

A Quality-Model-Based Approach for Describing and Evaluating Software Packages[†]

Xavier Franch, Juan P. Carvallo[‡]

Universitat Politècnica de Catalunya (UPC)

c/ Jordi Girona 1-3 (Campus Nord, C6) E-08034 Barcelona (Catalunya, Spain)

{franch, carvallo}@lsi.upc.es

Abstract

Selection of software packages from user requirements is currently a central task in software engineering. Selection of inappropriate packages may compromise some business processes and may interfere negatively in the functioning of the involved organization. Success of package selection is currently endangered because of many factors, being one of the most important the absence of structured descriptions of both the package features and the user quality requirements. In this paper, we propose a methodology for describing the quality factors of software packages using the ISO/IEC quality standard as framework. Following this standard, relevant attributes for a specific software domain are identified and structured as a hierarchy, and metrics for them are chosen. Software packages in this domain can be then described in a uniform and comprehensive way. Therefore, selection of packages can be ameliorated by transforming user quality requirements into requirements expressed in terms of the quality model attributes. We illustrate the approach by presenting in some depth a quality model for the mail servers domain.

1. Introduction

The growing importance of commercial *software packages* (also known as *COTS components* or *COTS products* [COTS02]) in software development requires adapting some software engineering practices to this framework. This includes traditional activities such as requirements elicitation, architectural design and testing, but also some specific of the field, among which selection of software packages (also known as *software package procurement* [FSR96, NM97]) plays a prominent role.

In the last years, some methodologies have been proposed for dealing with software package selection [Kon96, MN98, BEF+02]. In all of them, one key point is the comparison of user requirements with the capabilities of the evaluated packages. Requirements have to be with different kind of factors, such as managerial, political and of course quality characteristics, i.e. *quality requirements*.

Quality requirements are often difficult to check. This is partly due to their very nature, but we argue in this paper that there is another reason that can be mitigated, namely the lack of structured and widespread descriptions of *software domains* (i.e., categories of software packages, such as ERP systems, graphical or data structure libraries, etc.). This absence hampers the accurate description of software packages and the precise statement of quality requirements. As a consequence, the whole package selection activity is damaged, and confidence on the result of the process diminishes. In this paper, we propose the adoption of a structured *quality model* as an essential aid for solving this drawback. A structured quality model for a given software domain provides a taxonomy of software *quality features* and also *metrics* for computing their value. For defining the taxonomy, we may use any feasible existing quality standard; among them, we have selected the ISO/IEC one¹ [ISO91] for the following reasons:

- It just fixes some general characteristics, and so the quality model may be tailored to any specific software domain. This is a crucial point, because quality models may dramatically differ from one domain to another.
- The standard explicitly recognizes the convenience of creating hierarchies of quality features, which is essential in order to build structured quality models.
- It is widespread.

The main goal of this paper is to define a methodology for building ISO/IEC-based quality models for software domains. A skilled *quality team* including experts in the domain is supposed to be in charge of this construction. Once the quality model is available, both software package descriptions and user quality requirements may be translated into the quality concepts defined therein, favouring then the whole selection process and also increasing the confidence in its result.

The paper is structured as follows. In section 2 we give an outline of the ISO/IEC quality standard. Section 3 provides a general methodology for building ISO/IEC-based quality models, which is illustrated with an example in section 4. Section 5 outlines how the quality model may

[†] This work is partially supported by the Spanish research programme CICYT under contract TIC2001-2165.

[‡] Juan P. Carvallo's work has been supported by an AEI grant.

¹ Nevertheless, the concrete standard adopted is not really a crucial point of the methodology.

help in package description and requirement statement. Last, section 6 presents the conclusions.

2. The ISO/IEC Software Quality Standard

A set of ISO/IEC standards are related to software quality, being standard number 9126 (which is in process of substitution by 9126-1, 9126-2, 9126-3 and 9126-4), the most relevant one with respect to our work [ISO91].

The main idea behind this standard is the definition of a *quality model* and its use as a framework for software evaluation. A quality model is defined by means of general *characteristics* of software, which are further refined into *subcharacteristics*, which in turn are decomposed into *attributes*², yielding to a multilevel hierarchy; in fact, as mentioned by the standard, intermediate hierarchies of subcharacteristics and attributes may appear. At the bottom of the hierarchy there are the measurable software attributes, which values are computed by using some *metric*. Throughout the paper, we refer to characteristics, subcharacteristics and attributes as *quality entities*. *Quality requirements* may be defined as restrictions over the quality model.

The ISO/IEC 9126 standard fixes six characteristics: functionality, reliability, usability, efficiency, maintainability and portability. Furthermore, an informative annex of this standard provides an illustrative quality model that refines the characteristics.

Figure 1 presents an abridged UML conceptual model that summarizes the concepts outlined in this section. Some OCL-constraints are not included. The “measured by” association states that the metric for a measurable attribute may be different depending on the subcharacteristic or attribute where it appears.

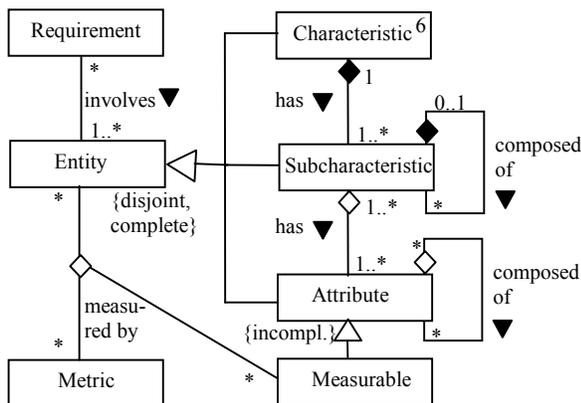


Figure 1 UML conceptual model for the ISO/IEC standard

² The concept of attribute does not appear in the standard presentation, but in the description of subcharacteristics.

3. Applying the ISO/IEC Standard

Using the ISO/IEC quality standard as framework, we propose in this section a methodology aimed at defining a quality model for a given software domain. This implies identifying the appropriated subcharacteristics and their attributes, and also the metrics for these attributes; attributes and even subcharacteristics will usually be organized as a hierarchy. Once this process is completed, requirements over the domain, as well as package features, may be stated with respect to the resulting quality model. The framework can therefore be used to support the classical characteristics–requirement negotiation process during software package selection (see figure 2).

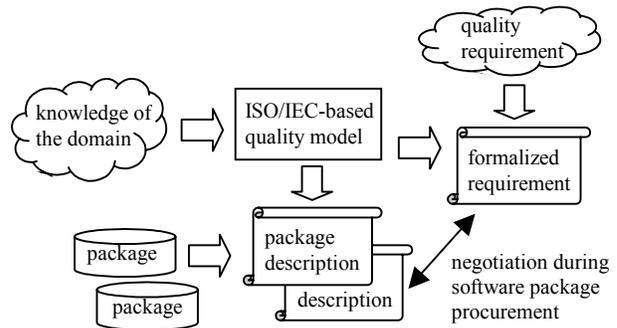


Figure 2 Using a quality model in software procurement

The methodology consists of six steps with a preliminary one. Although they are presented as they were sequential, they may be intertwined or iterated at any acceptable extend. We illustrate it with short examples, and we provide a more complete case study in section 4.

Step 0. Defining the domain

First of all, the domain of interest has to be carefully examined and described. With respect to the first point, experts of the field must participate in the quality team. Concerning the second point, a conceptual model can be built to keep track of all relevant concepts.

When performing this step, we have discovered that one of the most endangering points is the lack of standard terminology in packages of the domain. The same concepts are named different by different vendors or even worse, the same name may denote different concepts in different packages. It is utterly important to discover all these conflicts during this preliminary step in order to avoid semantic problems when identifying quality attributes.

Step 1. Determining quality subcharacteristics

The ISO/IEC standard fixes six quality characteristics but not their further refinement into subcharacteristics; the proposal of the annex is labelled as “informative”.

The first step in building a quality model for a software domain is therefore determining the division of characteristics into subcharacteristics.

As a general rule of thumb, we propose the annex as starting point: it is quite reasonable, and although not mandatory, it does appear in the standard itself. The quality team may then add new subcharacteristics specific to the domain, refine the definition of some existing ones, or even eliminate some (although this last situation will seldom take place). Some examples follow:

- In the domain of ERP systems, a subcharacteristic for keeping track of the areas covered (finance, staff, etc.) may be added to functionality.
- In the domain of data structures libraries, the time behaviour subcharacteristic may be refined as "execution time of the methods provided by the classes inside the library".

Step 2. Defining a hierarchy of subcharacteristics

In the general case, subcharacteristics may be further decomposed with respect to some factors, yielding thus to a hierarchy of them. It is important to remark that new subcharacteristics must be at the same abstraction level; otherwise they would be quality attributes.

A typical example appears in the suitability subcharacteristic of functionality. Successful software packages tend to bind applications that were not originally related to them. This is particularly true if one considers that product suppliers try to include some features to make their products different from the others. These added applications are not usually shipped within the original packages; they are offered separately, as extensions of the original one. But in many cases, they are referenced as a constitutive part of the functions provided by the package. As a result, it seems quite natural to split the suitability characteristic into two, basic suitability and added suitability, keeping track of both of them inside the model but in a clearly separated way.

Step 3. Decomposing subcharacteristics into attributes

Quality subcharacteristics provide a comprehensible abstract view of the quality model. But next it is necessary to go into the details, by decomposing these abstract concepts into more concrete ones, the quality attributes. An attribute keeps track of a particular observable feature of the packages in the domain. For example, attributes in the learnability subcharacteristic may include quality of the graphical interface of the product, number of languages supported (english, ...) and quality of the available documentation.

Sometimes it may not be possible to list all the quality attributes related to a particular kind of software, but it is certainly feasible to create a very complete list of the most relevant ones. Success on the identification of the right attributes in a particular domain requires not only inspecting documentation, talking to suppliers and experimenting with some representative packages, but also

including in the quality team members with a high level of conceptual knowledge of this domain. Concepts are the key elements when selecting quality attributes; it is necessary to keep in mind that the goal is to define a general framework for many applications of the same brand, not one for a particular product. To sum up, the quality team should look for a qualitative list of attributes instead of a quantitative one.

When decomposing characteristics into attributes, it turns out that some attributes are suited for more than one subcharacteristic. For instance, an attribute for the fault tolerance subcharacteristic in the domain of data structures libraries may be the type of error recovery mechanism. But in fact, this attribute may be seen also as a constituent of the testability subcharacteristic (a powerful error recovery mechanism makes testability of the library easier). As this is a natural situation, we do not force attributes to appear in a single subcharacteristic; they are allowed to be used in many of them (see the corresponding multiplicity in the UML model of fig. 1).

As stated in the conceptual model, attributes categorized under multiple subcharacteristics may use different metrics (see step 5) for each case depending of the concept they represent under each particular subcharacteristic.

Step 4. Decomposing derived attributes into basic ones

Some of the attributes emerging in step 3 may be directly measurable given a particular product (e.g., number of languages supported) but others may be still abstract enough to require further decomposition. This is the case of the quality of interface attribute mentioned above; quality may depend in various factors, as user-friendliness, depth of the longest path in a browsing process, types of interface supported (web interface, ...) and so on. Therefore, we distinguish between *derived attributes* and *basic ones* (which we have called measurable in the conceptual model). Derived attributes should be decomposed over and over until they are completely expressed in terms of basic ones.

A particular case of derived attributes appears when taking their scope into account. Let's consider again the domain of data structures libraries and error recovery. There are many mechanisms to deal with error recovery, for instance: a naive notify-and-abort action, an error notification via parameters, or an exception mechanism; even there could be no error recovery at all. But in fact, the error recovery attribute of the library should be defined in terms of the error recovery mechanism of the classes therein, represented with a new attribute bound to classes. The definition is made with a kind of AND-rule (i.e., if all the classes have the same error recovery mechanism, this is the error recovery mechanism of the library) slightly modified for the case of heterogeneous error recovery mechanisms.

In this example, the derived attribute has been completely defined in terms of their components, but it could not be the case. Giving a concrete definition of the quality interface attribute could be considered harmful, because it would force to use always the same definition without considering the requirements of a particular context. Sometimes requirements may give more importance to the user-friendness factor (e.g., for non-skilled users), sometimes to its type (for interoperability purposes) and so on. In this case, the definition of the derived attribute is postponed. We call the first case of derived attributes *context-free*, while the second ones are *context-dependent*.

Step 5. Determining metrics for basic attributes

Not only the attributes must be identified, but metrics for all the basic attributes must be selected, as well as metrics for those derived context-free attributes. We strongly recommend the use of mathematical concepts for describing precisely the meaning of the metric, and of course the general theory of metrics should be followed [Zus98]. The simplest kinds of metrics are:

- *Boolean*. To state presence or absence of a product feature. For instance, whether the data under the package control is encrypted or not.
- *Numerical*. The attribute states some kind of measure, either integer (e.g., the depth of the longest path attribute, see above) or real (e.g., execution time of a particular function). Upper and lower bounds should be declared, if they exist.
- *Label*. The attribute records a name. An example is the kind of protocol used by a mail server. The label domain may be *closed* (i.e., a domain by enumeration of its values) or *open* (i.e., a string).

In the case of basic attributes, metrics must be quantitative; in derived attributes, they could be either quantitative or qualitative, with explicit formula computing their value from their component attributes. It is worth remarking the qualitative label metrics, with values such as "good", "fair", "poor" and so on, which are referred to as *rating levels* in the ISO/IEC standard.

Some attributes require a more complex representation, yielding to *structured* metrics. More precisely:

- *Sets*. The attribute records a collection of values. It is the case of the number of languages supported by the interface (set of labels).
- *Functions*. The value of the attribute is not absolute, but depends on some other value. A typical case are the attributes that depend on the underlying platform. For instance, many attributes related to the time behaviour subcharacteristic may fall into this category. Any restriction in the function domain or range should be stated.

Metrics for some quality attributes may be difficult to define. However, it is our believe that this is the only way to have an exhaustive and fully-useful quality model. Also,

the structured description style adopted here, identifying basic attribute domains in terms of mathematical entities, and also the definition of the derived context-free ones, can be seen as the starting point for a formalization in some structured notation [CNYM00] or an interface description language [Fra98].

Step 6. Stating relationships between quality entities

To end up with a real complete quality model, it is not enough with identifying quality subcharacteristics and attributes; relationships between them must also be explicitly stated. The model becomes more exhaustive and as an additional benefit, quality user requirements may get implicitly extended once they have been expressed in terms of quality attributes.

Given two quality entities A and B we may identify various types of relationships:

- *Collaboration*. Growing of A implies growing of B . For instance, the security subcharacteristic collaborates with the maturity one. Sometimes the relationship is symmetric.
- *Damage*. Growing of A implies decrease of B . For instance, the error recovery mechanism attribute collides with the time behaviour subcharacteristic: the more powerful the mechanism is, the less fast the program runs. Sometimes the relationship is symmetric. Damages are more frequent than collaborations.
- *Dependency*. Some values of A require B fulfilling some conditions. For instance, having an exception-based error recovery mechanism requires the programming language being one with exception constructs.

More elaborate types and also intensities of these relationships may be built, as done in [CNYM00]. As done there, the relationships found may be depicted by means of a tabular representation. Attributes in rows contribute to attributes in columns, with either a positive influence (+, collaboration), a negative influence (-, damage) or a dependency (\leftrightarrow).

4. A Case Study: Mail Servers

As electronic mail services have grown in importance, companies have increased their use to improve inside and outside communication and coordination. An overwhelming number of mail-related products are currently available and organizations face the problem of choosing among them the ones that best fit their needs. For some companies, an inappropriate selection would compromise their success. For all these reasons, having a good quality model for this domain can be considered especially useful. For this and also for having some experience in the field, we have selected this domain for illustrating the general methodology seen in section 3.

Core components of mail systems are mail servers; a successful mail service deployment depends of their correct selection and configuration. In the remaining of this section we will use the methodology stated in section 3 to define a quality model for mail server packages.

Step 0: Defining the domain

The Internet Mail Consortium (IMC) [IMC02] describes the basic client-server mailing architecture (see fig. 3) as the process of relaying mail from an originator mail user agent (MUA), to a recipient one through one (or various) mail transfer agents (MTA). When mail arrives to destination, the final MTA delivers the message to the appropriated message store (MS); from this MS, mail can be accessed by the recipient.

In practice, MTA are software packages installed and running over a single mail server computer or groups of them (mail server cluster). Similarly, MUA are software packages known as mail clients running over the user local machine.

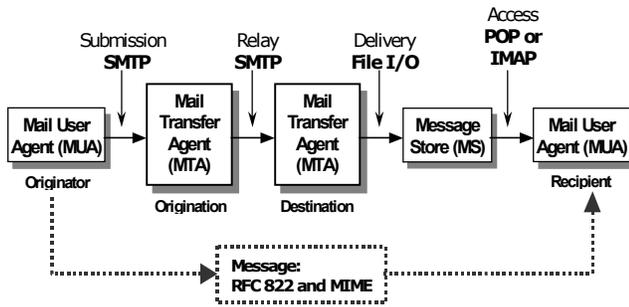


Figure 3. IMC basic mail architecture.

Important topics to be addressed in the mailing domain are protocols for exchanging messages, directories for locating information, types of mail client applications to access the messages, message security issues and other specialised ones, as clustering of server computers. In figure 4 some possible mail architectures are described in a graphical way³.

The accurate description of the domain has shown different semantic ambiguities that have been corrected. For example "junk mail filtering", "bulk mail handling" and "spammers thwarting" were mentioned in different sources. Although presented as different things, after carefully analysing them we found that all were making reference to the possibility of applying filters to incoming messages and of denying their reception based on the originator mail address. This reduces the amount of bulk mail, and helps thwarting spammers.

³ We do not include a conceptual model for lack of space.

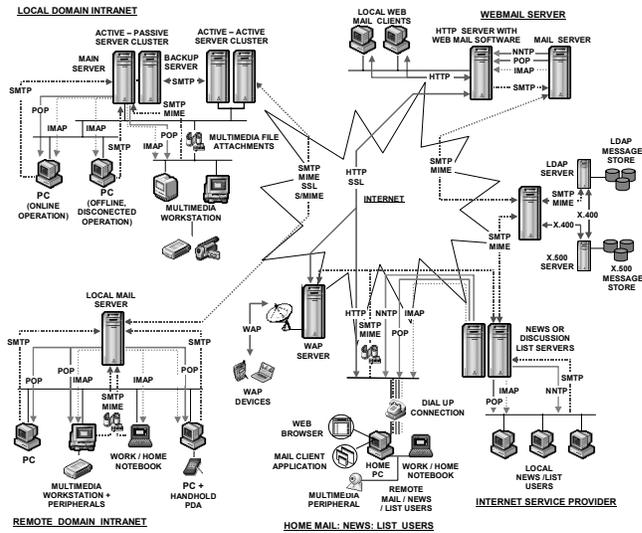


Figure 4. Mailing architectures.

Step 1: Determining quality subcharacteristics

We have decided that subcharacteristics suggested in the ISO/IEC standard are complete enough to be used as starting point and they have been adopted with just some minor modifications in their definition.

Step 2: Defining a hierarchy of subcharacteristics

According to the criteria mentioned in section 3, we decided to split the suitability subcharacteristic into mail server suitability and additional suitability subcharacteristics. This decision was taken because we found that some additional communication applications such as chat, instant messaging, whiteboarding or video conference, or even some collaborative ones such as workflow project management tools, were increasingly promoted for some vendors as if they were a constitutive part of their mail server products (the truth is that additional software is required). Many companies may be interested in using them and so we decided that it was important to create a functionality subcharacteristic to include them.

One could be tempted to apply this principle in other situations, but it must be done carefully. For instance, in some contexts the attributes categorized under the operability subcharacteristic of usability may be seen from two different points of view: the general user and the administrator. This is the case for mail servers products and for this reason at the beginning we considered to divide this subcharacteristic into two. At the end, we decided that general user operability on mail servers depends on the mail client and the privileges given by the administrator. We were not able to clearly see attributes related to clients that were independent of those related to administrators, and so we decided to keep only one subcharacteristic.

Step 3: Decomposing subcharacteristics into attributes

In order to identify the attributes, extensive research of the domain has to be done. Different, classical sources are used for this purpose: general articles about technologies involved in the domain; articles comparing products as well as related technologies; advertising information of products; technical documentation of packages; product demos as well as public or private presentations carried out by the vendor; documented selection cases of products of the domain; and hands-on experimentation.

Once identified, attributes have to be assigned to subcharacteristics. Contrary to what may be expected, this process is neither simple nor mechanical. Just to mention some of the problems that we may found:

- The number of elements may get to be very high, making it difficult to handle them. In our case we started with over 200 attributes, and after a detailed analysis this number was reduced to 160.
- In some cases, values of attributes may be confused with attributes themselves. For example, at the beginning we create one attribute instance to represent each of the POP3, IMAP4, SMTP and X.400 protocols (which were categorized under the compliance subcharacteristic). Later we realized that in practice there are only two attributes *mail transfer protocols* and *mail access protocols*.
- Some attributes may be identified as a single one but after some analysis may result better to break them into several ones. For example, average response time was listed as one of the attributes of the time behaviour subcharacteristic of efficiency. Later we split this attribute into two, the average response time and message throughput, one to represent the amount of time required for the server to identify new mail to be send and the other to represent the time per size unit required to actually send a message.
- As mentioned in section 3 some attributes are suited for more than one characteristic. For instance, message tracking and monitoring may be seen as a functional attribute that grants accurateness, or else as a analysability attribute of the maintenance characteristic.

It is important to notice that, due to their hierarchical nature, these kind of models help to discover these problems and facilitates their faster resolution.

Hands-on experimentation is necessary to obtain really independent information. For instance, attributes such as administrative or expert analysis tools were mentioned in the documentation of almost every product, but very vague description of them was given. We installed some products and had some hands-on experience to better understand these concepts. This experiences turned out to be very valuable as closing point for this step.

Step 4. Decomposing derived attributes into basic

As mentioned in step 4 of section 3 some attributes require to be decomposed because they are not directly measurable. We have identified several of them. For instance, the attribute for resources administration (characteristic usability, subcharacteristic operability) has been decomposed into the following basic attributes: maximum storage time of mail messages; maximum time of life for inactive accounts; mailbox quotes; mail file sizes; management of groups of servers as a single entity.

Step 5. Determining metrics for basic attributes

We also have determined metrics for attributes in model; some of them where hard to define, especially the qualitative ones. The concepts shown in step 5 of section 3 have been used in this section. Table 1 shows some attributes with different kind of metrics.

Attribute	Metric
Permanent message redirection?	Boolean
Maximum account size	Integer (megabytes)
Default folders provided	Set of labels
Average response time	Function from platform to real numbers

Table 1. Some attributes and their metrics.

Step 6. Stating relationships between quality entities

Having about 160 attributes, it is quite natural that a lot of relationships between them appear. We present some collaborations and damages in table 2.

CHARACTERISTICS		Efficiency	
SUBCHARACTERISTICS		Time behaviour	
ATTRIBUTES		Average response time	
Reliability	Recoverability	Full or selective replication and synchronization	-
		Single mailbox backup and recovery	-
		Online incremental backup	-
		Online restore	-
		Dynamic Log rotation	-
		Event Logging	-
		Transaction Logging	-
Efficiency	Resource behaviour	Number of concurrent mail users per server	-
		Number of active webmail clients	-
		Management of quotas on message and mail file size	+
		Message volume of their target customer	-
		Single copy store	+

Table 2. Related Efficiency – Reliability attributes.

Examples of dependencies are:

- If some certification system is selected, some encryption algorithm must also be used, because it is needed to grant confidentiality.
- If mail server supports news groups, it must support the NNTP protocol which is the TCP/IP standard application for this kind of message exchange.

5. Package and Requirement Descriptions

Once the quality model for a software domain is built, it becomes possible to describe packages in this domain and to express quality requirements to model the needs of a company in the process of selecting the package that best fits its needs.

When describing package quality characteristics, it turns out to be very difficult to find complete and reliable information of them. Let's consider again the mail server domain. Manufacturers tend to give just a partial view of their products. Either they put so much emphasis on their product benefits, without mentioning the weakness, or they give a partial look of the truth, making them seem capable of more features of which they really cover. Some third-party reports look very independent, but they have been strongly refuted for technical departments of parties involved, making them difficult to rely on. Other non-commercial articles compare features, but they base their reports on evaluators knowledge of the tools, and their particular taste, more than in serious technical tests.

This situation points again to the convenience of having expert users of the tools as members of the evaluation team; their presence reduces the time required for the description of product characteristics, improves their matching with quality attributes and also grants the accurateness of this information. Also, some degree of hands-on experimentation is required.

One of the most time-consuming and error-prone tasks are the computation of attributes whose values depend on the value of other attributes, which are modelled as function attributes, especially in the case of platform-dependencies. For example, average response time depends of the hardware been used and even of the operating system in which product is installed (see table 1). This requires multiple installations of the package and repetition of its evaluation.

Concerning quality requirements modelling, we have introduced complete sets of quality requirements that appeared in real mail server selection processes with very different characteristics (from a public institution giving service to 50.000 people to a small software consultant and ISP provider company). Some requirements were already presented in a structured way (for example as lists of interconnection-, functionality- and utilisation-related requirements) but others not, leading to some extra effort to arrange them.

Because of the extend of the paper we cannot present the complete list of requirements of any of these real cases. Nevertheless, we show in table 3 some requirements that illustrate some problems we found when expressing them in terms of the quality model.

Req.	Requirement Description
1	Spanish language support
2	Support for the most commonly used certification standard
3	Support for accessing the server from other applications
4	Protection against viruses and any other risks
5	Mail delivery notifications, possibility of configuring parameters such as maximum number of delivery retries, and time between them
6	Message throughput time must be inferior to 1 minute for messages with no attachments. For messages with attachments must be inferior to 5 minutes per megabyte

Table 3. Some example requirements.

Requirements such as 1 or 2 can be directly mapped into one single attribute of the model. The only difference is that requirement 2 demands the expert team for mapping the expression "most commonly used certification standard" to a concrete value of the corresponding attribute, that is the value "X.509".

Requirements 3 and 4 are example of too general requirements (what does it mean "other applications" and "other risks"?). Further interaction to get a more detailed specification must be provided to better classify them.

Requirements 5 is an example of requirement that either require or imply a mixture of functionalities, which may be supported by selecting several attributes. Although further feedback may be required in order to better classify this kind of requirements, we succeed in doing so for this particular requirement.

There are also some requirements that are originally expressed in a incorrect way but are somehow understandable. This is the case of requirement 6 which was originally expressed in terms of average response time. We reformulated it in terms that made its representation into the model feasible.

Once all the requirements for a particular company are incorporated into the model (after completing, discarding and reformulating them), they can be extensively compared with respect to available package descriptions. This allows to detect differences between products as well as determining to what extend they cover the expressed needs. Once we arrived to this point in our experience in the mail server domain, we had no doubts about utility of model as a valuable tool to help in selection of products (although this is not the final goal of our methodology and multi-criteria decision making techniques such as [Saa90] can be complementary used for this purpose). A feedback process is recommended once requirements are mounted in the model, in order to refine and extend them and also to complete their understanding if required.

6. Concluding Remarks

Reliable processing of quality requirements demands a proper quality model to be used as reference, especially in the context of software package selection. In this paper we have presented a methodology aimed at building quality models based on the ISO/IEC quality standard. The methodology is composed of a preliminary step for understanding the software domain subject of the work, and six more steps to organise quality concepts in a hierarchy, to establish their metrics and also to make explicit their relationships. We have applied our proposal to a specific software domain, the one of mail server products, outlined here and presented in more depth in <http://www.lsi.upc.es/dept/techreps/html/R02-36.html>. Once the model has been built, it can be used for describing software packages and for expressing quality requirements.

When applying our methodology, we have observed that building the quality model is a complex activity, endangered by many factors: poor description of the domain, lack of ability when identifying the quality entities, inappropriate metrics, etc. However, once available, it becomes a really powerful tool which provides a general framework to get uniform descriptions of the (potentially a great deal of) software packages of the involved domain. Comparison of these packages is then favoured. Also, quality requirements can be rewritten in terms of the quality concepts appearing in the model; this reformulation process may help to discover some ambiguities and incompleteness in the requirements and, once solved, the resulting requirements can be more easily compared with the package descriptions. In fact, a quality model obtained with our methodology can be expressed in an interface description language (as we have done in [Fra98]) and automatic support for package selection becomes then feasible [FPV99].

Not only the reliability of software package procurement can be improved with our proposal; also the cost of the very procurement process can. Just consider for a moment the amount of repeated work that is done in the mail server domain used as example in this paper. We know of many organisations that have faced exactly the same problems and have repeated the same process over and over, wasting human resources and money while doing so. The existence of a quality model for this domain makes mail server procurement a simpler task, once particular quality requirements of the organisation have been expressed in terms of the model.

One of the ongoing applications of our work has to be with software certification [Voa98]. We think that quality models can be the framework for expressing the quality requirements that define a certificate for a software domain. Certificates for a domain may be different depending in the kind of organisation (e.g., mail servers for

governmental departments may require stronger quality than ones for small companies), and also this characteristic may be expressed with respect to the quality model. The existence of software certification organisms from whom manufacturers might require basic quality models, in which they might place their product characteristics to be certified, and to whom companies might submit their particular requirements to be evaluated, would help in driving a more fair and clear competition between manufacturers of products. Manufacturers can be sure that their patents and development efforts would be certified and enforced by recognized independent organizations. Companies would also benefit because they would be granted that their selection would be based on a full understanding of their requirements and a more technical and precise evaluation of products.

Part of our future work is related to the adaptation of the presented methodology for its application in COTS-based system domains. By means of quality models, user requirements in these domains could be classified and used for the identification of the necessary COTS components as well as alternative architectures.

7. References

- [BEF+02] X. Burguès, C. Estay, X. Franch, J. Pastor, C. Quer. "Combined Selection of COTS Components". In [COTS02].
- [CNYM00] L. Chung, B. Nixon, E. Yu, J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.
- [COTS02] Proceedings of the 1st International Conference on COTS-Based Software Systems (ICCBSS), LNCS 2255, 2002.
- [FPV99] X. Franch, J. Pinyol, J. Vancells. "Browsing a Component Library Using Non-Functional Information". Procs. Ada-Europe 1999.
- [Fra98] X. Franch. "Systematic Formulation of Non-Functional Characteristics of Software". Procs. 3rd ICRE, 1998.
- [FSR96] A. Finkelstein, G. Spanoudakis, M. Ryan. "Software Package Requirements and Procurement". Procs. 8th IWSSD, 1996.
- [ISO91] ISO/IEC Standards 9126 (Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their use), 1991.
- [Kon96] J. Kontyo. "A Case Study in Applying a Systematic Method for COTS Selection". Procs. 18th ICSE, 1996.
- [MN98] N. Maiden, C. Neube. "Acquiring Requirements for COTS Selection", IEEE Software 15(2), 1998.
- [NM97] C. Neube, N. Maiden. "Procuring Software Systems: Current Problems and Solutions". Procs. 3rd REFSQ, 1997.
- [Saa90] T.L. Saaty. *The Analytic Hierarchy Process*. McGraw-Hill, 1990.
- [Voa98] J.M. Voas. "Certifying Off-The-Shelf Software Components". IEEE Software, June 1998.
- [Zus98] H. Zuse. *A Framework of Software Measurement*. DeGruyter Publisher, 1998.