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, The Chinese University of Hong Kong, Hong Kong, China[1]
Dept. of System Engineering and Engineering Management, The Chinese University of Hong Kong, Hong Kong, China[2]

{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 chapter.

1 Introduction

Based on the component-based software development (CBSD) approach [403], software systems are developed using a well defined software architecture and off-the-shelf components as building blocks [335]. This 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 [342]. Typically, COTS components are available from a component repository; users select the appropriate ones and integrate them to establish the target software system (see Fig. 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

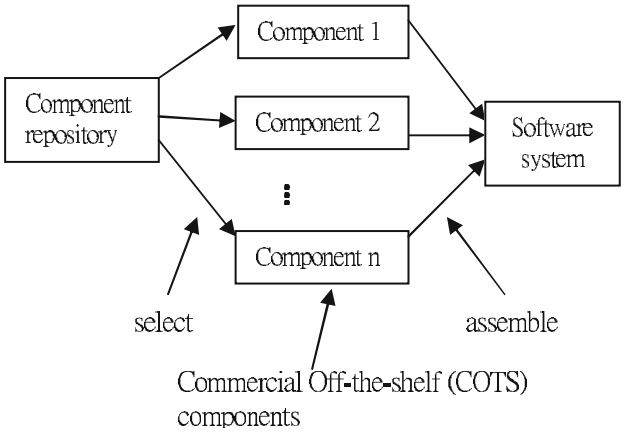

Fig. 1. Component-based software development.

in the context of a well defined architecture; and 3) it communicates with other components by its interfaces [47]. Current component technologies have been used to implement different software systems, such as object-oriented distributed component software [431] and Web-based enterprise applications [336].

The system architecture of a component-based software system is layered and modular [152, 176, 192]; see Fig. 2.

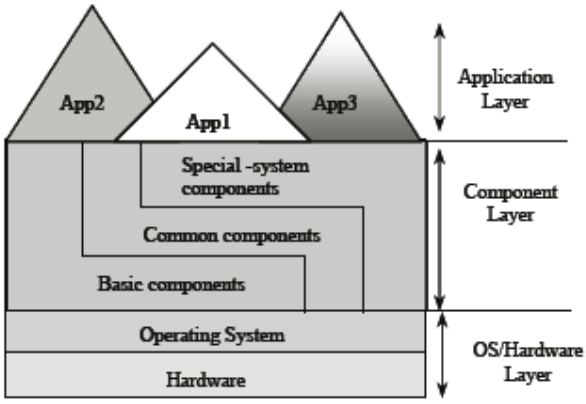

Fig. 2. System architecture of component-based software systems.

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 cross-system middleware components and includes software and interfaces common 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 involve the operating and hardware systems.

A CBSD-based software system is composed of one or more components that may be procured off-the-shelf, produced in-house, or developed through 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 in 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 [360]. Process metrics (e.g., reliability estimates) [224], static code metrics (e.g., code complexity) [251], and dynamic metrics (e.g., test thoroughness) [411] are widely used to predict the quality of software components at different development phases [122, 360].

Several techniques are used to model the predictive relationship between different software metrics and for component classification, i.e., for classifying software components into fault-prone and non fault-prone categories [144]. These techniques include discriminant analysis [297], classification trees [334], pattern recognition [45], Bayesian network [121], case-based reasoning (CBR) [216], and regression tree models [144]. There are also prototypes and tools [224, 253] which use such techniques to automate software quality prediction. However, these tools employ only one type of 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 quality of individual off-the-shelf components and the overall quality of software systems. We integrate different prediction techniques and different software metric categories to form a single environment, and investigate their effectiveness on quality prediction of components and CBS.

The rest of this chapter is organized as follows: we first give an overview of state-of-the-art CBSD techniques in Sect. 2, and highlight the quality assurance (QA) issues behind them in Sect. 3. Section 4 proposes a QA model which is designed for quality management in CBSD process. In Sect. 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 Sect. 6, the pros and cons of these prediction models are analyzed. Finally, Sect. 7 concludes this chapter.

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 [368]. 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 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++, and Java, and 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 between components [47]. Among the component infrastructure technologies that have been developed, there are three de facto industrial standards: OMG's CORBA, Microsoft Corporation's Component Object Model (COM) and Distributed COM (DCOM), and Sun Microsystem's JavaBeans and Enterprise JavaBeans [234].

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 vendors and object technology user companies [310]. CORBA manages details of component interoperability, and allows applications to communicate with one another despite their different locations and designs. Interfaces are the only way in 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 a 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 that 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 [431], including component-based software systems, because it offers a consistent distributed programming and runtime environment for common programming languages, operating systems, and distributed networks.

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

Component Object Model (COM) is a general architecture for component software [284]. It supports Windows- and Windows NT-based platform-dependent 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 that components and their clients must follow to ensure dynamic interoperability. This enables online software update and cross-language software reuse [417].

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 physical connections.

2.3 Sun Microsystems's JavaBeans and Enterprise JavaBeans

Sun Microsystem'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 [381].

The 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 multi-vendor servers; 2) propagation of transaction and security contexts; 3) multilingual clients; and 4) supporting ActiveX via DCOM/CORBA bridges.

JavaBeans and EJBs extend the native strength of Java incorporating portability and security into 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.

2.4 Comparison among Different Architectures

Comparisons between development technologies for component-based software systems can be found in [47, 337, 385]. Table 1 summarizes their different features.

Table 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

3 Quality Assurance for Component-Based Software Systems

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 [335]: 1) Requirements analysis; 2) Software architecture selection, construction, analysis, and evaluation; 3) Component identification and customization; 4) System integration; 5) System testing; and 6) 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 char-

acteristics of component systems are not accounted for. Although some QA techniques, such as the reliability analysis model for distributed software systems [429, 430] and the component-based approach to Software Engineering [308], have been studied, there are still no clear and well defined standards or guidelines for CBS. The identification of the QA characteristics, along with the models, tools, and metrics, have urgent need for standardization.

3.2 Quality Characteristics of Components

QA technologies for component-based software development have to address two inseparable questions: 1) How do we ensure the quality of a component? and 2) How do we 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 [150, 433]. Here, we propose a list of component features for the assessment: 1) Functionality; 2) Interface; 3) Usability; 4) Testability; 5) Maintainability; and 6) Reliability.

Software metrics can be proposed to measure software complexity [339, 340]. Such metrics are often used to classify components [211]. 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. It is not cost-effective to modularize a component that is too trivial. But, on the other hand, for a component that is too complex, it is hard to ensure high quality.

3) Reuse frequency. The number of times and different domains in which a component has been used previously is an indicator of its usefulness.

4) Reliability. This is the probability of failure-free operations of a component under certain operational scenarios [252].

4 A Quality Assurance Model for Component-Based Software Systems

Since component-based software systems are developed on an underlying process different from that for traditional software, their quality assurance model should address both the process of componentization and the process of the overall system development. Figure 3 illustrates this view.

Many standards and guidelines, such as ISO9001 and CMM model [376], are used to control the quality activities of a traditional software development

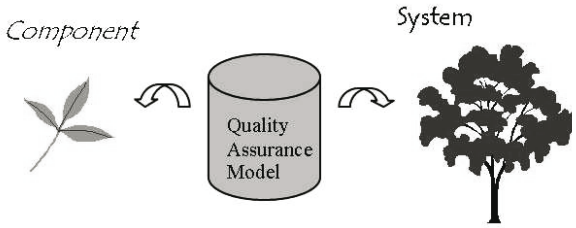

Fig. 3. Quality assurance model for both components and systems.

process. In particular, Hong Kong Productivity Council has developed the HKSQA model to localize the general SQA models [184]. 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.

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 that a component should realize, as well as the programming language, platform, and interfaces related to the component.

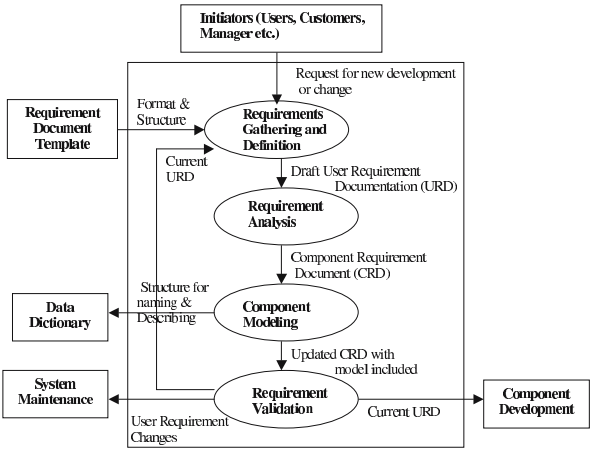

Fig. 4. Component requirement analysis process overview.

The component requirement process overview diagram is shown in Fig. 4. Initiated by the users or customers for new development or for 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.

4.2 Component Development

Component development is the process of implementing the requirements for a well functioning, high quality component with multiple interfaces. The objective of component development is the development of the final component products, their interfaces, and their 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.

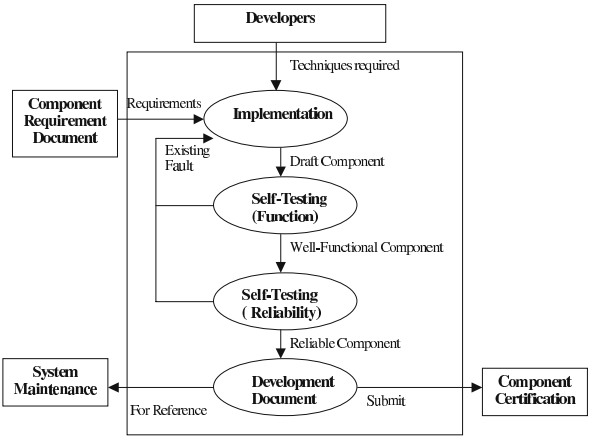

Fig. 5. Component development process overview.

The component development process overview diagram is shown in Fig. 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.

4.3 Component Certification

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

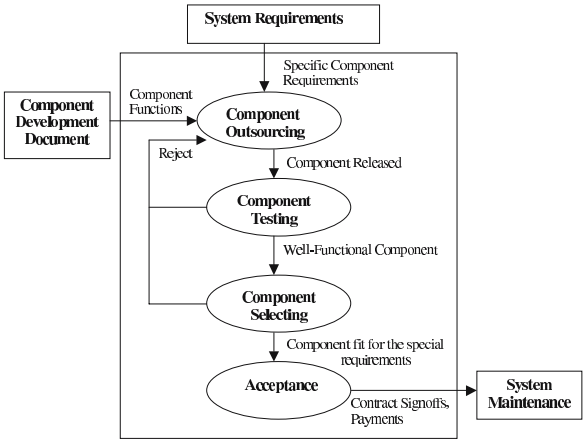

Fig. 6. Component certification process overview.

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 supervised 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 Fig. 6. The inputs to this phase are the component development documents, and the output is the testing documentation for system maintenance.

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 local platforms; and 3) upgrading the specific component to get better performance or 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 well with other components.

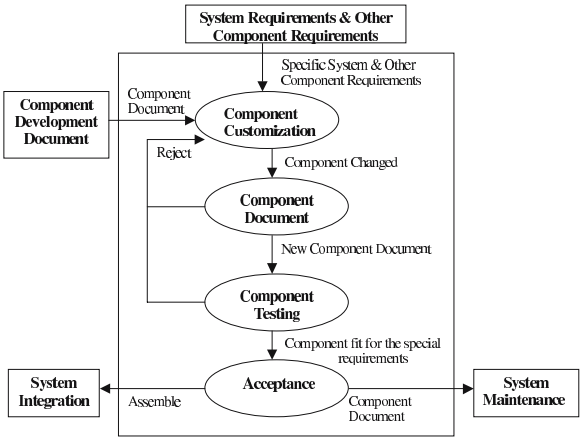

Fig. 7. Component customization process overview.

All components must be customized according to the operational system requirements or the interface requirements. The component customization process overview diagram is shown in Fig. 7. The inputs to component customization are the system requirements, the component requirements, and the component development documents. The outputs are the customized components, and documents for system integration and system maintenance.

4.5 System Architecture Design

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

System architecture design should compare the pros and cons of different system architectures and select the one most suitable for the target CBS. The process overview diagram is shown in Fig. 8. This phase consists of system requirement gathering, analysis, system architecture design, and system specification. The output of this phase comprises the system specification document for system integration, and the system requirements for the system testing and system maintenance phases.

4.6 System Integration

System integration is the process of properly assembling the components selected to produce the target CBS under the system architecture designed. The process overview diagram is shown in Fig. 9. The inputs are the system

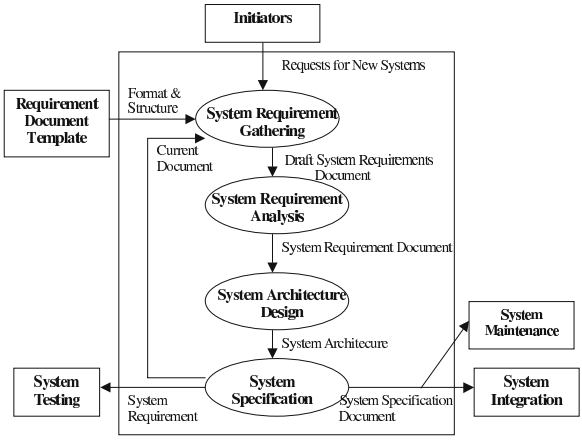

Fig. 8. System architecture design process overview.

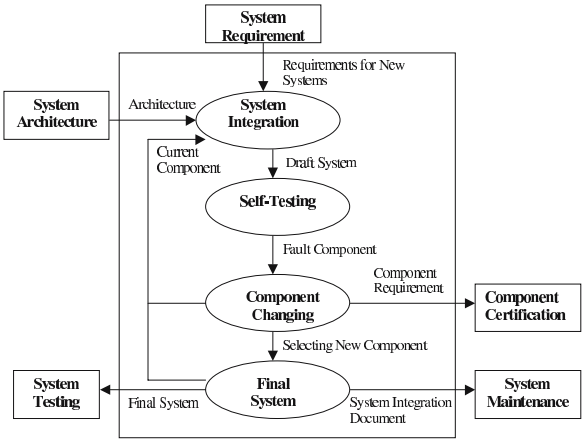

Fig. 9. System integration process overview.

requirement documentation and the specific architecture. There are four steps in this phase: integration, testing, changing components, and reintegration (if necessary). At the end of this phase, the final target system will be ready for system testing, with the appropriate document for the system maintenance phase.

4.7 System Testing

System testing is the process of evaluating a system to: 1) confirm that the system satisfies the specified requirements; and 2) identify and correct defects. System testing includes function testing and reliability testing. The process overview diagram is shown in Fig. 10. This phase consists of selecting a testing

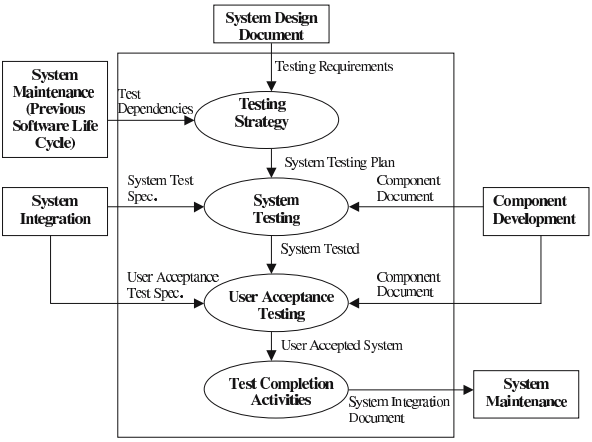

Fig. 10. System testing process overview.

strategy, system testing, user acceptance testing, and completion activities. The input comprises 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 to interaction testing between multiple components, and includes coordination issues and deadlocks.

4.8 System Maintenance

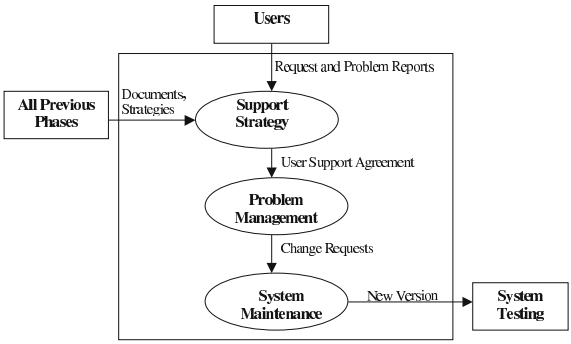

Fig. 11. System maintenance process overview.

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 Fig. 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.

5 A Generic Quality Assessment Environment for Component-Based Systems: ComPARE

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 faulty data collected in the development process. Different from other existing tools [253], 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, and inheritance metrics) that are adopted from object-oriented software engineering, and provides different estimation models for overall system assessment.

5.1 Overall Architecture

A number of commercial tools are available for the measurement of software metrics for object-oriented programming. There are also off-the-shelf tools for testing and debugging of software components [36]. However, few tools can measure the static and dynamic metrics of software systems, perform various types of 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 Fig. 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 the BBN model), the users can validate the selected models with previous failure data collections. If the users are not satisfied with the prediction result,

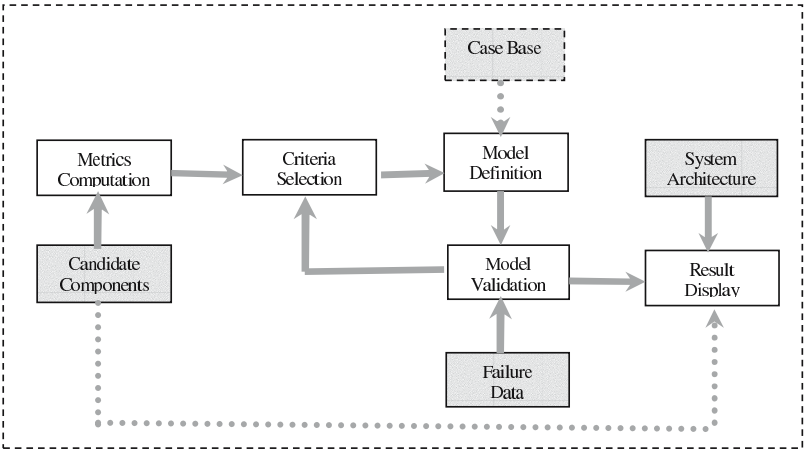

Fig. 12. Architecture of ComPARE.

they can go back to the previous step, redefine 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.

The objectives of ComPARE are summarized as follows:

1) To predict the overall quality by using process metrics and static code metrics, as well as dynamic metrics. In addition to complexity metrics, we use process metrics, cohesion metrics and inheritance metrics, as well as dynamic metrics (such as code coverage and call graph metrics) as inputs 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 results of different models. ComPARE integrates several existing quality models into one environment. In addition to selecting or defining these different models, the 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, the user can select from several quality prediction models and select the one most suitable for the prediction task at hand. Moreover, the user can also define his or her 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 real failure data. The user can redefine his or her 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 user 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 [274] and Jprobe [229] suites to measure the different metrics for the candidate components. These two tools, involving metrics, audits, and debugging, as well as code coverage, memory, and deadlock detected, are commercially available.

5.2 Metrics Used in ComPARE

Table 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 [218,362]. The process metrics we selected are listed in Table 2 [224]. 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 3 [251,274,384,411]. The dynamic metrics measure component features when they are executed. Table 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 [274] and Jprobe Testing Suite [229]. We adopt these metrics in ComPARE.

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 [144]. These techniques include discriminant analysis [297], classification trees [334], pattern recognition [45], Bayesian network [121], case-based reasoning (CBR) [216], and regression tree model [224].

Table 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, $$LCOM = (P - Q) \quad if \quad P > Q$$ $$= 0 \qquad \text{otherwise}$$

Table 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.

Up to now, there is no good quality prediction model for CBS. Here, we set some evaluation criteria for good quality prediction models [298]: 1) Useful quantities, i.e., the model can make predictions of quantities reflecting software quality; 2) Prediction accuracy, i.e., the model can make predictions of quality which can be accurately observed later; 3) Ease of measuring parameters, i.e., the parameters in the model are easily measured or simulated; 4) Quality of assumptions, i.e., the assumptions should be reasonable, rather than too narrow or limited; 5) Applicability, i.e., the model should be widely used in various projects or experiments; and 6) Simplicity, i.e., 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 with the benchmarks. The concept of the summation model is formulated as follows:

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

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:

$$Q = \prod_{i=1}^{n} m_i, \tag{2}$$

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

Classification Tree Model

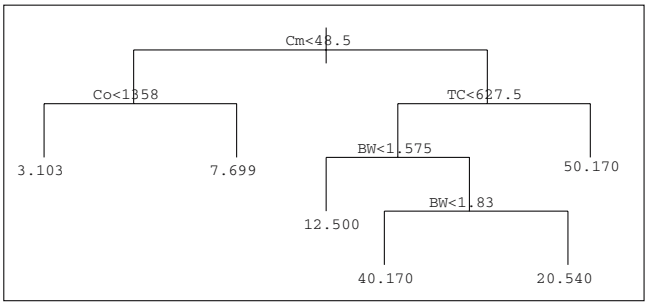

Fig. 13. An example of the classification tree model.

Classification tree model [334] classifies candidate components into different quality categories by constructing a tree structure. All candidate components (with a certain failure rate) form the leaves of the tree. Each node of the tree represents a metric (or a composed metric calculated from other metrics) with a certain value. All children of the left sub-tree of a node represent those components whose values of the same metrics are smaller than the value of the node. Similarly, all children of the right sub-tree of a node are those components whose values of the same metric are equal to or larger than the value of the node. Figure 13 gives an example of the classification tree model.

In ComPARE, a user can define the metrics and their values 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 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 bandwidth metric) are sample metrics [144]. At each node of the tree are metrics and values, and the leaves represent the components with a 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 [216]. 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 with any combination of parameters in CBR. For this reason, we adopt the simplest CBR classifier modeling with Euclidean distance, z-score standardization [216], and without a weighting scheme. Finally, we select the single, nearest neighbor for prediction.

Bayesian Network Model

Bayesian networks (also known as Bayesian Belief Networks, or BBNs) is a graphical network that represents probabilistic relationships among variables [121]. 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 of, as well as a sound mathematical basis for Bayesian probability [118]. The foundation of Bayesian networks is based on the following theorem, which is known as Bayes' Lemma:

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

where H, E, and c are independent events and P is the probability of such an 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 [121]. Users can also define their own BBN models in ComPARE, and compare the results with other models.

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 5.2.

Criteria Selection and Weighting

After computing different metrics, the users will select and weigh the criteria associated with 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 that affects the quality of the candidate component.

Model Validation

Model validation enables comparison between different models with respect to actual software failure data. It helps users compare different results based on a subset of the software failure data chosen 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.

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 architectures of component-based software development, namely, CORBA, DCOM, and Java/RMI.

Figures 14 and 15 show screen dumps of the ComPARE prototype. The computation of various metrics for software components and the application of

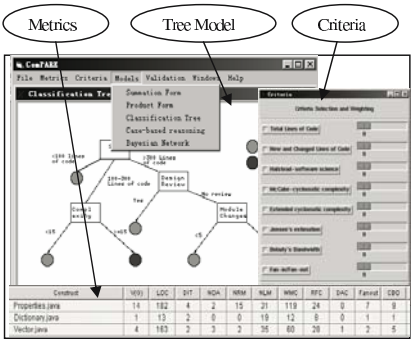

Fig. 14. GUI of ComPARE for metrics, criteria and tree model.

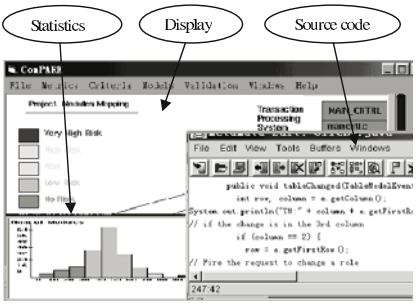

Fig. 15. GUI of ComPARE for prediction display, risky source code and result statistics.

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.

6 Experiment and Discussion

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 5.3 to predict and evaluate the relationship between the number of faults and the 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 [427]. We applied the selected prediction models to the final CORBA programs and investigated how well they behave. This information is useful to the users for determining the appropriate quality prediction models.

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 majoring in computer science. The duration of the project was four weeks. The programming teams (two or three students for each team) participating in this project were required to independently design and develop a distributed system. The system had to 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) and use different programming languages (Java or C++) for the client and 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 used VisiBroker and seven used Iona Orbix. For the 12 VisiBroker programs, nine used Java and two used C++ for both client and server implementations, and one 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 5. The meaning of the metrics and testing results are listed below:

- Total Lines of Code (TLOC): the total length of the 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 designed to raise exceptions that failed to work because the client-side of the program forbade it. In this situation, we were not sure whether the server was designed to properly 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, and $P_j = C - Fail - Maybe$.
- 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, $P_j = C - Fail - Maybe$; and M_j is the number of "Maybe" cases for program j.

Table 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

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 procedure below.

First of all, we collected the different metrics of all the programs. Metamata [274] and JProbe Suite [229] were used for this purpose. We designed test cases for these CORBA programs according to the specification. We used black-box testing methods, i.e., testing was on system functions only. Each operation defined in the system specification was tested. We defined some test cases for each operation. The test cases selected were from two categories: normal cases and cases that caused exceptions in the system. For each operation in the system, at least one normal test case was conducted in testing. In the other cases, all the exceptions were covered. But, in order to reduce the workload, we tried to use as few test cases as possible as long as all the exceptions had been accounted for.

We used the test results as indicators of quality. We applied different quality prediction models, i.e., the 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 during or after 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.

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 give equal weighting to all the metrics, e.g., the weights of all metrics equal 1. Also, we normalize the values of the metrics by using the ratio of the actual value to the maximum value of that particular metric, i.e., $m_1 = \frac{TLOC}{max(TLOC)}$, $m_2 = \frac{CLOC}{max(CLOC)}$, and so on, for every program. The overall quality mark, then, is $Q = m_1 + m_2 + \cdots$ for the 18 programs. The result of the summation model is listed in Table 6.

Product Model

The product model multiplies all the metrics selected and weighted by the user. The values of the metrics are also normalized to values close to 1, using the same method as above. The final result is the product of these normalized values. It is listed in Table 6.

Classification Tree Results Using CART

We adopted the commercial tool CART [118] 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 element of a CART analysis is 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).

We applied the metrics and testing results in Table 5 to the CART tool, and collected the classification tree results for predicting the quality variable "Fail". Table 7 is the option setting of the classification tree. The tree constructed is shown in Fig. 16, and the relative importance of each metric is listed in Table 8. From Fig. 16, we can see that the 18 learning samples are classified into nine groups (terminal nodes), whose information is listed in Table 9. 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 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 the three main metrics was analyzed, and the results are listed in Table 10.

Table 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 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 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			

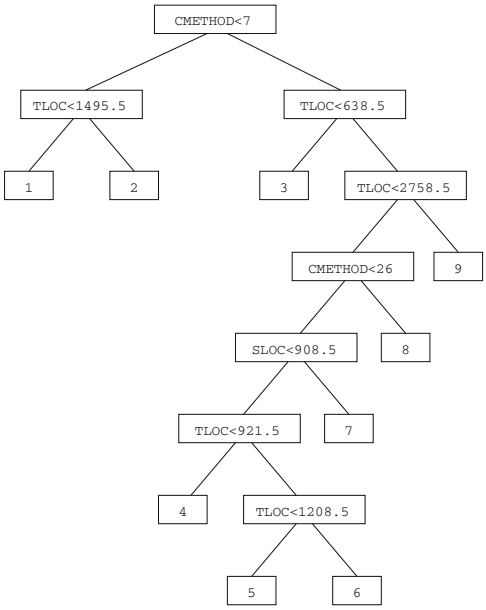

Fig. 16. Classification tree structure.

Table 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 10. Relationship between the classification results and 3 main metrics.

Terminal Node	Mean Faults	CMethod	TLOC	SLOC
4	2	7$\tilde{2}$6	638.5∼921.5	≤908.5
9	2	>7	≤638.5	-
6	3	7$\tilde{2}$6	1208.5∼2758.5	≤908.5
7	4	7$\tilde{2}$6	638.5∼921.5	>908.5
3	6	>7	≤638.5	-
5	7	7$\tilde{2}$6	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 was adopted [118]. 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, and 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 Section 6.2).

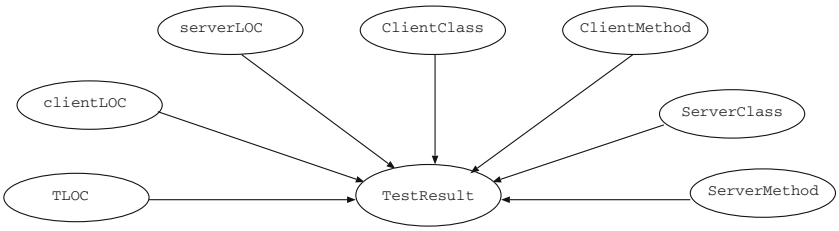

Fig. 17. The Influence Diagram of the BBN model.

We constructed an influence diagram for the CORBA programs according to the metrics and testing results collected in the testing procedure, as shown in Fig. 17. However, due to interactions between these metrics, some of the metrics are redundant. We assumed the worst scenario and considered every metrics. Each of these metrics shown in Fig. 17 had its own impact on the

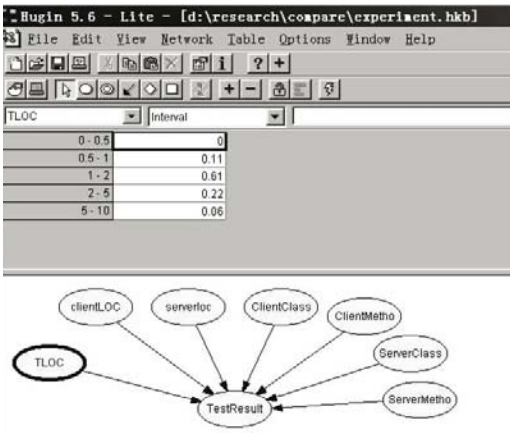

Fig. 18. The probability description of nodes in BBN model.

testing result. Once the influence diagram is constructed, we input the probability of the metrics and testing results collected in our test procedures, as shown in Fig. 18.

The result of the HUGIN tool are shown in Fig. 19 and Fig. 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; and (c) is the probability distribution of the metrics when the number of faults is between five and 10. Figure 19 shows the results of summation propagation, and Fig. 20 shows the results of maximum propagation.

Summation propagation shows the true probability of the states of the nodes with the total summation equal 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 value of the probability of the most probable configuration they found relative 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 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 evidence holds. In each node, a state having the value 100 belongs to the most likely combination of states. From Fig. 20(b), we can find the best combination of the metrics with respect to the corresponding testing results, as listed in Table 11. For test results between 0 and 5, the ranges of $CMethod$, $TLOC$, and $SLOC$ are very close to the results of the classification tree in Table 10.

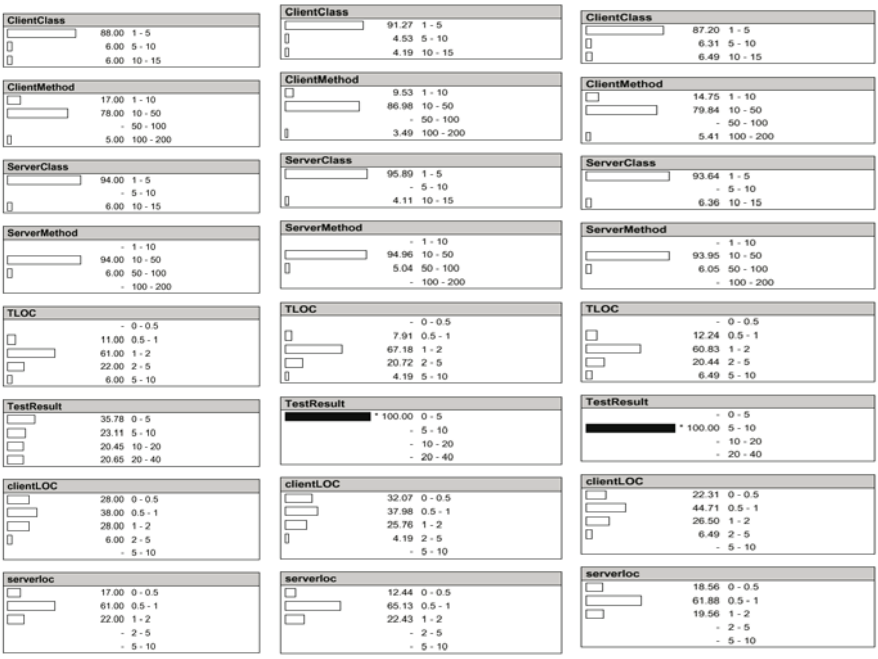

Fig. 19. The different probability distribution of metrics according to the quality indicator (sum propagation).

Table 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 metric values and quality levels should be established. When a new component is developed, the component most similar to it in the case base should be identified. The quality data of the case is then used for the new component. Case base is unavailable for CBS at present. Thus, we simply illustrate how the CBR model works with our own synthetic data set.

Assume we already have a case base containing 17 programs, i.e., P3 through P19. To predict the quality of a new program P2, we would find the most similar program in the case base (using, for example, Euclidean distance without weighting; see Table 12). We would then predict that program P2 had a quality level similar to that of the selected program, e.g., P17, with three faults, under a reliability indicator of 89%.

Fig. 20. The different probability distribution of metrics according to the quality indicator (max propagation).

Table 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

6.4 Discussion

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

The classification tree model predicts the quality of a program by constructing a tree model according to the metrics collected. If the learning sample is large enough, the prediction result of the 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 would 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 combinations of metrics. Also, it can suggest the best combination of metrics. This is more clear in BBN than in the classification tree. The obvious disadvantage of the BBN model is that the user is required to know well the dependency relationship in his or her specific domain before an effective influence diagram can be constructed. But such knowledge is available only after several runs.

The case-based reasoning model requires an established and sizable case base. Due to the lack of such 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 to entire systems in order to obtain a relationship of their qualities.

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 more metrics for software systems, including process metrics, static code metrics, and dynamic metrics for software components, integrates reliability assessment models from different techniques currently used in quality prediction field, and validates these models against real failure data. ComPARE can be used to assess live 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.

Acknowledgment

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).