular metric, i.e., $m_1 = \frac{TLOC}{max(TLOC)}$, $m_2 = \frac{CLOC}{max(CLOC)}$, etc. for every program. The overall quality mark then is $Q = m_1 + m_2 + \cdots$ for all the 18 programs. The result of the summation model is listed in Table 1.6.

Product Model

The Product model multiplies all the metrics selected and weighted by the user. The values of metrics are also normalized to values close to 1 using the same method as in above section. The final result is the product of these normalized values. It is listed in Table 1.6.

Classification Tree Results Using CART

We adopted commercial tools CART [48] in our classification tree modeling. 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: 1) splitting each node in a tree; 2) deciding when a tree is complete; and 3) assigning each terminal node to a class outcome (or predicted value for regression).

Table 1.6. Results of Summation Model and Product Model

Team	Summation Modeling	Product Model	Fail	Maybe	R	R1
P2	7.00	0.0000159	7	6	0.77	0.88
P3	1.62	0.0002658	3	6	0.84	095
P4	2.69	0.0000030	3	12	0.74	0.95
P5	1.62	0.0001134	2	1	0.95	0.96
P6	2.68	0.0000013	13	10	0.60	0.77
P7	1.82	0.0002813	3	14	0.70	0.95
P8	2.53	0.0000577	1	6	0.88	0.98
P9	1.97	0.0002036	4	2	0.89	0.93
P10	2.50	0.0000323	2	2	0.93	0.96
P11	2.08	0.0000007	6	3	0.84	0.89
P12	1.13	0.0788932	1	4	0.91	0.98
P13	5.44	0.0002482	17	19	0.37	0.70
P14	2.50	0.0001391	4	6	0.82	0.93
P15	2.49	0.0000040	2	5	0.88	0.96
P16	1.50	0.0000808	30	0	0.47	0.47
P17	2.94	0.0000853	3	3	0.89	0.95
P18	2.03	0.0000213	4	9	0.77	0.93
P19	1.83	0.0000047	35	1	0.37	0.39

Table 1.7. Option Setting of the classification tree

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 1.8. Importance of different variables in the classification tree

Metrics	Relative Importance	Number of Categories	Minimum 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			

We applied the metrics and testing results in Table 1.5 to the CART tool, and collected the classification tree results for predicting the quality variable "Fail". Table 1.7 is the option setting of the classification tree. The tree constructed is shown in Figure 1.16, and the relative importance of each metric is listed in Table 1.8. From Figure 1.16, we can see that the 18 learning samples are classified into 9 groups (terminal nodes), whose information are listed in Table 1.9. The most important vector was the number of methods in the client program (CMethod), and the next three most important vectors were

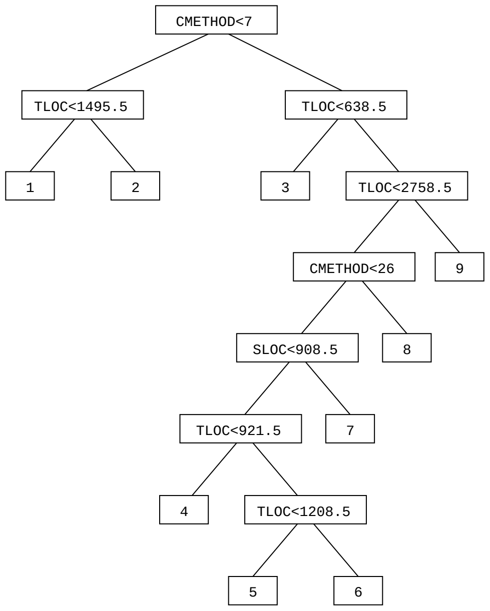


Fig. 1.16. Classification tree structure

TLOC, SCLASS and CLOC. From the node information, we 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 1.10.

Table 1.9. Terminal node information in the classification tree

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 1.10. Relationship between the classification results and 3 main metrics

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	-

BBN Results

The HUGIN System is adopted[48]. It 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 enables the user to define both discrete nodes and to some extent continuous nodes in the models.

Bayesian networks are often used to model domains, which are characterized by inherent uncertainty. This uncertainty may be caused by 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 behavior of the whole system. We have developed a prototype to show the potential of one of the quality prediction models, namely BBN, and illustrated its useful properties using real metrics data from the software engineering experiment(see Section1.6.2).

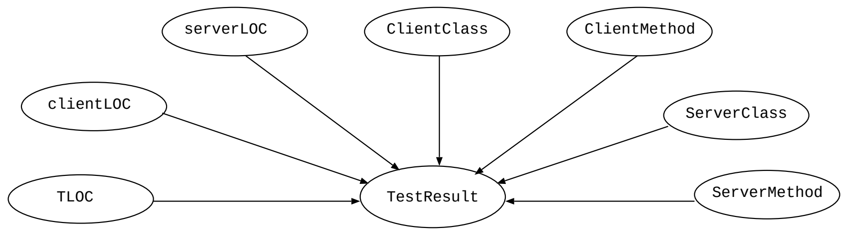
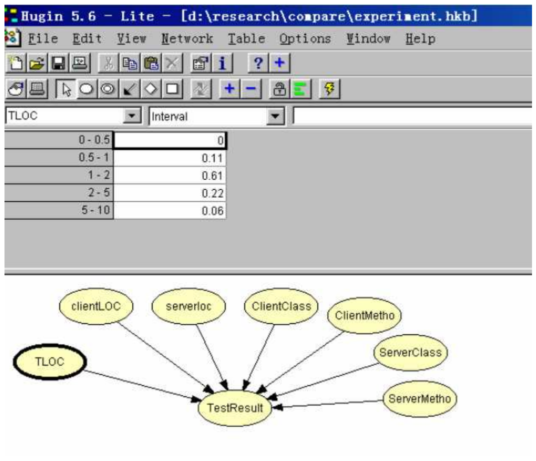


Fig. 1.17. The Influence Diagram of the BBN model

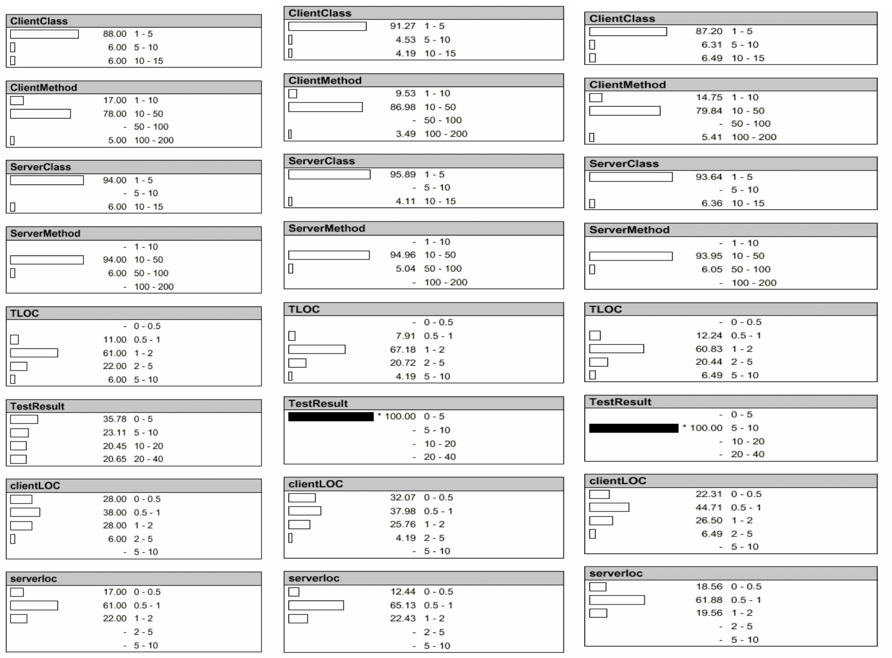
We constructed an influence diagram for the CORBA programs according to the metrics and testing results collected in the testing procedure, as shown in Figure 1.17. Although, due to interaction between these metrics, some of them are redundant. We assumed the worst scenario and considered every metrics. Each of these metrics shown in Figure 1.17 had its own impact on the testing result. Once the influence diagram is constructed, we input the


Hugin 5.6 - Lite - [d:\research\compare\experiment.hkb]
File Edit View Network Table Options Window Help
TLOC
Interval
clientLOC
serverloc
ClientClass
ClientMetho
ServerClass
ServerMetho
TLOC
TestResult

0 - 0.5	0
0.5 - 1	0.11
1 - 2	0.61
2 - 5	0.22
5 - 10	0.06

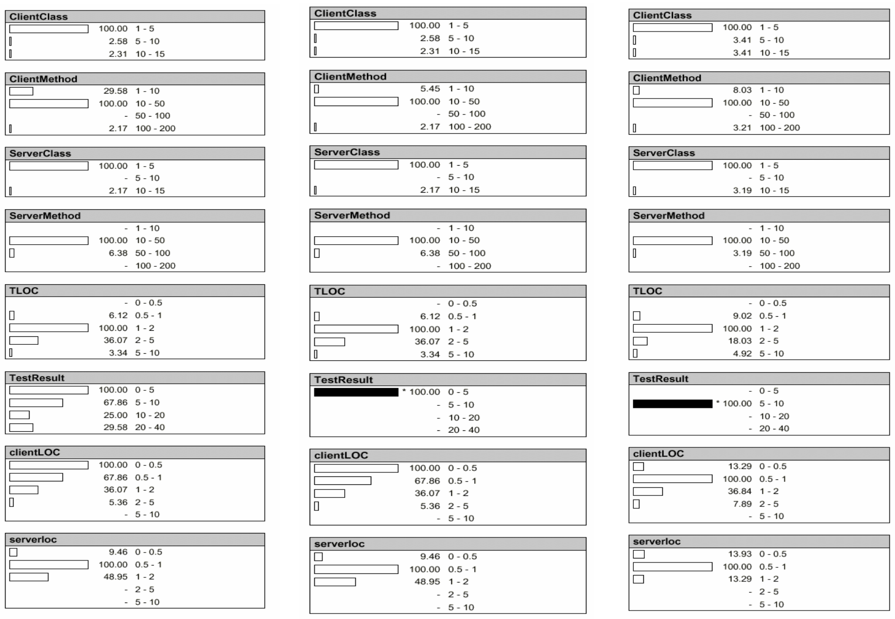
Fig. 1.18. The probability description of nodes in BBN model

probability of the metrics and testing results collected in our test procedures, as shown in Figure 1.18.



Node	State	Left	Middle	Right
ClientClass	1 - 5	88.00	91.27	87.20
	5 - 10	6.00	4.53	6.31
	10 - 15	6.00	4.19	6.49
ClientMethod	1 - 10	17.00	9.53	14.75
	10 - 50	78.00	86.98	79.84
	50 - 100	-	-	-
	100 - 200	5.00	3.49	5.41
ServerClass	1 - 5	94.00	95.89	93.64
	5 - 10	-	-	-
	10 - 15	6.00	4.11	6.36
ServerMethod	1 - 10	-	-	-
	10 - 50	94.00	94.96	93.95
	50 - 100	6.00	5.04	6.05
	100 - 200	-	-	-
TLOC	0 - 0.5	-	-	-
	0.5 - 1	11.00	7.91	12.24
	1 - 2	61.00	67.18	60.83
	2 - 5	22.00	20.72	20.44
	5 - 10	6.00	4.19	6.49
TestResult	0 - 5	35.78	* 100.00	-
	5 - 10	23.11	-	* 100.00
	10 - 20	20.45	-	-
	20 - 40	20.65	-	-
clientLOC	0 - 0.5	28.00	32.07	22.31
	0.5 - 1	38.00	37.98	44.71
	1 - 2	28.00	25.76	26.50
	2 - 5	6.00	4.19	6.49
	5 - 10	-	-	-
serverloc	0 - 0.5	17.00	12.44	18.56
	0.5 - 1	61.00	65.13	61.88
	1 - 2	22.00	22.43	19.56
	2 - 5	-	-	-
	5 - 10	-	-	-

Fig. 1.19. The different probability distribution of metrics according to the quality indicator (sum propagation)



ClientClass			
1 - 5	100.00	100.00	100.00
5 - 10	2.58	2.58	3.41
10 - 15	2.31	2.31	3.41

ClientMethod			
1 - 10	29.58	5.45	8.03
10 - 50	100.00	100.00	100.00
50 - 100	-	-	-
100 - 200	2.17	2.17	3.21

ServerClass			
1 - 5	100.00	100.00	100.00
5 - 10	-	-	-
10 - 15	2.17	2.17	3.19

ServerMethod			
1 - 10	-	-	-
10 - 50	100.00	100.00	100.00
50 - 100	6.38	6.38	3.19
100 - 200	-	-	-

TLOC			
0 - 0.5	-	-	-
0.5 - 1	6.12	6.12	9.02
1 - 2	100.00	100.00	100.00
2 - 5	36.07	36.07	18.03
5 - 10	3.34	3.34	4.92

TestResult			
0 - 5	100.00	* 100.00	-
5 - 10	67.86	-	* 100.00
10 - 20	25.00	-	-
20 - 40	29.58	-	-

clientLOC			
0 - 0.5	100.00	100.00	13.29
0.5 - 1	67.86	67.86	100.00
1 - 2	36.07	36.07	36.84
2 - 5	5.36	5.36	7.89
5 - 10	-	-	-

serverloc			
0 - 0.5	9.46	9.46	13.93
0.5 - 1	100.00	100.00	100.00
1 - 2	48.95	48.95	13.29
2 - 5	-	-	-
5 - 10	-	-	-

Fig. 1.20. The different probability distribution of metrics according to the quality indicator (max propagation)

The result of the Hugin tool are shown in Figure 1.19 and Figure 1.20, 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 1.19 shows the results of summation propagation, and Figure 1.20 the results of maximum propagation.

Summation propagation shows the true probability of the states of the nodes with the total summation equals to 1. For maximum propagation, if a state of a node belongs to the most 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 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 maximum 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 the most likely combination of states. From Figure 1.20(b), we can find the best combination of the metrics with respect to the corresponding testing results, as listed in Table 1.11. 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 1.10.

Table 1.11. Relationship between test result and metrics in BBN

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	1-2K	0.5-1K	0.5-1K

Case-Based Reasoning Model

To use Case-Based Reasoning model, a case base, containing a number of components with various metrics values and quality levels, should be established. When a new component is developed, the most similar component in the case base should be identified based on different metrics. The quality data of the case is then used for the new component. Case base for CBS is unavailable at present. Thus, we simply illustrate how CBR model works with our own synthetic data set.

Table 1.12. Result of Case-Based Reasoning Model

Team	Distance with P2	Fail	Maybe	R	R1
P3	914.7185	3	6	0.84	095
P4	470.6442	3	12	0.74	0.95
P5	2106.7950	2	1	0.95	0.96
P6	464.5589	13	10	0.60	0.77
P7	1992.6031	3	14	0.70	0.95
P8	490.4284	1	6	0.88	0.98
P9	1219.3470	4	2	0.89	0.93
P10	421.2268	2	2	0.93	0.96
P11	720.9598	6	3	0.84	0.89
P12	6114.3718	1	4	0.91	0.98
P13	1835.0995	17	19	0.37	0.70
P14	1087.2116	4	6	0.82	0.93
P15	514.7980	2	5	0.88	0.96
P16	672.7332	30	0	0.47	0.47
P17	392.1632	3	3	0.89	0.95
P18	750.7696	4	9	0.77	0.93
P19	949.3340	35	1	0.37	0.39

Assuming we have already had a case base containing 17 programs, i.e., P3 to P19. To predict the quality of a new program P2, we would find the most similar program in the case base (e.g., using Euclidean distance without weighting), see Table 1.12. We would then predict that program P2 had the similar quality level to that of the selected program, e.g., P17 with 3 faults under a reliability indicator of 89

1.6.4 Discussion

In our experiment, we used real CORBA programs as the testing data and applied them to the five quality prediction models to show the way they work. The effectiveness and applicability of these models could be evaluated using more data. Summation model and product model are the simplest compared with the other three models. They are intuitive and easy to construct. However, their prediction accuracy are not high. The meaning of these models is yet unclear. For this reason, they are not widely used.

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. 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.

BBN constructs an influence diagram depicting the dependency relationship of the metrics and testing result. It can predict a range of testing results using different combination of metrics. Also, it can suggest the best combination of metrics. This is more clear in BBN than in classification tree. The obvious disadvantage of BBN model is that the user is required to know the dependency relationship well in his specific domain before an effective influence diagram could be constructed. But such knowledge is only available after several runs.

Case-Based Reasoning model requires an established and sizable case base. Due to the lack of such archival data, the effectiveness of the CBR model for CBSD awaits further investigation.

The testing data used in our experiment is limited, i.e., only 18 programs were used to construct the models and to validate the prediction. To make the comparison more accurate, we will use more programs as test data in our future work. Also, if we could collect data from real component-based systems, we would apply these models to individual components as well as the entire systems in order to obtain the relationship of their qualities.

1.7 Conclusion

In this chapter, we introduce a component-based software development framework. We propose a QA model for component-based software development, which covers both the component QA and the system QA as well as their interactions. As far as we know, this is the first effort to formulate a QA model for developing software systems based on component technologies. We further propose a generic quality assessment environment for component-based software systems: ComPARE. ComPARE is new in that it collects metrics of more aspects for software systems, including process metrics, static code metrics, and dynamic metrics for software components, integrates reliability assessment models from different techniques used in current quality prediction area, and validates these models against the failure data collected in real life. ComPARE can be used to assess real-life off-the-shelf components and to evaluate and validate the models selected for their evaluation. The overall component-based software system can then be composed and analyzed seamlessly. ComPARE can be an effective environment to promote component-based software system construction with higher reliability evaluation and proper quality assurance.

Acknowledgement

The work described in this book chapter was supported by the following projects:

- "Open Component Foundation," an Industry Support Fund project supported by the Hong Kong Industry Department (Project No. AF94/99).
- a grant from the Research Grants Council of the Hong Kong Special Administrative Region (Project No. CUHK4360/02E).
- a strategic grant supported by the Chinese University of Hong Kong (Project No. 4410001).

References

1. P. Vitharana, “Risks and challenges of component-based software development,” *Communications of the ACM*, vol. 46, no. 8, pp. 67–72, 2003.
2. G. Pour, “Component-based software development approach: New opportunities and challenges,” in *Proceedings of Technology of Object-Oriented Languages Tools 26*, Santa Barbara, California, Aug. 1998, pp. 375–383.
3. T. Ravichandran and M. A. Rothenberger, “Software reuse strategies and component markets,” *Communications of the ACM*, vol. 46, no. 8, pp. 109–114, 2003.
4. A. W. Brown and K. C. Wallnau, “The current state of cbse,” *IEEE Software*, vol. 15, no. 5, pp. 37–46, 1998.
5. S. S. Yau and B. Xia, “Object-oriented distributed component software development based on corba,” in *Proceedings of COMPSAC’98*, Vienna, Austria, Aug. 1998, pp. 246–251.

6. G. Pour, "Enterprise javabeans, javabeans & xml expanding the possibilities for web-based enterprise application development," in *Proceedings of Technology of Object-Oriented Languages and Systems*, Nancy, France, June 1999, pp. 282–291.
7. M. L. Griss, "Software reuse architecture, process, and organization for business success," in *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering*, Dan Accadia, Herzliya, June 1997, pp. 86–98.
8. G. T. Heineman and W. T. C. (ed.), *Component-Based Software Engineering: Putting the Pieces Together.* Reading, MA: Addison-Wesley, 2001.
9. (2000) The ibm website. [Online]. Available: http://www4.ibm.com/software/ad/sanfrancisco
10. C. H. Schmauch, *ISO9000 for Software Developers.* Milwaukee, Wisconsin: ASQC Quality Press, 1994.
11. A. A. Keshlaf and K. Hashim, "A model and prototype tool to manage software risks," in *Proceedings of the First Asia-Pacific Conference on Quality Software*, Kowloon, Hong Kong, Oct. 2000, pp. 297–305.
12. M. R. Lyu, Ed., *Handbook of Software Reliability Engineering.* New York: McGraw-Hill, 1996.
13. J. Voas and J. Payne, "Dependability certification of software components," *The Journal of Systems and Software*, vol. 52, no. 2-3, pp. 165–172, 2000.
14. N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, 2000.
15. S. S. Gokhale and M. R. Lyu, "Regression tree modeling for the prediction of software quality," in *Proceedings of the Third ISSAT International Conference on Reliability and Quality in Design*, Anaheim, California, Mar. 1997, pp. 31–36.
16. J. Munson and T. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 423–433, 1992.
17. A. A. Porter and R. W. Selby, "Empirically guided software development using metric-based classification trees," *IEEE Software*, vol. 7, no. 2, pp. 46–53, 1990.
18. 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*, vol. 19, no. 11, pp. 1028–1034, 1993.
19. N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, 1999.
20. N. G. K. E. Emam, S. Benlarbi and S. N. Rai, "Comparing case-based reasoning classifiers for predicting high risk software components," *The Journal of Systems and Software*, vol. 55, no. 3, pp. 301–320, 2001.
21. M. R. Lyu, J. S. Yu, E. Keramidas, and S. R. Dalal, "Armor: Analyzer for reducing module operational risk," in *Proceedings of Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS-25)*, Pasadena, California, June 1995, pp. 137–142.
22. D. J. Smith, *Achieving Quality Software (Third Edition).* Chapman & Hall, 1995.
23. W. Kozaczynski and G. Booch, "Component-based software engineering," *IEEE Software*, vol. 15, no. 5, pp. 34–36, 1998.
24. (2000) The omg website. [Online]. Available: OMG:http://www.omg.org/corba/whatiscorba.html
25. (2000) The microsoft website. [Online]. Available: http://www.microsoft.com/isapi

26. Y. M. Wang, O. P. Damani, and W. J. Lee, “Reliability and availability issues in distributed component object model (DCOM),” in *Fourth International Workshop on Community Networking Proceedings*, Atlanta,Georgia, Sept. 1997, pp. 59–63.
27. (2000) The sun website. [Online]. Available: http://developer.java.sun.com/developer
28. G. Pour, M. Griss, and J. Favaro, “Making the transition to component-based enterprise software development: Overcoming the obstacles - patterns for success,” in *Proceedings of Technology of Object-Oriented Languages and systems*, Nancy, France, June 1999, pp. 419–419.
29. C. Szyperski, *Component Software: Beyond Object-Oriented Programming.* New York: Addison-Wesley, 1998.
30. S. M. Yacoub, B. Cukic, and H. H. Ammar, “A component-based approach to reliability analysis of distributed systems,” in *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, Lausanne, Switzerland, Oct. 1999, pp. 158–167.
31. ——, “Scenario-based reliability analysis of component-based software,” in *Proceedings of 10th International Symposium on Software Reliability Engineering*, Boca Raton,Florida, Nov. 1999, pp. 22–31.
32. J. Q. Ning, K. Miriyala, and W. Kozaczynski, “An architecture-driven, business-specific, and component-based approach to software engineering,” in *Proceedings of Third International Conference on Software Reuse: Advances in Software Reusability*, Rio De Janeiro, Brazil, Nov. 1994, pp. 84–93.
33. M. goulao and F. B. e Abreu, “The quest for software components quality,” in *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC’02)*, Oxford,England, Aug. 2002, pp. 313–318.
34. Y. Yu and B. W. Johnson, “A bbn approach to certifying the reliability of cots software systems,” in *Proceedings of Annual Reliability and Maintainability Symposium*, Tampa, Florida, Jan. 2003, pp. 19–24.
35. C. Rajaraman and M. R. Lyu, “Reliability and maintainability related software coupling metrics in C++ programs,” in *Proceedings 3rd IEEE International Symposium on Software Reliability Engineering (ISSRE’92)*, North Carolina, USA, Oct. 1992, pp. 303–311.
36. ——, “Some coupling measures for C++ programs,” in *Proceedings of TOOLS USA 92 Conference*, Santa Barbara, California, Aug. 1992, pp. 225–234.
37. I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach.* Reading, MA: Addison-Wesley, 1992.
38. M. R. Lyu, “Software reliability theory,” in *Encyclopedia of Software Engineering*, J. J. Marciniak, Ed. New York: Wiley, 2001.
39. F. Stallinger, A. Dorling, T. Rout, B. Henderson-Sellers, and B. Lefever, “Software process improvement for component-based software engineering: An introduction to the oospice project,” in *Proceedings of the 28th EUROMICRO Conference (EUROMICRO’02)*, Dortmund, Germany, Sept. 2002, pp. 318–323.
40. (2000) Hong kong productivity council. [Online]. Available: http://www.hkpc.org/itd/servic11.htm
41. S. Beydeda and V. Gruhn, “Merging components and testing tools: The self-testing cots components (stecc) strategy,” in *Proceedings of the 29th EUROMICRO Conference (EUROMICRO’03)*, Belek-Antalya, Turkey, Sept. 2003, pp. 107–114.

42. (2001) The metamata website. [Online]. Available: http://www.metamata.com
43. (2001) The klgroup website. [Online]. Available: http://www.klgroup.com
44. S. H. Kan, *Metrics and Models in Software Quality Engineering (Second Edition).* Reading, MA: Addison-Wesley, 2003.
45. S. Sedigh-Ali, A. Ghafoor, and R. A. Paul, "Metrics and models for cost and quality of component-based software," in *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, Hokkaido, Japan, May 2003, pp. 149–155.
46. T. Systa, Y. Ping, and H. Muller, "Analyzing java software by combining metrics and program visualization," in *Proceedings of the Fourth European Software Maintenance and Reengineering*, Zurich, Switzerland, Mar. 2000, pp. 199–208.
47. J. D. Musa, *Software Reliability Engineering.* New York: McGraw-Hill, 1998.
48. (2001) The hugin expert website. [Online]. Available: http://www.hugin.com
49. G. Xing and M. R. Lyu, "Testing, reliability, and interoperability issues in the corba programming paradigm," in *Proceedings of 1999 Asia-Pacific Software Engineering Conference (APSEC'99)*, Takamatsu, Japan, Dec. 1999, pp. 530–536.