Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 49 (2007) 122–141
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

A quality framework for developing and evaluating original software components

Andreas S. Andreou *, Marios Tziakouris

University of Cyprus, Department of Computer Science, 75 Kallipoleos str., CY1678 Nicosia, Cyprus

Received 18 April 2005; received in revised form 15 March 2006; accepted 22 March 2006
Available online 3 May 2006

Abstract

Component-based software development is being identified as the emerging method of developing complex applications consisting of heterogeneous systems. Although more research attention has been given to Commercial Off The Shelf (COTS) components, original software components are also widely used in the software industry. Original components are smaller in size, they have a narrower functional scope and they usually find more uses when it comes to specific and dedicated functions. Therefore, their need for interoperability is equal or greater, than that of COTS components. A quality framework for developing and evaluating original components is proposed in this paper, along with an application methodology that facilitates their evaluation. The framework is based on the ISO9126 quality model which is modified and refined so as to reflect better the notion of original components. The quality model introduced can be tailored according to the organization-reuser and the domain needs of the targeted component. The proposed framework is demonstrated and validated through real case examples, while its applicability is assessed and discussed.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Software components; Quality assessment; Reuse; Development

1. Introduction

Recently, component-based software design has emerged as a viable and economic alternative to the traditional software design process [27,20,26,34]. The ability to build complete system solutions by interconnecting through public interfaces independently created and deployed components, is the driving force behind the recent success of component-based software engineering (CBSE). The underlying heterogeneity of distributed computing, along with the problems emerged by the implementation of similar systems with conventional software development, gave rise to the need for component-based software development.

CBSE does not only solve the complexities of distributed computing, it also plays an increasing role in general software systems development [29]. Components are anticipated to reduce the cost and time to market of systems they are part of and increase their quality. In recent years there is a significant rise to the software engineering aspects of components production and usage. The reuse of software components has the ability to significantly reduce both software development costs, as well as the development life cycle. As a result, the time to market is also reduced mainly due to the fact that complexities involved in large heterogeneous systems are resolved by the use of ready-made components specialized for this task. Components provide a flexible way of shipping functionalities in a conveniently packed black-box fashion that makes distribution simpler. It is also believed that components, although they require some wrapping to use them, they are more flexible and simpler to be reused than other similar reuse approaches like design patterns. This argument finds a lot of supporters recently, especially in large-scale systems [1]. In addition, the usage of CBSE results to increased

* Corresponding author. Tel.: +35722892692; fax: +35722892701.
E-mail addresses: aandreou@ucy.ac.cy (A.S. Andreou), Marios.Tziakouris@CSE.com.cy (M. Tziakouris).

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.03.007

reliability of software systems. If the components have been reused in several occasions, they are likely to be more reliable than other software developed from scratch, as they were tested under a larger variety of conditions [8].

A major problem with CBSE is the quality of the components used in a system [14,12,5]. It is well known that the strength of a chain is equal to the strength of its weakest link. Similar to this concept, the reliability of a component-based software system depends on the reliability of the components that is made of. In CBSE, the proper search and selection process of components is considered the cornerstone for the development of any effective component-based system. So far the software industry was concentrated on the functional aspects of components, leaving aside the difficult task of assessing their quality. If the quality assurance of in-house developed software is a demanding task, doing it with software developed elsewhere, often without having access to its source code and detailed documentation, presents an even greater concern [13]. Introducing software components of unknown quality may have catastrophic results.

In terms of reuse, components can in general be divided into **original** components and components that come close to full software products (i.e., Commercial Off The-Shelf (COTS) systems – see also [14,31]), primarily due to their difference in scope. Broadly speaking, COTS represent larger systems and usually they are considered as complete software solutions. They can be thought as a family of original components, which aim to offer a more complete service to the user. For instance, Microsoft Excel can be considered as a COTS component in its wider sense. It consists of various other components, i.e., math functions, charting, data exports, and it represents a complete spreadsheet solution (Fig. 1). On the other hand, original components may be conceived as software units that are produced and marketed as single-task or limited-function-set building blocks of a larger system. Therefore, they have a narrower scope and their primary task is to provide functionality to perform

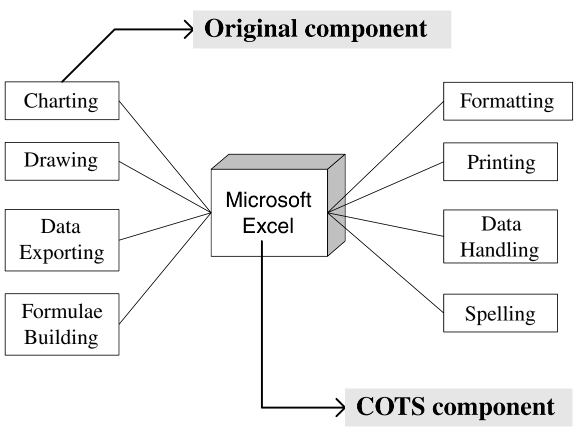

Fig. 1. COTS vs original software components.

specific actions in a repetitive manner. For instance, a component that provides certain charting capabilities can be considered as an original component. Original components can also be viewed as components that participate with other components to form software applications or other COTS.

This paper is an attempt to address the quality issues of *original software components*. Building upon the ISO9126 quality model for software systems, it provides a quality framework with which original components can be assessed and evaluated. It aims at representing a solid reference, which will be valuable not only to re-users but also to original software component developers.

The Quality of Service (QoS) of a component is an indication given on behalf of its owner, about its confidence to carry out the required services. The QoS offered by each component depends upon the computation it performs, the algorithm used, its expected computational effort, the required resources, the motivation of the developer and the dynamics of supply and demand [12]. However, evaluating the QoS of a component is not a straightforward task and it is not accidental the fact that till today there is not a wide-accepted model for evaluating the quality requirements of software components. Reasons include the fact that there is no general consensus on the quality characteristics that need to be considered and also the fact that software vendors do not provide much information about the quality attributes of their components. In addition to this, there are no metrics defined that could help to the evaluation of the quality attributes objectively.

Based on what mentioned, the need for the creation of a Quality of Service Framework is essential. Such a framework should contain detailed descriptions about QoS attributes of software components (see e.g., [12]), along with appropriate metrics, evaluation methodologies and interrelationships with other attributes. This QoS Framework for software components could prove to be a valuable tool for:

(i) the component developer by:
  - identifying areas of improvement that can result to potential competitive advantage
  - evaluating the QoS attributes of the component and comparing it with competitive products in a quantifiable manner
  - representing a solid reference for incorporating QoS attributes into the components being developed
  - enabling him to capitalize on potential advantages of the component by exposing not visible QoS attributes with quantifiable metrics using the QoS metrics

(ii) the software system developer (re-user) by:
  - providing him with a clear methodology for making comparisons of components with similar functionality based on widely accepted QoS metrics and ultimately selecting the most suitable component

- allowing him to define the QoS requirements of the components that are going to be incorporated into the system being developed
- allowing him to assess the level of QoS offered by a component

A limited amount of research literature is available for components quality. Most research effort has been placed on the quality of COTS components where some quality models have already been proposed [5,6,12,30]. Bertoa and Valecillo proposed a modified version of the ISO9126 software quality model along with associated metrics for the effective evaluation of COTS components [12]. They claim that international standards from ISO and IEEE are too general for dealing with the specific characteristics of software components and thus a more customized quality model is required to address this issue. In the same sense, the model they propose cannot be efficiently applied to original components as the differences with COTS in terms of scope, size and number of functions provided, necessitate the reformation, adjustment and enhancement of existing models (including Bertoa's) so as to tackle the special characteristics of this type of software units.

A similar effort from Brahnmath et al., calls for an objective paradigm for quantifying the quality of service of COTS components [12]. They proposed a QoS catalog for software components as a first step in quantifying their quality attributes. Their work is a part of a larger project (Uniframe), which targets at unifying the existing and emerging distributed component models under a common meta-model for the purpose of enabling discovery, interoperability, and collaboration of components via generative programming techniques. The Quality Objects (QuO) framework provides QoS to distributed software applications composed of objects. QuO is intended to bridge the gap between the socket-level QoS and the distributed object level QoS [12,3]. This work mainly emphasizes on specification, measurement, control and adaptation to changes in quality of service. QuO extends the CORBA functional IDL with a QoS description language (QDL). Our work aspires to complete the aforementioned relevant literature focusing on the missing part of original software components quality.

The rest of the paper is organized in five sections. Section 2 provides a general description of the ISO9126 quality characteristics, along with some guidelines for the modifications needed when customizing it for original software components. Section 2 also describes the proposed quality framework for original components along with functional metrics in order to provide rough evaluations of the quality of software components. Section 3 provides guidelines as to how the quality framework can be applied and assesses its applicability. Finally, Section 4 summarizes findings, draws some conclusions and outlines the focus of our further research.

2. Quality in software components

As already mentioned, the available research literature addresses the quality of COTS components only and not of original software components. Thus, the present paper may be considered as a first attempt to fill this gap and provide a framework for evaluating the quality of original components. We believe that this is essential for the software development industry since nowadays original components far outnumber COTS components as more and more developers are adopting CBSE to speed up their development tasks.

Before moving to present how quality in software components may be analyzed and assessed, we will briefly outline widely known key issues on software quality and attempt to indicate the relation of the quality model we will propose with other models.

Garvin [10] discusses software product quality from five different perspectives: (i) transcendental, which represents an abstract, ideal picture of quality we would like the product to have; (ii) user, which consists of measurements of product characteristics; (iii) manufacturing, which takes a process view and examines conformance to good software process standards; (iv) product, which is essentially an evaluation of a product's inherent characteristics; and (v) value based, a view according to which quality depends on what the customer is willing to pay for. Our approach takes into account the user and product views and addresses quality based on the product characteristics, although the difference between external and inherent characteristics is not so clear. We look at the software component as a set of different and measurable characteristics and we work towards identifying the quality level for each one. How this is practically done will be analytically described in the following sections. The basic idea, though, is that we produce a quality model especially for *original software components* and use it as a guide to assess quality. Quality models have been developed years ago, the most widely known of which being McCall's and Boehm's, and more recently the ISO9126. A brief reference to these models follows:

McCall's model is an early, product-oriented quality model, which shows how external quality factors, such as correctness, reliability, efficiency, etc., relate to product quality criteria, such as completeness, accuracy, error tolerance, etc. These product criteria can be measured and thus external quality factors may be assessed [23]. Boehm [6] proposed the same hierarchical structure as McCall's model, but also put emphasis on users' expectations and hardware performance. The complexity of Boehm's model is equal to that of McCall's, that is, the quality criteria are related to a variety of quality attributes with relations sharing common attributes. Finally, the ISO9126 model follows the same approach as the other two aforementioned models, with the important difference being that the level of complexity is lower here as the hierarchy is more strict: each quality characteristic is related only to exactly one attribute, with no common attributes between characteristics.

Our framework relies on a quality model that shares similar features with the three models mentioned above, following a hierarchical structure of quality characteristics which are decomposed to sub-characteristics and attributes. Our model adopts the strict approach of the ISO9126 standard as regards its hierarchy, but also gives details on how to measure each attribute, an issue which is not addressed by the ISO model. Furthermore, as Pfleeger states [25, p. 524] there is no clear justification of the decomposition of higher and lower level quality characteristics in any of the previously mentioned quality models, let alone their interconnections and dependencies. We will attempt to provide such a justification and explain the hierarchy of our model by describing the essence of each higher and lower level characteristic.

Following is a more detailed analysis of the ISO9126, which will serve as a raw basis to refine and customize, thus forming our model, taking into account the particularities observed when dealing with original software components.

2.1. ISO9126 software quality model: Similarities and differences

The description of the proposed framework utilizes the same terminology as the one used in the ISO9126 in order to make the two models comparable. Therefore, our framework (quality evaluation methodology and quality model based on the ISO9126) includes a set of characteristics and sub-characteristics, as well as the relationships between them that provides the basis for specifying and evaluating the quality of original components. A quality characteristic may be conceived as a set of properties of a software product by which its quality can be described and evaluated. A characteristic may be refined into multiple levels of sub-characteristics called attributes. An attribute is a quality property to which a metric can be assigned, but not all attributes have to carry a metric.

According to the complete list of the ISO9126 quality model [19] depicted in Table 1, there are six major characteristics, namely functionality, system reliability, usability, efficiency, maintainability and portability, along with their associated sub-characteristics. Functionality expresses the ability of a system to provide the required services and functions, when used under specified conditions, while reliability is an indication of the confidence that the software will live up to the expectations. Usability indicates the understandability of software as well as the easiness to learn and operate it. Efficiency is related to the performance of software and maintainability to the means provided by the software to be tested, upgraded and customized. Finally, portability indicates the level of adaptability/installability of a software product to different environments, as well as its conformance to related standards.

Table 1
ISO9126 quality model

Characteristic	Sub-characteristics
Functionality	Suitability
	Accuracy
	Interoperability
	Compliance
	Security
Reliability	Maturity
	Recoverability
	Fault tolerance
Usability	Learnability
	Understandability
	Operability
Efficiency	Time behavior
	Resource behavior
Maintainability	Stability
	Analyzability
	Changeability
	Testability
Portability	Installability
	Conformance
	Replaceability
	Adaptability

The original software components quality framework is created mainly by modifying and/or refining the quality model of ISO9126 in order to reflect better the notion of original components. Specifically, we need to modify the model in light of the differences between software products and components. For example, portability, as described in the next sub-section, is completely removed from the list since not only it is regarded as an inherent characteristic of all components but it is also effectively superseded by the interoperability sub-characteristic of the functionality characteristic. At this point, it is important to identify the users of this quality framework: The users of the framework are considered to be primarily software developers (original component developers and re-users), which will handle the integration of the components to their systems. This fact has implications to the usability characteristic of ISO9126, which now targets a different set of users than in the traditional software product. Additionally, the framework can also be a valuable tool to component developers since it will enable them not only to assess the quality of their components but also to identify areas of improvement that can result to a potential competitive advantage. Of course, the end users can indirectly be considered as users of our model since eventually they will also be benefited from a quality component, but their demands may be regarded as part of the requirements that re-users have to fulfill.

Looking at the problem from a general perspective, one has to consider the intricacies of original components in contrast to COTS components. For instance, we need to consider the fact that original components have to be made sufficiently abstract to be reusable a number of times, while at the same time they need to be sufficiently specific in scope so that they are distinguished from COTS components [2]. We also have to bear in mind that since original components are lesser in scope, they find more uses and therefore their need for interoperability increases. Original components are usually less customizable than COTS components in the sense that they do not include that much functionality which may be tuned or refined through settings and/or preferences to meet the specific needs of the working environment in which COTS are installed and run.

A final factor to be considered here is the type of original component we are evaluating. Components are generally distinguished to black-box and white-box components [22,24]. Some of the quality characteristics have different interpretations for each group, the most notable of which is the customizability of a component. White-box components are completely customizable (with few exceptions), since the source code is available, whereas black-box components are only customizable to the level of parameterization provided by the component vendor. The analysis of each quality characteristic participating in the proposed model, which is given in the next sub-section, includes also a description of the diversifications observed in the interpretation of the characteristic for each group (black-box, white-box).

2.2. Original software components quality model (OSCQM)

Based on the previous discussion, which briefly outlined the features of components that had to be viewed under a quality perspective and the main differences from a software system quality, it is evident that not all characteristics of a software product as defined by the ISO9126 (Table 1) are applicable to original components. As it can be seen in Table 2, which proposes a list with all characteristics, sub-characteristics and attributes of original components, most of the ISO9126 quality characteristics remain intact, but their sub-characteristics are modified in order to reflect better the notion of components. Consequently, new attributes are defined along with metrics that are essential in providing a meaningful assessment of an original component [36]. The only major change to the ISO9126 model is the removal of the Portability characteristic from the list, which is superseded by the interoperability sub-characteristic of Functionality, and therefore there is no meaning in evaluating it.

Following are detailed explanations of the changes to the ISO9126 model, their reasoning and the proposed attributes and metrics required to give quantifiable measures to the quality characteristics (where possible):

2.2.1. Functionality

This characteristic indicates the ability of a component to perform according to its specifications. It measures the level to which the services and functions of the component satisfy the users (component developers and re-users). In addition, the functionality characteristic attempts to evaluate the ability of a component to be reused (interoperability) and its ability to offer secure service according to user needs. It directly matches with the functionality characteristic of ISO9126 but with different sub-characteristics. More specifically the Interoperability, Security and Suitability sub-characteristic maintain their meaning with the only exception that the term Suitability is replaced with the term Completeness which reflects better the scope of the sub-characteristic and the attributes to be evaluated. Additionally, Accuracy is moved to the Reliability characteristic (Result Set sub-characteristic) since we consider accuracy to be a feature of reliability instead of functionality. Finally, for evaluation reasons, Compliance is temporarily removed since currently there are still no official standards to which components must adhere to. In case that such standards emerge in the future, this model may be revised to reflect on them since compliance to standards is considered as an important parameter in components' evaluation.

2.2.1.1. Interoperability. This is the most important characteristic of a component since it designates the ability of a component to be reused. Reusability is considered the cornerstone of the CBSE success and thus the evaluation of interoperability is of prime importance. Attributes are:

- **Platform Independence**: Indicates whether the same component can be used in different component models, i.e., the same component can be used in Java model architecture (RMI and Gini), CORBA model architecture, Distributed Component Object Model architecture (COM/DCOM) and other similar models. *It is measured by the ratio of the number of the platforms supported to the number of the most-used platforms*
- **Operating System Independence:** Indicates whether the component can be installed in different operating systems like Windows, Linux, other Unix-flavored Operating Systems, AIX etc. *It is measured by the ratio of the number of Operating Systems supported to the number of the most-used Operating Systems.*
- **Hardware Compatibility:** Designates whether the component can be used in various types of hardware, i.e., personal computers, laptops, Personal Digital Assistants (PDA's) Smart Phones etc. *It is measured by listing the type of devices supported.*
- **Data open-format Compatibility:** Examines whether the data outputted from the component is compatible with well-known formats. If only proprietary formats are supported, then the ability of a component to be reused is severely handicapped. *A Provided or Not Provided metric is used with "provided" implying that an open format is used (XML, ASCII) and "not provided" implying that a proprietary format is used.*

2.2.1.2. Completeness. Expresses how well the component fits the user (re-user) requirements and whether it offers the service advertised at a satisfied level or not. It attempts

Table 2
Proposed quality model for original components

Characteristics	Sub-characteristics	Attributes	Metrics
Functionality	Interoperability	Platform Independence	Ratio of platforms supported to most-used platforms
		OS Independence	Ratio of OS supported to most-used OS
		Hardware Compatibility	List of types of devices supported
		Data open-format compatibility	Provided or Not
	Completeness	User satisfaction	Level of satisfaction
		Service Satisfaction	Functions Ratio
		Achievability	Provided or Not
	Security	Access Control	Provided or Not
		Resistance to privilege escalation	Provided or Not
		Auditing	Provided or Not
		Data Encryption	Provided or Not & Number of encryption algorithms
Reliability	Service Stability	Error Prone	Number of errors/crashes per unit of time
		Error Handling	Provided or Not
		Recoverability	Provided or Not
		Availability	Average uptime
	Result Set	Correctness	Ratio of correct results to total results
		Transactional	Provided or Not
Usability	Learnability	Time to use	Average time to learn how to use
		Time to configure	Average time to learn how to configure
		Time to administer	Average time to learn how to administer
	Help tools	Help completeness	Provided or Not – Level of satisfaction
		User Manual	Provided or Not – Level of satisfaction
		Installation and Administration Documentation	Provided or Not – Level of satisfaction
		Support Tools	Provided or Not
	Operability	Operation effort	Level of satisfaction
		Customizability effort	Level of satisfaction
		Administration effort	Level of satisfaction
	Identifiability-Reachability	Directory Listing	Ratio of popular directory listings to total popular listings
		Search & Retrieve	Level of satisfaction
		Categorization	Provided or Not
		Explainability	Level of satisfaction
Efficiency	Response Time	Throughput	Number of requests per unit of time
		Capacity	Number of concurrent users
		Parallelism	Provided or Not
	System Overhead	Memory Utilization	RAM size
		Processor Utilization	Level of processor speed
		Disk Utilization	Secondary memory size
Maintainability	Changeability	Upgradeability	Level of satisfaction Provided or Not
		Debugging	Provided or Not
		Backward compatibility	Provided or Not
	Testability	Trial version	Level of satisfaction
		Test Materials	Level of satisfaction
	Customizability	Parameterization	Black-box: Ratio of changeable parameters to total parameters White-box: Level of satisfaction
		Adaptability	Level of satisfaction
		Priority	Provided or Not

to give an overall idea of the level to which the component covers the needs of the users in terms of services offered. Attributes are:

- **User Satisfaction:** It measures the level to which the component meets the re-user requirements who acts like a broker of end-users. It is noted that this attribute can only be measured by the re-users (software architects or developers) and not by the component provider. Thus, it might not give direct guidelines to component providers as to how to build quality components. *Expressed as a level of satisfaction.* Satisfaction level works on a scale of 5 values having 1 as "Not satisfied at all" and 5 as "Completely satisfied".
- **Service Satisfaction:** It expresses the level to which the component offers everything required for the service advertised. For example, if the component provides a calculator service then it has to be evaluated on the number of calculator functions that are implemented. *This attribute is measured by the ratio of the number of functions desired by the user to the total number of functions provided by the component.*

- **Achievability:** It indicates whether a component can provide a higher degree of service and/or more services than expected. For example, a component providing a charting service might additionally offer a 3D-view option, a feature that may not be needed originally. *A Provided or Not Provided metric is used, coupled with the radio of the functions required by the user and the functions provided by the component.*

2.2.1.3. Security. A very important sub-characteristic since it directly relates to the user confidence regarding the component. It denotes whether the component employs control methods for accessing its services and whether a security failure of the component will result to a system wide security failure. Also, it indicates the presence of auditing mechanisms and methods for data protection (e.g., encryption).

- **Access Control:** Indicates the presence of access control mechanisms like authentication and authorization in accessing the services of a component. *A Provided or Not Provided metric is used.*
- **Resistance to Privilege escalation:** It examines whether a security flaw in the component can result to privilege escalation and hence to a system security breach. Recent events, for example security flaws in Microsoft side technologies (buffer overflows, cross-site scripting), indicate that this is a very important attribute that should be part of a quality framework. *Again a Provided or Not Provided metric is used.*
- **Auditing:** Designates whether auditing mechanisms are implemented by the component in order to record user access to the system and associated data handling. Auditing includes tracking unauthorized access to the system as well as reporting/tracing of the users actions. *A Provided or Not Provided metric is used for this attribute indicating at least the presence of an auditing mechanism.*
- **Data Encryption:** It expresses the ability of the component to offer encryption services to protect the data it handles. This is vital in mission critical systems but it is becoming more and more important for other conventional systems as well. *It can be measured by a Provided or Not Provided metric, coupled with the number of encryption algorithms present in the component.*

2.2.2. Reliability

This characteristic denotes the confidence that the original component exerts to its users. It is a direct measure of the stability of the original component as well as an indication of its ability to return correct results in a quality manner. It is considered an essential characteristic that contributes to the level to which a component can be reused. This characteristic also addresses the recovering capabilities of the original component. It directly matches with the corresponding characteristic in ISO9126, but with some modified sub-characteristics: Maturity is maintained as a sub-characteristic but it is renamed to Service Stability in order to encompass the meaning of all sub-characteristics (error prone, error handling, recoverability, availability). The other two characteristics listed in the standard (Recoverability and Fault Tolerance) are included as measurable attributes and the Results Set characteristic is introduced as mentioned earlier.

2.2.2.1. Service stability. It is a measure of the confidence that the original component is free from errors. In addition, it examines the ability of a component to recover after a failure and it also measures the duration that a component is available to offer a particular service. Attributes include:

- **Error Prone:** It examines whether the original component is susceptible to system errors and it addresses the frequency of the errors and their relative importance. Also, it denotes whether the component errors can result to a complete software system failure. *The number of errors per unit of time or the number of critical errors (crashes) per unit of time can be used as a metric for this attribute.*
- **Error Handling:** It denotes whether the original component can handle errors and refers to the mechanisms used to handle these errors. *A Provided or Not Provided metric is used in this case denoting whether a mechanism for error handling is provided or not.*
- **Recoverability:** It designates the ability of a component to recover when errors occur (fault tolerance). In addition, it indicates the level of efficient recoverability, i.e., whether data is lost, or whether the component failure will result to a system failure. *A Provided or Not Provided metric is used.*
- **Availability:** Indicates the duration that a component is available to offer a particular service. *The average uptime of the original component without serious errors or crashes can be used as a metric.*

2.2.2.2. Result set. This sub-characteristic asserts the correctness and the quality of the results returned by the component. It can be measured as the possibility to obtain incorrect results or no results at all. In addition, it specifies whether the component supports transactional-based computations, i.e., in case that a transaction fails all changes are rolled-back. Attributes include:

- **Correctness:** Evaluates the ability of the component to return correct results. In addition, it evaluates the quality of the results, for example their precision and their computational accuracy. *It can be expressed as a ratio of correct/precise results (i.e., the "as expected", or "should behave" results) returned to the total number of results.*

- **Transactional:** Indicates whether the component supports transactional processing and whether it can be used in transactional applications. Transactional components should be able to rollback all changes if a transaction fails. *A Provided or Not Provided metric is used.*

2.2.3. Usability

The Usability characteristic attempts to evaluate the easiness to learn how to use and ultimately how to incorporate the component into a software system, and also to indicate the presence and quality of help materials that will facilitate this process. This characteristic represents the most prominent example of the difference in meaning between software quality models (i.e., ISO9126) and component quality models. The difference lies to the fact that the end-users of original components are system architects and developers, whereas the end-users of traditional software are the people that interact with them. Although most of the sub-characteristics remain the same in terms of naming (only understandability is renamed to help tools) the meaning and the attributes are somehow different. Help tools need to target different people, and configuration capabilities are more important in software components than in software systems. Administering a component is different than administering a software system and documentation must emphasize on different things in each case (i.e., interfaces in the case of components, as opposed to "how to do" in user manuals). The content and quality of help materials are also different in the case of white-box and black box components. Additionally, Identifiability/Reachability is introduced in order to reflect the need for efficiently tracing and retrieving the desired components [16]. Sub-characteristics include:

2.2.3.1. Learnability. This attribute indicates the time required for a user to learn how to use, configure and administer a component. A user in this case is considered a software developer of average experience and knowledge (i.e., a user that is not at the trainee stage, has been involved with several projects in the past, but at the same time cannot be considered as a matured development leader in terms of knowledge and experience) that re-uses the original component. It is noted that this attribute does not apply to the original developers of the component completely since they are not the ones to evaluate it. However, learnability is a quality characteristic that has to be seriously considered by original developers when developing components since it will highly contribute to the overall acceptance of their components.

- **Time to use:** Estimates the time required for an average user to learn how a component works and how it can be used in a software system. *A time metric is used in this case that will measure the average time needed for a developer to learn how to use the component.*
- **Time to configure:** Estimates the time required for an average user to configure the component and make it ready for use. *The average time needed for a developer to understand the configuration parameters and learn how to configure the component is used as a metric for this attribute.*
- **Time to administer:** Estimates the time required for an average user to perform administration tasks on the component, i.e., temporary files that have to be removed. *Following the same pattern, the average time needed for a developer to learn how to administer the component is used as a metric.*

2.2.3.2. Help tools. This sub-characteristic denotes the availability of a help facility for the usage of the component, as well as the effectiveness of the help facility. Moreover, it indicates whether other type of documentation, like user and installation manuals or tutorials, is included with the component. It must be emphasized that the extent and the depth of these materials vary with the type of component. For white-box components these materials have to be very detailed since they also have to explain the source code of the component and describe the overall programming philosophy. For black-box components there is no need for this.

- **Help completeness:** Indicates whether help files are available for the component and the completeness of the help system. A number of metrics are used to evaluate this attribute. *A Provided or Not Provided metric is used to identify the presence of a help facility and a level of satisfaction metric is used to express the completeness of the help system.*
- **User Manual:** Indicates the presence of a user manual. *A Provided or not Provided metric is used, coupled with a level of satisfaction metric that attempts to evaluate the completeness, readability and understandability of the manual.*
- **Administration and Installation Manual:** Denotes the presence of these two manuals. *Metrics are the same with those in the user manual.*
- **Support tools:** Denotes the presence of tutorials, demos, sample code and examples that will facilitate the usage of the original component. Additionally, it specifies the presence of additional support, like on-line support mechanism and/or support forums and newsgroups. *A Provided or Not Provided metric is used, along with a listing of the additional support tools provided.*

2.2.3.3. Operability. Indicates the level of effort required by re-users to operate and administer the component. In addition, it evaluates the effort required to customize the component. Attributes include:

- **Operation effort:** It evaluates the effort required to operate the component on a regular basis. For example, whether the component requires some tasks to be executed manually in order to operate, or whether any log files have to be reviewed first in order to investigate malfunctions. *A level of satisfaction metric is used designating the user's satisfaction by operating the component. Little effort results to more satisfaction and vice versa.*
- **Customizability effort:** It evaluates the effort required to customize the component through pre-defined interfaces. This attribute goes hand by hand with the customizability sub-characteristic of maintainability, but in this case we are evaluating the overall effort required and not the individual factors affecting customizability. *A level of satisfaction metric is used.*
- **Administration effort:** Same concept as the operation effort attribute, but the focus is to evaluate the effort that pertains to administration tasks. *A level of satisfaction metric is used as before.*

2.2.3.4. Identifiability-reachability. This sub-characteristic denotes the ability of the component to be identified (discovered) and retrieved for usage. It attempts to address a severe problem that most developers face when dealing with CBSE, that is, the ability to find quickly the component that really suits their needs. This sub-characteristic touches upon issues like directory listing, component categorization, and explainability of the component. It is directly related to the quality of the documentation provided by the component vendor. It is noted that this sub-characteristic depends heavily on the component vendor, but it affects the quality of the component in a broader context.

- **Directory Listing:** This attribute evaluates the easiness with which a component can be found. Components are generally found in directory listings provided by dedicated sites in the World Wide Web or special software magazines. A component present in these listings can be more easily identified by potential re-users. *The metric in this case is the ratio of popular directory listings that the component is advertised to the total number of popular directory listings (e.g componentsource, tucows, cnet, etc.).*
- **Search & Retrieve:** Evaluates the easiness to search and retrieve the component. For example, it evaluates the ranking of the component in the search result of popular search engines or directory listings. In addition it designates the simplicity of retrieval (download), for example whether there is a certain procedure to obtain a trial version of the component (e.g. registration, terms of service, etc.). *The metric in this case is a level of satisfaction arising from the search rankings, as well as a level of satisfaction regarding the retrieval procedure. The simpler the retrieval (download) the more the satisfaction of the user.*
- **Categorization:** Designates whether the component is classified to certain categories. For example the component can be classified by the platforms supported (e.g. COM/DCOM, JAVA, etc.), or the area of interest (e.g. charting, database, Internet, etc.). Categorization eases the search-and-retrieve process and gives a preliminary indication of the scope of the component. *A Provided or Not Provided metric is used.*
- **Explainability:** This attribute evaluates the ability to understand the component capabilities without installing it. It targets mainly the information that the component vendor provides when advertising the component. *A level of satisfaction metric is used in this case.*

2.2.4. Efficiency

The efficiency characteristic evaluates the performance of the component both in terms of requests serving speed as well as of the system overhead it requires to run. Both are considered equally essential in assessing the efficiency of a component and its ability to handle incoming requests rapidly. It corresponds to the efficiency characteristic of software quality in ISO9126 and sub-characteristics are virtually the same but have different naming. The different naming is applied for simplicity reasons.

2.2.4.1. Response time. It evaluates the time difference between invoking a component method and receiving a response from the component. Since a component might expose multiple interfaces with different input and output parameters, an average value will be considered. It is regarded as a very important characteristic since it usually affects the overall impression of the end users. For instance, a slow component will rarely be widely accepted irrespective of the functionality it offers.

- **Throughput:** Specifies the speed of the component to serve requests and produce output over a given period of time for a given environment. *It is measured as the ratio of the number of successfully served requests per unit of time, for example 45 requests/sec.*
- **Capacity:** Defines the number of concurrent users that can be handled by the component simultaneously without compromising performance. This is a very important attribute for web-based components where the number of users is generally very large. *An Integer metric can be used, specifying the number of concurrent users that can be supported.*
- **Parallelism:** Indicates whether the component supports synchronous or asynchronous invocation. In certain areas, asynchronous invocation can speed the whole process since it can be executed separately from normal execution of the software system. *A Provided or Not Provided metric is used.*

2.2.4.2. System overhead. This sub-characteristic evaluates the system resources required to run the component. It is very essential and coupled with response time it can give a complete picture of the original component performance. It targets to identify components with high performance levels but very resource-demanding. Attributes include:

- **Memory Utilization:** Designates the amount of memory needed by a component to operate. It is an essential attribute especially for wide-scope systems that incorporate a lot of components. *The number of KiloBytes (KB) of RAM required by the component to run is used as a metric, along with the relevant fluctuations (minimum, maximum).*
- **Processor Utilization:** Designates the amount of processing time (in CPU cycles) needed by a component to operate. The importance here is similar to that of memory utilization. *The average number of processor cycles required by the component to run is used as a metric.*
- **Disk Utilization:** Designates the amount of disk space needed by a component to operate. It includes both the disk space for installing the component and other material (documentation), as well as the disk space used temporarily during execution time. *It is measured by the number of KiloBytes (KB) the component requires on the disk, along with any fluctuations regarding temporary space.*

2.2.5. Maintainability

This characteristic evaluates the ability of the component to be maintained. Maintainability is expressed as the easiness with which the component is upgraded to a new version or is updated to fix errors. It addresses issues like changeability, testability of changes and level of customizability. The major change regarding the ISO9126 is the addition of customizability which is an important parameter in evaluating components, given the fact that in most cases the code is not released. Furthermore, changeability and testability maintain their status, whereas stability is demoted to an attribute of upgradeability. Analyzability is completely removed since for black-box components there is virtually nothing to analyze. Sub-characteristics include:

2.2.5.1. Changeability. It denotes the easiness and the effort required to modify the component. There are a number of issues to elaborate on, especially the obvious distinction between white-box and black-box components and the backward compatibility which is also crucial. Generally, white-box components are more upgradeable than black box components since the source code is made available to the re-user. However this must not be taken for granted since there are a number of other issues regarding the changeability of white-box components, like the clarity and the structure of the source code, the availability of comments, the readability and others. In the case of black-box components the re-users usually rely on the component vendor for upgrades and the automated features provided by the component. Attributes are:

- **Upgradeability:** It measures the easiness with which a component is upgraded to a new version. This attribute attempts to evaluate the features provided by the component in order to be upgraded and also it denotes the smoothness of the upgrade. For example, if a component upgrade will result to a complete system downtime then this will result to negative impressions. In addition, rollback features are investigated in case an upgrade is not successful. *It is measured with a level of satisfaction metric.*
- **Debugging:** It specifies whether the component facilitates functional debugging and evaluates the efficiency of the error messages returned by the component in order to understand the erroneous functioning of the component and correct it (i.e., error message “out of paper” for a printer monitoring component). The meaning of this attribute should not be confused with that of debugging at the source code level, since the latter is a feature embedded in the programming environment used to modify the code statements of the component, in case the source code is released. *The ratio of descriptive and understandable errors to the total number of error messages is used as a metric for this attribute. Also, a Provided or Not Provided metric is used to denote the existence of a debugging facility.*
- **Backward Compatibility:** This attribute designates whether a new version of the component is compatible with previous versions. It is a very important attribute since if backward compatibility is not provided then a component re-user might be forced to re-write the software system in order to accommodate system changes. It is generally advisable that new component versions must at least maintain their interfaces intact. *A Provided or Not Provided metric is used.*

2.2.5.2. Testability. It examines the features provided by the original component that facilitate its testing in terms of modifying the component as well as initially testing the component. The latter might as well belong to the usability characteristic, but for avoiding duplications we decided to include it here. Attributes include [11,28]:

- **Trial Version:** It denotes the existence of a trial version that will enable the re-user to evaluate the functionality of the component. In addition, the time and scale of the trial version is considered, i.e., whether the trial version is a full or a limited version. Also it specifies whether the component is freeware, shareware or commercial.*It is measured with a level of satisfaction metric.*

- **Test Materials:** It denotes the existence of other useful test materials like demos, sample code, Logical and Data Flow diagrams, source code and test cases. *Again a level of satisfaction metric is used.*

2.2.5.3. Customizability. This sub-characteristic evaluates the ability of a component to be customized according to user needs. White-box components are expected to have a greater ability in customization than black-box components, since the source code is provided along with the component. In the case of black-box, customization is achieved only through functionality provided by the component vendor. Attributes are:

- **Parameterization:** It indicates the number of parameters exposed by the component available for change. As already mentioned this is not true for white-box components since the source code is provided. In this case, the easiness to alter the component through its source code is measured. *Black-box components parameterization is measured by the ratio of the number of parameters that can be changed to the total number of parameters desired by the user. White box components parameterization is measured with a level of satisfaction metric pertaining to the readability of the source-code.*
- **Adaptability:** It evaluates the ability of a component to adapt itself to a changing environment at runtime. For instance, a component-based agent can be considered a highly adaptable component. This must not be confused with the interoperability sub-characteristic which evaluates the ability of a component to be used in different environments. *A level of satisfaction metric is used.*
- **Priority:** It indicates whether the component provides prioritized service, i.e., some of its functions assume priority at runtime [19]. *A Provided or Not Provided metric is used.*

3. Application of the original components quality framework

Earlier in this paper we referred to a number of quality models, emphasizing on the fact that none of these models addresses the issue of how to perform software evaluation in practice. We have to admit, though, that this issue is very difficult to tackle and it can be safely inferred that there is no single answer, that is, there is no single and straightforward application method for these models. Rather, each component developer or re-user has to view a given quality model under the perspective that suits the needs of the application system to be developed. In this section we attempt to move one step forward and suggest approaches of how to apply the proposed quality evaluation framework according to the type of user (re-user or developer). A set of specific and simple steps are proposed and described, which build upon those characteristics of our quality framework that are considered as essential for the evaluation of components.

As mentioned earlier, there are two types of users that can be assisted and benefited by our framework: the re-users of the component and the component developers. Re-users usually employ two different methods when selecting and evaluating which components are suitable for the system they build. The first is the application-driven method (Fig. 2a) in which the system architecture is decided first, the components to be needed are specified next and finally the discovery, evaluation and incorporation of the appropriate components follows. The second is the reuse-driven method (Fig. 2b), where an outline of the system requirements is prepared and then a preliminary search and evaluation takes place to identify possible component candidates and their capabilities. Then the system requirements are modified, where possible, according to the findings, a final architectural design is prepared and the search, evaluation and incorporation of components are finalized.

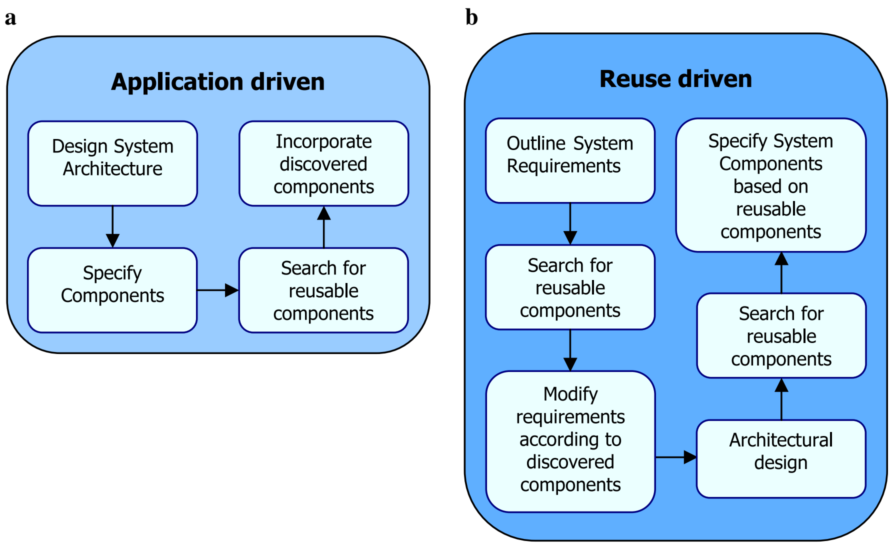
a
Application driven
Design System Architecture
Specify Components
Search for reusable components
Incorporate discovered components
b
Reuse driven
Outline System Requirements
Search for reusable components
Modify requirements according to discovered components
Architectural design
Search for reusable components
Specify System Components based on reusable components

Fig. 2. Software process and reuse: (a) opportunistic reuse (b) development with reuse.

In both methods the application of our quality framework can greatly enhance the evaluation process yielding more entrusted results. The approach we suggest is that the people involved in this process (architects, developers, integrators) should prepare a customized checklist with all the characteristics, sub-characteristics and attributes required and place weights on them according to their significance. A weighting scheme is necessary for the application of the quality model since it allows for emphasizing on the most-wanted characteristics for each occasion and additionally it provides for quick adaptation of the model according to the component being evaluated. This scheme represents a simple, effective and time-saving way for providing an evaluation mechanism for this framework in contrast to other methods (e.g., fuzzy logic [4,30]) which are complicated and time-consuming. For instance, if an original component that handles the access to a software system is to be developed, then the security sub-characteristic and relevant attributes will gain more importance than the learnability sub-characteristic. The customized characteristics checklist is then applied to all candidate components and quality is evaluated according to the metrics specified in our framework. It is noted that this process is not expected to yield a single result about the suitability of a component. The metrics are so different (e.g., time, integer, level of satisfaction) that there is no way to generate a formula that considers them all. Therefore, the evaluation must be performed on a per-characteristic basis considering the weights of the individual attributes that constitute a characteristic. Generally speaking, the checklist essentially contains a set of minimum acceptable quality criteria which must be met in any case. The framework (implemented through the checklist) ensures that all necessary quality elements are addressed and priorities are identified according to the user needs.

The procedure of evaluating a ready-made component available in the market is outlined in steps as follows.

Step 1. Identify the component's general scope and requirements
Step 2. Prepare a checklist with quality characteristics and attributes based on the proposed framework, along with relative weights and metrics
Step 3. Apply the checklist to all candidate components
Step 4. Discard components that do not meet the minimum acceptable quality criteria
Step 5. Evaluate the components on a per-characteristic basis
Step 6. Integrate the most appropriate component

In the case of original developers, the application of our quality framework is somehow different. If a component that will join an existing market is to be developed, the component developer needs first of all to identify and assess the competitive components. A custom checklist as described above can be prepared and all competitive components may be assessed according to the metrics specified in our framework. Having all these quality information in hand, the component developer can isolate areas of improvement, or even better, areas that were not covered at all, giving him/her an important competitive advantage. After the development of the component, the developer can re-assess his/her component quality in order to measure how it fairs with competition. In the case that the component will join a new market the assessment of existing components will simply be omitted. However, the framework will mainly be used as a point of reference in order to ensure that all quality attributes are incorporated in the component and there are no significant omissions that will hinder the successful deployment to the market. It is noted that a customized checklist that will reflect the scope of the component to be developed should be prepared as before and the minimum acceptable criteria must be met. The steps are outlined below:

Step 1. Identify the component's general scope and requirements
Step 2. Search the market for existing components
Step 3. Prepare a checklist with quality characteristics and attributes based on the proposed framework, along with relative weights and metrics
Step 4. Apply the checklist to all associated components
Step 5. Identify areas of improvement (in terms of quality)
Step 6. Identify features not covered at all but are present in the proposed framework
Step 7. Build the component
Step 8. Re-assess the quality of the component using the proposed framework and evaluate it against competition

Perhaps the most difficult part of the procedures described above is the identification of the correct and most relevant set of attributes, as well as of their weights, that will serve as the bearing minimum of the quality offered by the component under evaluation. One may also argue that this is a quite subjective task, and this is true of course as it entails the subjective judgment of the evaluator. Nevertheless, the requirements to be met by the component, both functional and non-functional, will eventually determine which of the characteristics and attributes should be included in the checklist.

In the next sub-section the issue of subjectivity addressed above is handled through the utilization of experts in the field of CBSE, the contribution of which was catalytic in determining the quality attributes in the demonstration examples that follow.

3.1. Evaluation of components: A real case scenario

Focusing on the minimum quality criteria that must be met for every component, we will now describe a simple rationale to shrink down the original list of quality characteristics. Although all of the characteristics included in both the ISO9126 model and our quality framework are considered highly important, the scope of original components is sometimes so limited that it does not allow for the realization of all the characteristics and sub-characteristics. Consequently, the short list of must-have characteristics and sub-characteristics to be produced should represent the minimum acceptable criteria in evaluating the quality of original components. At this point we would like to make clear that this short list does not by any mean underestimate the importance of the other characteristics, but may be considered as a bearing minimum for revealing a sufficient overall quality picture of the component involved. For large-grained original components most probably the complete list has to be employed in order to correctly assess a component's quality.

The reduced list is shown in Table 3 and includes sub-characteristics and attributes from the majority of the quality characteristics. The selection of these characteristics was not made arbitrarily; it was based on a survey contacted from a pool of 20 developers in 2 different countries that have been involved with CBSE. Developers were given the quality framework and were asked to specify and weigh which items believed that are a must-have in order to incorporate a component into a system they are building. More specifically, developers were guided to select the smallest number of the most significant attributes for which a reuser or original component developer almost always demands a certain threshold level of quality. Above this level a component under evaluation may be regarded as a good case and below it the component is rejected. This selection was to be performed taking into consideration three development perspectives: (i) The application domain; (ii) The specific software application; and (iii) The stakeholders (developers, reusers, end-users). As each of these perspectives adds a piece of the overall functional behaviour and constraints, the number of different approaches as regards to how significant a specific quality attribute may be increases. One thing, though, that should be kept constant was the list of attributes to which all of the aforementioned perspectives draw their attention for achieving high quality.

A three level scale to characterize each attribute was then proposed, with the linguistic values of "absolutely necessary", "highly significant" and "moderately significant" reflecting the significance of the attribute to the component's quality arising from one or more perspectives. Developers then marked the significance level for each attribute. If a percentage of equal or more than 75% of the developers indicated the "absolutely necessary" level then the corresponding attribute was added in the checklist. The selected characteristics are outlined in Table 3, the last column of which also indicates in parentheses the number of developers that characterized a specific attribute as absolutely necessary and the corresponding percentage.

The list of the must-have characteristics was then distributed to the developers participated in the process for commenting, agreeing and finalizing the list. It was a common belief that the list corresponds indeed to a set of characteristics that developers mostly look to evaluate even if they are not following a formal evaluation procedure. The completeness of the components in terms of user and service satisfaction (software application, stakeholder perspectives) and the security (software application, application domain perspectives) offered by the components are inherent properties asked by any developer. The same is valid for the stability of the service and the correctness of the results (both addressed by the software application and the application domain perspectives). Efficiency (software application, application domain perspectives) is a common quality attribute for every software system, not only components. A slow component or a component that fails under load will never survive the market. Finally, the maintainability is always an issue for re-users. Backward compatibility and upgradeability easiness are the items developers mostly look at (both attributes addressed by the software application and the application domain axes). Of course most of the developers indicated that the list mostly depends on the application domain for which the component is build, but generally there was a consensus that the above list satisfies the minimum quality requirements for the vast majority of components.

Once the list was fixed, the next question was what values the relevant weights should take to accompany each metric. All developers agreed that a weighting scheme is representative only of the situation under study, that is, each attribute may be assigned the appropriate weight value that reflects best the requested functionality, the criticality of the software system under development and the constraints that must be met. Thus, it was decided that weight values should not be á-priori defined but be best decided on the case involved. This is better illustrated through our two case studies that follow.

The rest of the characteristics and sub-characteristics are used once candidate components pass the must-have quality evaluation to select the best among components of similar or almost equal quality. In this case these attributes are evaluated according to the scope of the component and relevant weights are placed. For instance, if a component is providing a calculator service then the parallelism attribute is of limited importance. Some of the attributes might be

Table 3
Minimum acceptance criteria

Characteristic	Sub-characteristic	Attribute
Functionality	Completeness	User Satisfaction (20–100%)
		Service Satisfaction (10–100%)
	Security	Access Control (16–80%)
Reliability	Service Stability	Error Prone (17–85%)
	Result Set	Correctness (19–95%)
Efficiency	Response Time	Throughput (18–90%)
		Capacity (15–75%)
Maintainability	Changeability	Upgradeability (19–95%)
		Backward compatibility (16–80%)

highly suitable for a certain case; some of them might be not.

In order to illustrate better how our framework can be applied, we demonstrate how we can apply it in searching for a two-way SMS messaging component to be incorporated in an on-line trading platform (acting as a re-user) and in developing a charting component that will compete in the market of providing charting capabilities in the financial area (acting as a component developer). The first case represents a real-world situation (we actually used the selected component into a real software application) whereas the second case is just an illustration of the process (we did not actually aim to develop a charting component to compete with the existing ones).

CASE STUDY 1 We begin first with the re-user scenario. According to the steps of our methodology the first thing to be done was to define the scope and requirements of the component. The major requirements set for the component were the following:

- ActiveX/COM component for Windows platforms
- Send/receive up to 50 messages per minute
- Support database integration
- Support various transports (GSM modems, GSM phones, TAP, Internet)
- Provide notifications about SMS messaging

Since the scope of the component is not that broad we decided that the short list with the must-have quality characteristics was sufficient and was not enriched with other attributes. Following, we needed to prepare the checklist and place relevant weights (Table 4).

The next step to perform was to select the candidate components and apply the checklist to each of them. Candidate components were selected from multiple sources basically the Internet and software magazines. Four components were selected: ActiveSMS from Intellisoftware [18], SMSDemon from dLoad [32], SMServer from GSM-Active [15] and SMSZyneo by Zyneo [33]. Table 5 illustrates the results of the evaluation. It is not the scope of this paper to explain the rationale of our evaluation for all characteristics (i.e., how marking was performed), so we are providing only some indicative examples so as to introduce the philosophy followed in our case. For the Service Satisfaction attribute a list of the desired functions was prepared (12 in total) and each component was evaluated according to the functions it supported. In the case of Correctness, the components were configured to send 100 messages one after the other and the messages not received were tracked. From Table 5 one can easily discern that the SMSZyneo component failed to deliver 15% of the messages, thus it was scored less than the rest of its competitors.

According to our major requirements, two components failed to meet the performance condition (throughput) and therefore were rejected (SMS Demon and SMSZyneo). The remaining two components were to be evaluated on a per-characteristic basis according to their weight. Since their differences lie at the quality characteristics of service satisfaction and throughput/capacity, the deciding factor

Table 4
Checklist with weights for Case study 1

Characteristic	Sub-characteristic	Attribute	Metric	Weight (%)
Functionality	Completeness	User Satisfaction	Level of satisfaction	20
		Service Satisfaction	Functions Ratio	20
	Security	Access Control	Provided or Not	5
Reliability	Service Stability	Error Prone	Number of errors/crashes per unit of time	10
	Result Set	Correctness	Ratio of successful SMS sending	10
Efficiency	Response Time	Throughput	Number of requests per unit of time	15
		Capacity	Number of concurrent users	10
Maintainability	Changeability	Upgradeability	Level of satisfaction	5
		Backward compatibility	Provided or Not	5
				100

Table 5
Checklist with scores for Case study 1

Attribute	ActiveSMS	SMSDemon	GSMActive	SMSZyneo
User Satisfaction	5	2	5	1
Service Satisfaction	9/12	5/12	8/12	6/12
Access Control	Provided	Provided	Provided	Provided
Error Prone (errors/day)	0/day	1/day	0/day	0/day
Correctness (correct results/total results)	1	1	1	0.85
Throughput (messages/min)	60/min	8/min	120/min	8/min
Capacity (number of GSM modems supported)	8	1	16	1
Upgradeability	5	4	5	4
Backward compatibility	Provided	Provided	Provided	Provided

would be their weight. Therefore the ActiveSMS component was selected since the weight for the service satisfaction was higher than the weight of the throughput/capacity characteristic. Of course this is a difficult decision to make since the weight difference is just 5% but the performance difference is double. This is a call for each developer to take and it highly depends on the application to be developed. In our case the extra function provided by the ActiveSMS component was deemed as crucial and this was the main reason why it was preferred. This is to prove our earlier statement that there is no single formula that can be applied to select a component but rather an approach based on the importance of quality characteristics, emerging from the special needs of the system to be developed. Nevertheless the application of our framework greatly assisted towards the quick identification of the right component to use and provided significance guidance through the evaluation. The evaluation lasted two working days.

CASE STUDY 2 In our second case (developer case) we assumed that we wanted to develop a component with charting capabilities that aimed at competing in the financial market with other components with similar functionality. The first step in implementing our quality framework was to identify the component scope and requirements. The scope has already been defined and the major requirements apart the ones addressed by the quality framework were as follows:

- ActiveX/COM component for Windows platforms
- Compatible with various data sources
- Handle more than 500 users concurrently
- Chart export in various formats
- Support for web applications
- Support 3D rendering
- Multiple chart types (bars, pie, lines)
- Supports zooming and scrolling

The second step was to assess the competition. In searching for similar components in popular sources like componentsource.com, downloads.com, tucows.com (links are listed in the references section) we identified four components with similar functionality that were deemed as the top class for charting applications. Our target was to develop a charting component that would overshadow all others available. The four components were the ComponentOne [7] Chart, the Infragistics NetAdvantage [17] 2003, the Dundas Chart [9] and the TeeChart Pro ActiveX [35]. Our next task was to prepare the checklist based on the proposed quality framework. A charting component with complete capabilities as needed by today's financial applications can definitely be classified as a large-grained original component. Therefore, the complete list of characteristics of our quality framework (except some which are completely out of scope) was utilized in order to evaluate the selected components and furthermore to assess the quality of our component (Table 6).

As expected for a charting component more weight was placed on functionality (25%) and efficiency (24%) issues and then on usability, maintainability and reliability. This list represented the desired functionality that to be provided by the component. By filtering the selected components using this list not only we identified the strong points of each candidate component but we also managed to reveal areas of weaknesses where our component could exploit in order to gain competitive advantage. Table 7 illustrates the results of the evaluation which lasted 4 days, one for each component evaluated. Again it would take too long to explain the rationale of our evaluation and therefore to save space we will present only some indicative examples. For the Error Handling attribute the documentation and samples of each component were reviewed in order to find out whether an exceptions class was available, which would allow for catching operation errors of the component and display proper messages to the user. None of the components offered this feature and this was verified later when operating the components. Regarding customization effort the level of satisfaction was decided after trying to customize the component with most of its advertised features.

The next and most important step after we prepared the list was to identify areas of improvement and/or areas which were not addressed at all by the competition in order to gain an advantage. It was obvious from the evaluation list that the competition was very strong in the field of charting components. All four components perform very well at the most important quality characteristics (functionality and efficiency) and this implies that a lot of work was needed in order to stay in touch with the competitive products. Some areas of improvement were identified at the usability characteristic, especially for the quality of the user manual and the installation and administration documentation and also at the operation/customizability/administration effort where except from one product (TeeChart) the others do not fare very well. However, only these enhancements could not truly make the difference and drive customers to select our product.

A closer look to the evaluation list indicated that there were additional quality areas that were not addressed at all by the competition like data encryption, error handling, recoverability, support tools and debugging. This by no means implies that if these quality characteristics were added to our component then automatically would gain a competitive advantage. On the contrary, in cases like this it must be very carefully evaluated whether there will be potential customers that need these extra characteristics and whether these characteristics will be really useful for the component to be developed. For example, do charting components need enhanced security based on encryption? Most probably not, but on the other hand with the increased security concerns that we have nowadays this might prove to be a good move. In addition, we also need to have in

Table 6
Checklist with weights for Case study 2

Characteristics	Sub-characteristics	Attributes	Metrics	Weights (%)
Functionality	Interoperability	Platform Independence	Number of platforms supported	2
		OS Independence	Number of OS supported	5
		Data open-format compatibility	Provided or Not	3
	Completeness	Service Satisfaction	Functions Ratio	10
	Security	Resistance to privilege escalation	Provided or Not	3
		Data Encryption	Provided or Not & Number of encryption algorithms	2
Reliability	Service Stability	Error Prone	Number of errors/crashes per unit of time	4
		Error Handling	Provided or Not	3
		Recoverability	Provided or Not	3
	Result Set	Correctness	Ratio of correct results to total results	5
Usability	Learnability	Time to use	Average time to learn how to use	2
		Time to configure	Average time to learn how to configure	2
		Time to administer	Average time to learn how to administer	2
	Help tools	Help completeness	Provided or Not – Level of satisfaction	1
		Users Manual	Provided or Not – Level of satisfaction	1
		Installation and Administration Documentation	Provided or Not – Level of satisfaction	2
		Support Tools	Provided or Not	2
	Operability	Operation effort	Level of satisfaction	2
		Customizability effort	Level of satisfaction	2
		Administration effort	Level of satisfaction	2
	Identifiability-Reachability	Directory Listing	Number of popular directory listings	1
		Search & Retrieve	Level of satisfaction	1
		Explainability	Level of satisfaction	1
Efficiency	Response Time	Throughput	Number of requests per unit of time	6
		Capacity	Number of concurrent users	5
	System Overhead	Memory Utilization	RAM size	6
		Processor Utilization	Level of processor speed	5
		Disk Utilization	Secondary memory size	2
Maintainability	Changeability	Upgradeability	Level of satisfaction	3
		Debugging	Provided or Not	2
		Backward compatibility	Provided or Not	3
	Testability	Trial version	Level of satisfaction	2
		Test Materials	Level of satisfaction	2
	Customizability	Parameterization	Black-box: Ratio of changeable parameters to total parameters White-box: Level of satisfaction	3

mind that the weights placed on these quality characteristics were more to the low-end than to the high-end and therefore it is difficult to safely determine whether they will make the difference.

The next step involved the actual implementation of the component. In our case, though, the implementation did not proceed as it was beyond the purpose of demonstrating the use of the proposed framework. In other cases the new component may be built, provided of course that the management makes the decision that there is still room to compete in this area. The quality framework was used as a reference in order to ensure that no important quality characteristics will be left out, competition metrics are met or surpassed and niche areas are added to give the competitive edge. Once the component is build then the quality framework, along with the relevant weights, is re-applied on all the competitive components and a final assessment is made. If expectations are not met then an iterative process can be used until the desired component is developed.

3.2. Validation of the methodology

The aim of this subsection is to validate the efficiency of the proposed methodology in terms of accuracy on one hand, and time/effort on the other. In this context, the developers that participated in defining the list of must-have quality criteria were asked to take part in a final validation procedure. According to this procedure, each developer would assess the quality of the candidate components used in the SMS exchange example of the previous subsection, and select the most appropriate one using the proposed methodology. The purpose of this validation round was twofold: Firstly, to identify possible drawbacks or weaknesses in applying the methodology by experts in the CBSE field, and secondly, to collect and compare data on the time and effort required in practice by different people.

Seven developers responded positively, the rest thirteen decided to withdraw from the experiment due to other obligations and work deadlines. The first step of the validation procedure was to train developers how to apply the framework, starting with describing in detail the meaning of the

Table 7
Checklist with scores for Case study 2

Attributes	ComponentOne	Infragistics NetAdvantage	Dundas Chart	TeeChart Pro ActiveX
Platform Independence	33.3% (1/3)	33.3% (1/3)	33.3% (1/3)	33.3% (1/3)
OS Independence	20% (1/5)	20% (1/5)	20% (1/5)	20% (1/5)
Data open-format compatibility (export in various formats – data sources supported)	Provided	Provided	Provided	Provided
Service Satisfaction (# of desired functions)	15/15	15/15	14/15	14/15
Resistance to privilege escalation	Provided	Provided	Provided	Provided
Data Encryption	Not Provided	Not Provided	Not Provided	Not Provided
Error Prone (number of errors per 5 h of usage)	1	2	1	0
Error Handling	Not Provided	Not Provided	Not Provided	Not Provided
Recoverability	Not Provided	Not Provided	Not Provided	Not Provided
Correctness (ratio of correct results)	99%	98%	98%	98%
Time to use	4 h	4 h	2 h	3 h
Time to configure	1 h	<1 h	<1 h	1 h
Time to administer	<1 h/day	<1 h/day	<1 h/day	<1 h/day
Help completeness	Provided – 5	Provided – 5	Provided – 4	Provided – 4
Users Manual	Provided – 4	Provided – 4	Provided – 3	Provided – 3
Installation and Administration Documentation	Provided – 3	Provided – 3	Provided – 3	Provided – 3
Support Tools	Not provided	Not provided	Not provided	Provided
Operation effort	3	4	3	5
Customizability effort	3	4	3	5
Administration effort	4	4	4	5
Directory Listing	80% (4/5)	80% (4/5)	80% (4/5)	60% (3/5)
Search & Retrieve	5	5	5	5
Explainability	4	4	4	5

quality attributes and explaining how to apply the relevant metrics, and finishing with analyzing the specific steps of the methodology. The second step involved the application of the methodology and the recording of the time/effort required to conclude the assessment and select the winning component. The third and final step included a review approach towards the proposed methodology taken by the participating developers, commenting on possible difficulties and offering overall impressions.

Table 8 summarizes the results of the validation procedure. According to the first row, the average time spent on learning how to use the proposed framework did not exceed the 3.5 man hours. The difference observed in this time duration between developers was sourced primarily by the different levels of familiarity with the distinct quality characteristics and by the fact that in the D4, D6 and D7 cases the training course was interrupted several times due to unexpected tasks that had to be taken care urgently by the corresponding developer. Thus, the time to resume knowledge gained thus far added an overhead to the total learning time. The second row of Table 8 lists the time effort put by each developer. It is worth noting here that the requirements of the desired component, the candidate components, as well as the weights for the quality attributes in the checklist were the same as those used in our example Case 1, so as to achieve uniformity and objectivity in the results. One can easily discern a variation in man hours devoted to the components' quality assessment and selection, with time ranging from 5 to 9.5 man hours. This variation was mostly the result of the different effort devoted to assess the attribute that required a satisfactory level metric (Service Satisfaction), while effort was also affected by the programming experience of the developers involved.

The results of the quality assessment of all seven developers indicated the ActiveSMS component as the most appropriate for selecting and integrating to the system, something which is consistent with our previous findings. The developers were then asked to provide their comments on the methodology followed, both positive and negative, as well as if they would adopt the proposed framework to use during their everyday development tasks. Their answers and statements are summarized in the third row of Table 8. More specifically, all seven developers recognized the value of the methodology and appreciated the support it offered. Some of them were very satisfied by the systematic approach and accuracy offered, while some commented negatively the effort needed to carry out the assessment. Five developers stated that they would like to use it from that time onwards, changing their current (usually ad-hoc) components' quality assessment procedures, while it is worth mentioning that one of them was among those who expressed earlier their concerns about the time required by the methodology to conclude. As he stated, the time overhead introduced is counterbalanced by the safety he felt that the assessment is correct and accurate. Two of the developers stated that they preferred to continue using their own ad-hoc methods due to the shorter length and effort required. They think that their rich experience in the field is the key to easily identify major or minor weaknesses of candidate components and therefore they do not need such a methodology.

Table 8
Validation results

	Time to learn and use the methodology (man hours)	Time/effort spent man (hours)	Review comments
D1	2.5	6	Very supportive and accurate – Will adopt
D2	3	8	Good and balanced approach – Rather time consuming – Will adopt
D3	2.4	7.5	Difficult at first but very helpful – Will adopt
D4	4	9.5	Time consuming but accurate – Will not adopt
D5	3.5	7	Systematic, organized approach – Requires significant effort but better than existing practice – Will adopt
D6	4.5	5	Good approach – Will not adopt, continue with current approach
D7	4	8.5	Time consuming – Will not adopt, continue with existing process

Summarizing the validation procedure we may reach to the following concluding remarks:

- The proposed quality framework (model and methodology) can indeed be used in practice presenting no difficulties or drawbacks.
- Experts in the CBSE area were in general very pleased and contented with the systematic approach taken by the methodology and the accuracy of its results.
- The time overhead introduced by the methodology is not high; therefore it does not constitute a negative factor for its global acceptance. Of course, time and effort depends on the complexity of the component searched for, the criticality of the system to be integrated to, and the number of quality attributes included in the checklist by the individual developer-evaluator. Thus, in some cases this time and effort is minimal, while in other cases it may be enormous. This, though, is not something that is sourced by the steps of the proposed methodology, which are simple and easy to execute.
- The most difficult part of the methodology is the assessment of attributes that carry a satisfaction level metric. As it was revealed by the validation procedure, the time needed to assess such an attribute cannot be defined á-priori; it is quite subjective and depends on the attribute itself, the significance (weight) of the particular attribute and the experience of the developer-evaluator. The empirical evidences showed that a medium to high significance attribute may take from 0.5 to 2 man hours to be evaluated by an average experience developer. This, though, may be conceived only as an indication.

4. Discussion and conclusions

In this paper we have developed a quality framework for *evaluating original components* along with a methodology to be applied in order to facilitate their evaluation. Although there have been some attempts so far to address the quality issues of original components, to the best of our knowledge, our quality framework and accompanied methodology of applying the framework is one positive step to the right direction. It represents a straightforward and effective framework, which can be applied within reasonable amount of time. Utilizing the ISO9126 as a solid reference, we tried to touch upon every aspect of CBSE and identify the areas that need a quality perspective. During this process it was evident that the most difficult task was to define appropriate metrics for each attribute and moreover to arrive to a single result that points the most qualified component. The adversity of the metrics, not only in terms of their definition but also in terms of their measurement units, makes this task practically impossible. However, we described thoroughly in Section 3 how our quality model can be applied and we proposed some criteria that we consider as the minimum acceptable for assessing the quality of a component. Our evaluation examples showed that the model can greatly assist in isolating the most suitable components, but sometimes a developer's call is necessary. Additionally, our case studies suggested that the proposed framework may significantly enhance the development process of original components.

The time, effort and human resources needed for applying a framework always depends on the importance of the component to the software system to be developed. Component evaluation in general may range from one developer testing for 1 day, to a pool of developers testing for a month. Broadly speaking, with the exception of the performance and error tests and the tests that are measured as a level of satisfaction, the application of the proposed framework for "simple" components normally should not take more than one or two days. This was the case with our practical experiments conducted with the aid of seven developers acted as experts from the field of CBSE to demonstrate the applicability of our framework. For the exceptions, it is up to the user to decide how long a component should be tested to achieve the desired satisfaction. Of course all these are valid provided that the component supplier provides a full trial version for evaluation along with any necessary documentation regarding the features and the usage of the component.

Unfortunately, the story for the quality of original components does not end here. There is a lot of work needed to be done the most important of which is to convince component vendors that adopting a quality framework will eventually benefit them. Unless a consensus is reached on this issue, the development of CBSE will be severely handicapped and will soon be overshadowed by other promising methodologies for software development such as agent-oriented software engineering. Our future work will primarily target at enhancing the proposed model by new or/and more detailed metrics where possible. As the software technology advances, different reuser needs emerge that need to be addressed in terms of quality characteristics, as for example with mobile and pervasive computing. Furthermore, we plan to develop an XML schema for providing the ability to evaluate the framework in a world-wide sense. To be more specific, we are currently working on implementing a dedicated web application, which will offer free access to the proposed quality model, along with the corresponding guidelines for its practical use. Potential users of the model will be required to provide a certain structured form of feedback in terms of component quality evaluation results and difficulties experienced when applying the proposed framework via XML. Thus, developers/re-users will be able to share their evaluations in a structured manner.

The final piece of future work will concentrate on a formal evaluation of the proposed framework (quality model and method for its application) using the concepts described in the Desmet project [21]. This will enable the identification of possible gaps from the application point of view, which then may lead to appropriate modifications and revisions where appropriate.

References

[1] L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, K. Wallnau, Volume I: Market Assessment of Component-Based Software Engineering, Software Engineering Institute, Technical Note CMU/SEI- 2001-TN-007, 2001.

[2] L. Baum, M. Becker, Generic components to foster reuse, in: Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-PACIFIC) 2000, pp. 266–277.

[3] BBN Corporation, Quality Objects Project. URL: <http://www.dist-systems.bbn.com/tech/QuO>, 2001.

[4] A.D. Belchior, A Fuzzy Model to Software Quality Evaluation, University of Fortaleza, Doctoral Thesis, 1997.

[5] M.F. Bertoa, A. Vallecillo, Quality attributes for COTS components, in: Proceedings of the Sixth ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002), Malaga, Spain, 2002, pp. 54–66.

[6] B.W. Boehm, J.R. Brown, J.R. Kaspar, M. Lipow, G. MacCleod, Characteristics of Software Quality, North Holland, Amsterdam, 1978.

[7] Component One. <http://www.componentone.com>.

[8] M. Dehlin,S. Yacoub, A. Mili, C. Kaveri, A Model for Certifying COTS Components for Product Lines, Workshop on Continuing Collaborations for Successful COTS Development, in Conjunction with the 22nd International Conference on Software Engineering (ICSE2000) Limerick, Ireland, June 4–11, 2000.

[9] Dundas Chart. <http://www.dundas.com>.

[10] D. Garvin, What does 'product quality' really mean? Sloan Management Review (1984) 25–45.

[11] J. Gao, Component testability and component testing challenges, in: Proceedings of the Third ICSE Workshop on Component-based Software Engineering, 2000.

[12] G. Brahnmath, R.R. Raje, A. Olson, B. Bryant, M. Auguston, C. Burt, A quality of service catalog for software components, in: Proceedings of the Southeastern Software Engineering Conference, Huntsville, Alabama, April 2002, pp. 513–520.

[13] M.A. Goulão, A.F. Brito, The Quest for Software Components Quality, COMPSAC'2002, Oxford, England. August 2002, pp. 313–320.

[14] M.A. Goulão, A.F. Brito, Towards a Component Quality Model, Work in Progress Session of the 28th EUROMICRO Conference, Dortmund, Germany, September, 2002.

[15] GSMActive. <http://www.gsmactive.com>.

[16] W.J. Hansen., An original process and terminology for evaluating COTS software. <http://www.sei.cmu.edu/staff/wjh/Qesta.html>, August 2001.

[17] Infragistics NetAdvantage. <http://www.infragistics.com>.

[18] Intellisoftware-ActiveSMS. <http://www.intellisoftware.co.uk/ActiveSMS>.

[19] ISO/IEC-9126. Information technology – Software product evaluation – Quality characteristics and guidelines for their use, International Standard ISO/IEC 9126, International Standard Organization, December 1991.

[20] P. Kallio, E. Niemela, Documented quality of COTS and OCM components, in: Proceedings Fourth ICSE Workshop on Component-Based Software Engineering, May 14–15, 2001 Toronto, Canada, pp. 111–114.

[21] B. Kitchenham, DESMET: A Method for Evaluating Software Engineering Methods and Tools, Technical Report TR96-09, University of Keele, UK, 1996.

[22] B. Korel, Black-Box Understanding of COTS Components, in: Proceedings of the Seventh International Workshop on Program Comprehension, IEEE Press, May 05–07, 1999, pp. 226–233.

[23] J.A. McCall, P.K. Richards, G.F. Walters, Factors in Software Quality, National Technical Information Service, Springfield, VA, 1977, vol. 1, 2 and 3.

[24] C. Mueller, Korel B., Automated black-box evaluation of COTS components with multiple-interfaces, in: Proceedings of the Second International Workshop on Automated Program Analysis, Testing, and Verification (ICSE2001), Toronto, Canada, 2001.

[25] S.L. Pfleeger, Software Engineering Theory and Practice, Second ed., Prentice-Hall, New Jersey, 2001.

[26] O. Preiss, A. Wegmann, J. Wong, On quality attribute based software engineering, in: Proceedings of the 27th Euromicro Conference, 2001, pp.114–120.

[27] R. Raje, B. Bryant, M. Auguston, A. Olson, C. Burt, A unified approach for the integration of distributed heterogeneous software components, in: Proceedings of the 2001 Monterey Workshop Engineering Automation for Software Intensive System Integration, Monterey, California, 2001, pp. 109–119.

[28] D.S. Rosenblum, Adequate Testing of ComponentBased Software, Department of Information and Computer Science, University of California, Irvine, CA, Technical Report 97-34, August 1997.

[29] J. Sametinger, Software Engineering with Reusable Components, Springer, 1997.

[30] R.P.S. Simao, Quality Characteristics for Software Components, University of Fortaleza, Master's Thesis, 2002.

[31] V. Seppanen, N. Helander, E., Seija Komi-Sirviö, Original Software Component Manufacturing: Survey of the State-of-the-Practice. EUROMICRO 2001, pp. 138–145.
[32] SMS Demon. <http://www.dload.com.au/home/home.cfm>.
[33] SMSZyneo. <http://www.zyneo.com>.
[34] C. Szyperski, Component Object-Oriented Programming, Addison-Wesley, 1998.
[35] TeeChart Pro ActiveX. <http://www.teemach.com>.
[36] R.M.B. Villela, Components Searching and Retrieving in Software Reuse Environments, Doctoral Thesis, 2000.