

# Quality Attributes for COTS Components

Manuel F. Bertoa y Antonio Vallecillo

*Departamento de Lenguajes y Ciencias de la Computación*

*Universidad de Málaga. 29071 Málaga, España*

{bertoa,av}@lcc.uma.es

## Abstract

As Component-based Software Development (CBSD) starts to be effectively used, some software vendors have commenced to successfully sell and licence commercial off-the-shelf (COTS) components. One of the most critical processes in CBSD is the selection of the COTS components that meet the user's requirements. Current proposals have shown how to deal with the functional aspects of this evaluation process. However, there is a lack of appropriate quality models that allow an effective assessment of COTS components. Besides, the international standards that address the software products' quality issues (in particular, those from ISO and IEEE) have shown to be too general for dealing with the specific characteristics of software components. In this paper we propose a quality model for CBSD based on ISO 9126, that defines a set of quality attributes and their associated metrics for the effective evaluation of COTS components.

## 1 Introduction

In the last decade, *Component-Based Software Development* (CBSD) has generated tremendous interest due to the development of plug-and-play reusable software, which has led to the concept of *commercial off-the-shelf* (COTS) software components. This approach moves organizations from application development to application assembly. Constructing an application now involves the use of prefabricated pieces, perhaps developed at different times, by different people, and possibly with different uses in mind. The ultimate goal, once again, is to be able to reduce development times, costs, and efforts, while improving the flexibility, reliability, and reusability of the final application due to the (re)use of software components already tested and validated.

In CBSD, the proper search and selection processes of COTS components have become the cornerstone of any effective COTS development. So far, most of the Software Engineering community have concentrated on the functional aspects of components, leaving aside the (difficult) treatment of their quality and extra-functional properties. However, this kind

of properties deserve special attention, since they are essential in any commercial evaluation process.

There are several reasons that difficulty the effective consideration of the extra-functional and quality requirements of software components. First, there is no general consensus on the quality characteristics that need to be considered. Thus, different authors propose different (separate) classifications: McCall's quality factors proposed in 1977 [12], Barry Boehm's quality model presented in 1976 [3], the quality attributes proposed by international standards ISO 9126 [11] and ISO 14598 [10], the list of quality attributes used in the COCOTS model [1]—which is based on IEEE standards [6]—, and many others [4, 14].

The next issue is the lack of information about quality attributes provided by software vendors. The Web portals of the main COTS vendors show this fact. Visit for instance Componentsource ([www.componentsource.com](http://www.componentsource.com)), Flashline ([www.flashline.com](http://www.flashline.com)), or WrldComp ([www.wrldcomp.com](http://www.wrldcomp.com)).

In addition to this, there is an absence of any kind of metrics that could help evaluating quality attributes objectively. Even worse, the international standards in charge of defining and dealing with the quality aspects of software products (eg. ISO 9126 and ISO 14598) are currently under revision. The SquaRE project [2] has been created specifically to make them converge, trying to eliminate the gaps, conflicts, and ambiguities that they currently present.

Another drawback of the existing international standards is that they provide very general quality models and guidelines, but very difficult to apply to specific domains such as CBSD and COTS components.

In order to address many of these issues, this paper tries to provide a quality model specific for COTS components. Focusing on a very concrete domain, it builds on the existing approaches and proposes a set of quality attributes and their corresponding metrics.

This paper is organized in 7 sections. After this introduction, section 2 introduces the terminology used, as well as an initial classification of the quality characteristics of software products. Section 3 discusses the ISO 9126 model, and shows how the quality

characteristics it defines do not perfectly match the particular needs of COTS components. Our proposal is described in section 4, in which a refinement of the ISO 9126 quality model is defined. Then, section 5 proposes the use of XML for documenting documenting component attributes. Finally, sections 6 and 7 discuss some related works, draw some conclusions, and outline future research activities.

## 2 Components Quality Characteristics

In general, there is no consensus on how to define and categorize software product quality characteristics. Here we will try to follow as much as possible an standard terminology, in particular the one defined by ISO 9126. In it, a *quality characteristic* is 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*.

An *attribute* is a quality property to which a metric can be assigned, where a *metric* is a procedure for examining a component to produce a single datum, either a symbol (e.g. Excellent, Yes, No) or a number. Please note that not all properties are measurable (e.g. Demonstrability).

A *Quality model* is the set of characteristics and sub-characteristics, as well as the relationships between them, that provide the basis for specifying quality requirements and for evaluating quality. Of course, the quality model used will depend on the kind of target product to be evaluated. In this sense, the current standards and proposals define “generic” quality models.

The main contribution of this exercise is the definition of a quality model specific for software components, which is described in section 3. Our main goal is to define the attributes that can be described by COTS vendors (no matter whether they are internal or external providers) as part of the information provided about them. These attributes will allow the COTS components’ assessment and selection by software designers and developers.

Before we start, we need to define what a software component is. Here we will adopt Szyperski’s definition, whereby components are binary units of possibly independent production, acquisition and deployment that interact to form a functioning system [14]. The adjective COTS will refer to a special kind of (usually large grained) components, which are specially designed, developed and marketed to be used in CBSD environments.

Table 1 shows the characteristics and sub-characteristics which define the ISO 9126 general software quality model. From this quality model, our idea is to refine and customize it in order to accommo-

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

Table 1: ISO 9126 Quality Characteristics

date to the particular characteristics of COTS components.

The first step is to identify several kinds of quality characteristics, classifying them according to different criteria.

1. First, we need discriminate between those characteristics that make sense for individual components (that we will call *local* characteristics) and those that must be evaluated at the software architecture level (*global* characteristics). For instance, Fault Tolerance is a typical quality characteristic that depends on the software architecture of the application. On the contrary, Serialiable is a property applicable to individual components only.
2. The moment in which a characteristic can be observed or measured also allows to establish another classification. Thus, we have those characteristics observable at runtime (e.g. Performance) and those observable during the product cycle-life (e.g. Maintainability) [13].
3. It is also important to identify the target users of the quality model, as ISO standards explicitly states. In our case, these users are mainly software architects and designers, which need to evaluate the COTS components available in software repositories (or that can be bought from software components vendors) in order to be incorporated into the software product they are building. In this sense, we are focused more

on the “programmatic” interfaces of components than on their “user” (GUI) interfaces, i.e., we are particularly concerned with the API’s defining the services provided by the components so they can be composed and integrated with other programs.

4. For COTS components, it is essential to distinguish between internal and external metrics. Internal metrics measure the internal attributes of the product (e.g. specification or source code) during design and coding phases. They are “white-box” metrics. On the other hand, external metrics concentrate on the system behavior during testing and component operation, from an “outsider” view. External metrics are more appropriate for COTS components, due to its “black-box” nature. However, internal metrics cannot be completely discarded, since some internal attributes of a component may provide an indirect measurement of its external characteristics. Similarly, they may even affect the final architecture’s properties. For example, the size of a component can be important when taking care of the final application space (eg. memory) requirements.

Finally, it is important to note that there are other kind of marketing characteristics such as price, technical support, license conditions, etc.—not directly related to quality—that may be of great importance when selecting components. In this paper we will concentrate on quality characteristics only, leaving the rest of characteristics for further research.

### 3 Quality characteristics

As previously mentioned, not all the characteristics of a software product as defined by ISO 9126 are applicable to COTS components. Table 2 shows the quality model we propose for this kind of components.

As we can see, it is basically the ISO quality model (see Table 1), where the *Portability* and *Fault tolerance* characteristics disappear, as well as the *Stability* and *Analyzability* sub-characteristics. Two new sub-characteristics also appear: *Compatibility* and *Complexity*. Besides, other characteristics (shown in bold) have changed their meaning in this new context. The following list discusses the main changes to the ISO 9216 proposal.

**Functionality** This characteristic maintains the same meaning for components than for software products. It tries to express the ability of a component to provide the required services and functions, when used under the specified conditions.

The sub-characteristic *Compatibility* has been added in our model, which indicates whether former versions of the component are compatible with its current version, i.e., whether the component could work when integrated in a context where a prior version correctly worked.

**Reliability** This characteristic is directly applicable to components, and essential for reusing them. The *Maturity* sub-characteristic is measured in terms of the number of commercial versions and the time intervals between them. On the other hand, *recoverability* tries to measure whether the component is able to recover from unexpected failures, and how it implements these recovery mechanisms.

**Usability** This characteristic and all its sub-characteristics are perhaps the best example of characteristics that have a completely different meaning for software components. The reason is that, in CBSD, the end-users of components are the application developers and designers that have to build applications with them, more than the people that have to interact with them. Thus, the usability of a component should be interpreted as its ability to be used by the application developer when constructing a software product or system with it.

Under this characteristic we have included attributes that measure the component’s usability during the design of applications, incorporating *Complexity* as a new sub-characteristic that provides a measure of the component’s complexity when integrating and using it within a software product or system.

**Efficiency** We will respect the definition and classification proposed by ISO 9126 (which distinguishes between *Time behavior* and *Resource behavior*), although many people prefer to talk about *Performance* and use other sub-classifications. In any case, the attributes we have identified for this characteristic are applicable independently of the name or sub-classification used.

**Maintainability** This characteristic describes the ability of a software product to be modified. Modifications include corrections, improvements or adaptations to the software, due to changes in the environment, in the requirements, or in the functional specifications. The user of a component (ie. the developer) does not need to do the internal modifications but (s)he does need to adapt it, re-configure it, and perform the testing of the component before it can be included in the

Characteristics	Sub-characteristics (Runtime)	Sub-characteristics (Life cycle)
Functionality	Accuracy Security	Suitability Interoperability Compliance <i>Compatibility</i>
Reliability	Recoverability	Maturity
<b>Usability</b>		<b>Learnability</b> <b>Understandability</b> <b>Operability</b> <i>Complexity</i>
Efficiency	Time behavior Resource behavior	
Maintainability		Changeability testability

Table 2: Quality Model for COTS components

final product. Thus, *changeability* and *testability* are defined as sub-characteristics that must be measured for components.

**Portability** This characteristic is defined as the ability of a software product to be transferred from one environment to another. In CBSD, portability is an intrinsic property to the nature of components, which are in principle designed and developed to be re-used in different environments (it is important to note that in CBSD, re-use means not only to use more than once, but also to use in different environments[14]).

## 4 Component Attributes

Once we have discussed the general ISO 9126 quality model, in this section we will describe our proposal, i.e. the quality attributes we propose for measuring the characteristics of COTS components. Quality attributes will be divided into two main categories, depending on whether the attributes are discernible at run-time, or observable during the product life cycle.

The metrics that will be used for measuring attributes are the following:

**Presence** This metric identifies whether an attribute is present in a component or not. It consists of a boolean value and a string. The *boolean* value is used to indicate whether the attribute is present and, if so, the *string* describes how the attribute is implemented by the component.

Examples of attributes that are measured by this metric are Data Encryption or Serializable.

**Time** This metric is used to measure time intervals. It uses an integer type variable to indicate the absolute value, together with a string variable that indicates the units (seconds, months, etc.)

**Level** This metric is used to indicate a degree of effort, ability, etc. It is usually a subjective measure. It is described by an integer variable that can take any of the following values: 0 (Very Low), 1 (Low), 2 (Medium), 3 (High), 4 (Very High).

**Ratio** This metric is used to describe percentages. It is measured by an integer variable with values between 0 and 100.

Apart from these metrics, *indexes* will be used too. Indexes are derived measures, calculated from the values of two basic attributes, generating what is called an “indicator”. For example, the Complexity Ratio is an attribute that compares the number of configurable parameters of the component with the number of its provided interfaces. In general, indicators need to be distinguished from basic metrics because the former ones are derived from the latter. Although it could be argued that they are disposable, their expressiveness has moved us to include them in our quality model.

## 4.1 Attributes measurable at runtime

Table 3 shows the quality attributes for COTS components observable during execution, grouped by sub-characteristics and indicating the kind of metric they use.

### 4.1.1 Attributes associated to “Accuracy”

#### – Precision

This attribute evaluates the percentage of results obtained with the precision (ie. granularity) specified by the user requirements.

Please note that this attribute not only allows us to measure the computational precision of the operations performed by the component, but also it can be used for measuring the level of “freshness” of the information returned by the called operations. For instance, this is the case of a component which returns data from a cache in order to improve performance, at cost of returning information not completely up-to-date.

This attribute is measured by a *Ratio* variable, calculated by dividing the number of adequate results returned by the total number of results obtained in a given series of calls.

#### – Computational Accuracy

This attribute evaluates the number of accurate results returned by the component operations, according to the user specifications. It is measured by a *Ratio* variable, calculated by dividing the number of accurate results returned by the total number of results obtained in a given series of calls.

### 4.1.2 Attributes associated to “Security”

- **Data Encryption** This attribute expresses the ability of a component to deal with encryption in order to protect the data it handles. A *Presence* metric is used, indicating the encryption method(s) used if so.

#### – Controllability

This attribute indicates how the component is able to control the access to its provided services. Examples are components that provide interfaces with functionality to identify or authenticate users. A *Presence* metric will be used, indicating the control mechanisms implemented.

#### – Auditability

This attribute shows whether a component implements any auditing mechanism, with capabilities for recording users access to the system and to its data. For instance, the component may

provide functionality for recording each operation performed by its users, together with its related data, access date, etc. A *Presence* metric will be used to measure this attribute, indicating how operations are recorded and retrieved.

### 4.1.3 Attributes associated to “Recoverability”

#### – Serializable

This attribute denotes the ability of a component to serialize its code and state, so it can be transferred to a different machine, or stored for persistency. A *Presence* metric will be used to measure this attribute.

#### – Persistent

This attribute indicates whether a component can store its state in a persistent manner for later recovery. A *Presence* metric will be used to measure this attribute.

#### – Transactional

This attribute indicates whether the component provides any interface for implementing transactions with its operations. For instance, a CORBA component implementing the “Resource” interface. A *Presence* metric will be used to measure this attribute.

#### – Error Handling

This attribute indicates whether the component can handle error situations, and the mechanism implemented in that case (eg. exceptions). A *Presence* metric is used, in which the string variable describes the level of error handling implemented. Examples of this are:

**Detection** Errors are detected but no corrective actions are taken

**Detection and warning** Errors are detected and a warning is generated

**Handling** Errors are detected and an exception mechanism is implemented (the implemented mechanism is also described)

### 4.1.4 Attributes associated to “Time behavior”

We need to differentiate here between those attributes which are appropriate for measuring discrete values, and those appropriate for data streaming. In the first group we will define the following attribute:

#### – Response Time

This attribute can be associated to any of the methods implemented in any of the component

interfaces, and measures the time taken since a request is received until a response has been sent. Since this time may depend on the input parameters, the best, worst, or average time has to be specified.

In case this attribute is associated to an interface, then the best (worst or average) response time will indicate the best (worst or average) response time of the interface's operations. It is measured by a *Time* metric.

Two attributes deal with data streaming:

– **Throughput**

This attribute measures the output that can be successfully produced over a given period of time. An *Integer* value is used to record this attribute, together with a string with the units used.

– **Capacity**

This attribute measures the amount of input information that can be successfully processed by the component over a given period of time. An *Integer* value is used to record that attribute, together with a string with the units used.

**4.1.5 Attributes associated to “Resource behavior (Resource utilization)”**

– **Memory utilization**

The amount of memory needed by a component to operate. This attribute is a very relevant factor in embedded and other critical systems. An *Integer* variable is used to determine the number of kilobytes required. Besides, the minimum, maximum or estimated memory size may be indicated (or a recommended size for optimum performance).

– **Disk utilization**

This attribute specifies the disk space used by a component, including both the space used for storing its code and constituent parts, and the space used temporarily or permanently during the execution. An *Integer* variable indicates the number of kilobytes required.

**4.2 Attributes measurable during Component Life Cycle**

The quality attributes for COTS components observable during life cycle are detailed in the following sections and are summarized in Table 4, grouped by sub-characteristic.

**4.2.1 Attributes associated to “Suitability”**

The level of suitability tries to express how well the component fits the user's requirements. Suitability will be measured by dividing the number of user-required interfaces by the total number of interfaces provided by the component, ie. how many of the provided interfaces are really required.

This attribute will depend on the user requirements for the specific application, and therefore it cannot be directly measured by the component provider.

– **Coverage**

This attribute tries to measure how much of the required functionality is covered by the component implementation. It is measured by a *Ratio* metric according to the following formula:

$$100 * \frac{Needed\ Interfaces \cap Provided\ Interfaces}{Needed\ Interfaces}$$

– **Excess**

This attribute relates the number of the interfaces effectively used in the application with the number of interfaces provided by the component. A *Ratio* metric is used according to the following formula:

$$100 * \frac{Used\ Interfaces \cap Provided\ Interfaces}{Provided\ Interfaces}$$

– **Service implementation coverage**

It is possible that some component implementations do not completely cover the services specified by the standard. This attribute tries to measure the number of implemented operations compared to the total number of specified operations.

For instance, some CORBA or EJB component vendors provide component implementations that do not cover the whole functionality as described in the service specification. This attribute serves to measure these cases. It is measured by a *Ratio* metric according to the following formula:

$$100 * \frac{Implemented\ Operations}{Specified\ Operations}$$

**4.2.2 Attributes associated to “Interoperability”**

– **Data compatibility**

This attribute indicates whether the format of the data handled by the component is compliant

Sub-characteristic	Attribute	Type
Accuracy	1. Precision	Ratio
	2. Computational Accuracy	Ratio
Security	3. Data Encryption	Presence
	4. Controllability	Presence
	5. Auditability	Presence
Recoverability	6. Serializable	Presence
	7. Persistent	Presence
	8. Transactional	Presence
	9. Error Handling	Presence
Time behavior	10. Response time	Time
	11. Throughput	Integer
	12. Capacity	Integer
Resource behavior	13. Memory utilization	Integer
	14. Disk utilization	Integer

Table 3: Quality attributes for COTS components which are measurable at runtime

with any international or 'de facto' standard or convention. The idea is to determine whether the component handles open or proprietary data formats. A *Presence* metric indicates this ability and, if so, which standard format is used (eg. ASN1, XML, etc.)

#### 4.2.3 Attributes associated to “Compliance”

##### – Standardization

This attribute indicates the component conformance to international standards. We will use a *Presence* metric to indicate whether the component conforms to an international standard, convention or regulation or not and, if so, to which ones.

##### – Certification

Apart from conforming to a standard, the component may be certified by any internal or external organization (eg. X/open). This attribute uses a *Presence* metric to indicate whether it has some kind of certification.

#### 4.2.4 Attributes associated to “Compatibility”

##### – Backwards Compatibility

This attribute is used for indicating whether the component is backwards-compatible with its previous versions or not. If so, this means that the new component can substitute the previous ones without being noticed by its prior clients. A *Presence* metric is used to measure this attribute. In case of backwards-compatibility, the string variable contains the numbers of the previous versions for which the compatibility is maintained.

#### 4.2.5 Attributes associated to “Maturity”

The component maturity is measured in terms of the number of commercial versions that have been marketed, and the interval between those versions.

##### – Volatility

This attribute measures the average time between commercial versions, and we also refer to it as “Mean Time Between Versions”. We have named this attribute Volatility because it give us an indication about the life time of a version in the market. A *Time* metric is used for measuring this attribute.

##### – Evolvability

The number of versions that have been marketed for a given component may provide an indication about the product maturity, and how it has evolved from its first version. This attribute may be interesting in conjunction with Volatility to indicate whether new versions are expected in a short time or not.

This attribute is measured by an *Integer* value that stores the absolute number of versions that have been commercially produced.

##### – Failure removal

A Maturity indicator is the number of bugs fixed in a given version of the component. Although a priori a high number of bugs fixed in a version could indicate that the new version is more stable, the authors' experience shows that the more bugs discovered in a product, the more bugs remain to fix. Thus, the number of fixed bugs is a good measure of the bugs that still remain hidden. An *Integer* is used to store the average number of bugs fixed in all previous versions.

#### 4.2.6 Attributes associated to “Learnability”

Related to this characteristic, there is a set of attributes that try to measure the time and effort needed to master some specific tasks (such as usage, configuration, or administration of the component). These times, initially assigned by the component vendor or estimated by third parties, provide an estimation of the learnability of the component. In this context, the times will correspond to technical staff, with an average experience and knowledge (ie. neither a novice nor an expert).

- **Time to use**

This attribute measures the average time needed for a developer to learn how to correctly use the component.

- **Time to configure**

This attribute measures the average time needed for a developer to learn how to correctly configure the component, and for properly understanding its configuration parameters.

- **Time to admin**

This attribute measures the average time needed for a developer or system administrator to learn how to correctly administer the component.

- **Time to expertise**

This attribute measures the average time needed for a developer for mastering all the functionality and possibilities offered by the component.

#### 4.2.7 Attributes associated to “Understandability”

These attributes deal with the component documentation, descriptions, demos, and tutorials available, which have a direct impact on the understandability of the component.

It is important to notice that this characteristic is closely related to Learnability, since in order for an entity or service to be learned, it has to be understood first. Thus, under this characteristic we have grouped those attributes that facilitate the understandability of a component, and that therefore influence its learnability.

- **User Documentation**

This attribute measures the quality of the user documentation, in terms of its completeness, clarity, and usefulness. A *Level* metric will measure this subjective attribute.

- **Help System**

This attribute measures the quality of the Help System provided with the component for discovering and understanding its services, in terms of

its completeness, clarity, and usefulness. A *Level* metric will measure this attribute.

- **Computer Documentation**

This attribute indicates whether the component provides any kind of mechanism or documentation that can be used by component tools or platforms for discovering and understanding its services, and for dynamically invoking them. Examples include reflective mechanisms [14], or UML or MOF (Meta-Object Facilities) descriptions of the component services and context. A *Presence* metric will measure this attribute.

- **Training**

This attribute is measured by a *Presence* metric that indicates whether training courses are available for the component, and information about them if this is the case.

- **Demonstration Coverage**

This attributes tries to measure the percentage of the component services that are shown in a demo, compared to the total number of provided services (interfaces). A *Ratio* metric is used to measure this attribute, according to the formula:

$$100 * \frac{\text{Interfaces shown in a demo}}{\text{Provided Interfaces}}$$

#### 4.2.8 Attributes associated to “Operability”

- **Effort for operating**

This attribute indicates, using a *Level* metric, the level of effort needed to properly operate the component.

- **Tailorability**

This attribute indicates, using a *Level* metric, the level of effort needed to properly customize the component by configuring its parameters.

In addition to the number of the component configuration parameters (which is measured by the Customizability attribute) other issues need to be considered when measuring the effort required for configuring the component correctly. Examples are the number of possible values these parameters allow, their possible meanings, or their implications on the component’s behavior.

- **Administrability**

This attribute indicates, using a *Level* metric, the level of effort needed to properly administer the component.

#### 4.2.9 Attributes associated to “Complexity”

This characteristic aims at measuring the complexity of using and integrating the component into the final system. For that we will measure the number of provided and required interfaces, and the average number of operations per interface.

##### – Provided Interfaces

This attribute counts the number of provided interfaces by the component as an indirect measure of its complexity. The greater the number, the greater the complexity to use and, probably, its functional complexity. This attribute is measured by an *Integer* variable.

##### – Required Interfaces

This attribute counts the number of interfaces that the components requires from other components to operate. It provides an indicator of the complexity of the component for integration in a system, as well as the level of dependency with other external components. An *Integer* variable measures this attribute.

##### – Complexity Ratio

This attribute shows the average number of operations per provided interface. It is measured by a *Index* derived metric calculated according to the following formula:

$$\frac{\textit{Operations in all Provided Interfaces}}{\textit{Provided Interfaces}}$$

#### 4.2.10 Attributes associated to “Changeability”

##### – Customizability

This attribute measures the number of customizable parameters that the component offers.

##### – Customizability Ratio

This attribute compares the number of parameters offered by the component with the number of its provided interfaces. This measure gives us an indication of its ability to be customized. Thus, a component with very few interfaces and lots of parameters will probably be very customizable, although difficult to handle. On the contrary, a component with lots of interfaces but very few parameters does not offer a high degree of customizability. An *Index* variable measures this attribute according to the following formula:

$$\frac{\textit{Number of Parameters}}{\textit{Number of Interfaces}}$$

##### – Change Control Capability

This attribute tries to capture the user ability to easily identify the current version of a component. It is measured by a *Level* variable.

#### 4.2.11 Attributes associated to “Testability”

These attributes indicate whether the component provide some sort of tests or test suites that can be performed to the component to check its functionality inside (or in isolation of) the final system in which the component will be integrated.

##### – Start-up self-test

This attribute describes the component ability to test itself and the environment for successful operation. It is measured by a *Presence* metric which indicates the provision of start-up self-tests and, if so, their description.

##### – Tests suite provided

This attribute indicates whether some test suites are provided for checking the functionality of the component and/or for measuring some of its properties (eg. performance). A *Presence* variable describes whether such tests are available, as well as information about them if applicable.

## 5 Attribute documentation

Once we count with a set of quality attributes for describing some of the extra-functional properties of components, this section discusses how to document them.

Among the possible options, probably the ODP “properties” approach is the most convenient [9]. In this approach, each attribute is described by a pair (name,value). Each name has also a type, which determines the possible values it may hold.

One of the benefits of this way of describing properties, apart from being compliant with an international standard, is that it is easy to document with XML templates. The following example shows the description of a couple of attributes, using the W3C properties schemas and our quality attributes:

```
<property name="Computational Accuracy">
  <type>xsd:ratio</type>
  <value>100</value>
</property>
<property name="Help Documentation">
  <type>xsd:level</type>
  <value>3</value>
</property>
```

This notation is also consistent with other proposals for documenting COTS components, and with some of the existing tools for searching and trading

Sub-characteristic	Attribute	Type
Suitability	1. Coverage	Ratio
	2. Excess	Ratio
	3. Service Implementation Coverage	Ratio
Interoperability	4. Data Compatibility	Presence
Compliance	5. Standardization	Presence
	6. Certification	Presence
Compatibility	7. Backwards Compatibility	Presence
Maturity	8. Volatility	Time
	9. Evolvability	Integer
	10. Failure removal	Integer
Learnability	11. time to use	Time
	12. Time to configure	Time
	13. Time to admin	Time
	14. Time to expertise	Time
Understandability	15. User Documentation	Level
	16. Help System	Level
	17. Computer Documentation	Presence
	18. Training	Presence
	19. Demonstration Coverage	Ratio
Operability	20. Effort for operating	Level
	21. Tailorability	Level
	22. Administrability	Level
Complexity	23. Provided Interfaces	Integer
	24. Required Interfaces	Integer
	25. Complexity Ratio	Index
Changeability	26. Customizability	Integer
	27. Customizability Ratio	Index
	28. Change Control Capability	Level
testability	29. Start-up self-test	Presence
	30. Tests suite provided	Presence

Table 4: Quality Attributes for components COTS measurable during life cycle

for COTS components in software repositories or in the Web [7, 8].

## 6 Related work

There are two kinds of proposals that can be directly related to ours. In the first place we find those works that provide definitions and classifications of the quality characteristics of software products. The international standards ISO 9126 [11] and IEEE/ANSI 830-1993 [6] are among those works, together with some other academic proposals [3, 4, 12, 13]. The main difference between these works and ours is the size of the domain for which the quality attributes are defined. Basically, their generality hinders its practical applicability (specially in industrial environments). In this sense, our proposal is much more focused on a particular domain, concentrating on the specific characteristics of the COTS components.

The second line of works that can be related to ours is concerned with the evaluation process required

for selecting COTS components [1, 5, 10]. Although these proposals cover a wider scope than we do in this work, they sit at a very high level of abstraction, failing to specify any detailed set of attributes or metrics. In this sense, our work complements these proposals, allowing them to “touch the ground” and provide some implementable processes.

Part of our future work is the closer integration of both the process and product aspects of quality evaluation, using the quality model proposed here as a common base.

## 7 Discussion

In this paper we have presented a particularization of the ISO quality model, adapted to deal with the specific characteristics of COTS components. A set of quality attributes for this kind of components has been identified, together with a set of metrics for measuring them. It is important to notice the relationship between the quality model we propose and the iden-

tified attributes, something that most of the current proposals and standards do not cover.

Our long term objective is the definition and provision of better component selection and evaluation processes and tools. Although it is a very interesting goal to pursue, we are also aware of many of the difficulties involved. First, we do not think that component vendors will ever reach a complete agreement on the quality attributes that need to be included in a quality model for COTS components. Usually, vendors will try to include those attributes for which their products are competitive, while not giving any importance to those not implemented or for which no good figures are obtained. And even if a compromise is reached, most of the vendors will not be happy to provide all the requested information. Marketing, image, and commercial reasons will hinder vendors to disclose negative information about their products, such as the number of bugs reported for a given version, or the lack of an important property. We personally think that these are part of the difficulties that ISO standards are finding for getting support on quality matters—apart from being too general, of course.

We think however that a real Software Engineering cannot be achieved without metrics for effectively evaluating software products. Two main factors can help solve this conflict. First, we need agreements on specific and detailed quality models in order for these models to be applied. Our work provides a modest proposal in this respect. We do not try to impose our model, but hope it can serve as a starting point for further discussions. And second, we think that the evaluation of the quality of COTS components (ie. the assessment of their quality attributes) needs to be done by independent parties, at least until software vendors acquire the level of maturity that hardware vendors currently have. We are still far from counting with the hardware *data sheets* and catalogues available for hardware components, that describe all their characteristics. However, we need to have them for software components too if we want to talk about a “real” Component-based Software Engineering.

## References

- [1] Chris Abts, Barry W. Boehm, and Elizabeth Bailey Clark. Cocots: A cots software integration lifecycle cost model - model overview and preliminary data collection findings. Available at [sunset.usc.edu/publications/TECHRPTS/2000/usccse2000-501/usccse2000-501%.pdf](http://sunset.usc.edu/publications/TECHRPTS/2000/usccse2000-501/usccse2000-501%.pdf), 2000.
- [2] Motoei Azuma. Square the next generation of the ISO/IEC 9126 and 14598 international standards series on software product quality. *ESCOM (European Software Control and Metrics conference)*, April 2001. Available at <http://www.escom.co.uk/conference2001/papers/azuma.pdf>.
- [3] B.W. Boehm and al. Qualitative evaluation of software quality. In *Proc. 2nd ICSE*, pages 592–605, 1976.
- [4] Jan Bosch. *Design & Use of Software Architectures*. Addison Wesley, 2000.
- [5] Wilfred J. Hansen. A generic process and terminology for evaluating cots software. Available at <http://www.sei.cmu.edu/staff/wjh/Qesta.html>, August 2001.
- [6] IEEE/ANSI. Recommended practice for software requirements specifications. International Standard 830-1993, IEEE, 1993.
- [7] Luis Iribarne, Carina Alves, Jaelson Castro, and Antonio Vallecillo. A non-functional approach for cots-components trading. In *Proc. of WER 2001*, Buenos Aires, Argentina, 2001.
- [8] Luis Iribarne, José M. Troya, and Antonio Vallecillo. Trading for COTS components in open environments. In *Proc. of the 27th Euromicro Conference*, pages 30–37, Warsaw, Poland, September 2001. IEEE CS Press.
- [9] ISO/IEC. RM-ODP. Reference Model for Open Distributed Processing. Rec. ISO/IEC 10746-1 to 10746-4, ITU-T X.901 to X.904, ISO/ITU-T, 1997.
- [10] ISO/IEC-14598. Software Engineering - Product evaluation. International Standard ISO/IEC 14598, ISO.
- [11] 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.
- [12] James A. McCall, Paul K. Richards, and Gene F. Walters. Factors in software quality, volume III: Preliminary handbook on software quality for an acquisition manager. Technical Report RADC-TR-77-369, vol. III, Hanscom AFB, MA 01731, 1977.
- [13] Otto Preiss, Alain Wegmann, and Jason Wong. On quality attribute based software engineering. In *Proc. of the 27th Euromicro Conference*, Warsaw, Poland, September 2001. IEEE CS Press.
- [14] Clemens Szyperski. *Component Software Beyond Object-Oriented Programming*. Addison-Wesley and ACM Press, Boston, Ma, 1998.