

Using Quality Models in Software Package Selection

Xavier Franch and Juan Pablo Carvallo, *Universitat Politècnica de Catalunya*

The growing importance of commercial off-the-shelf software packages¹ requires adapting some software engineering practices, such as requirements elicitation and testing, to this emergent framework. Also, some specific new activities arise, among which selection of software packages plays a prominent role.²

All the methodologies that have been proposed recently for choosing software packages compare user requirements with the packages' capabilities.³⁻⁵ There are different types of requirements, such as managerial, political, and, of course, quality requirements.

Quality requirements are often difficult to check. This is partly due to their nature, but there is another reason that can be mitigated, namely the lack of structured and widespread descriptions of package domains (that is, categories of software packages such as ERP systems, graphical or data structure libraries, and so on). This absence hampers the accurate description of software packages and the precise statement of quality requirements, and consequently overall package selection and confidence in the result of the process.

Our methodology for building structured quality models helps solve this drawback. (See the "Related Work" sidebar for information about other approaches.) A structured quality model for a given package domain provides a taxonomy of soft-

ware quality features, and metrics for computing their value. Our approach relies on the International Organization for Standardization and International Electrotechnical Commission 9126-1 quality standard,⁶ which we selected for the following reasons:

- Due to its generic nature, the standard fixes some high-level quality concepts, and therefore quality models can be tailored to specific package domains. This is a crucial point, because quality models can dramatically differ from one domain to another.
- The standard lets us create hierarchies of quality features, which are essential for building structured quality models.
- The standard is widespread.

After we build the quality model, we can state the domain requirements as well as package features with respect to the model. We can then use the framework to support negotiation between user requirements and product capabilities during software package selection (see Figure 1).

This methodology drives the construction of domain-specific quality models in terms of the ISO/IEC 9126-1 quality standard. These models can be used to describe the quality factors of software packages uniformly and comprehensively and to state requirements when selecting a software package in a particular context.

The ISO/IEC software quality standards

Among the ISO and ISO/IEC standards related to software quality are the families of 9126 and 14598 for software product quality and evaluation. These standards can be used in conjunction with others concerning the software life cycle (ISO/IEC 12207), process assessment (ISO/IEC 15504), and quality assurance processes (ISO 9001).

ISO/IEC 9126-1 specifically addresses quality model definition and its use as a framework for software evaluation. A 9126-1 quality model is defined by means of general software characteristics, which are further refined into subcharacteristics, which in turn are decomposed into attributes, yielding a multilevel hierarchy. At the bottom of the hierarchy are measurable software attributes, whose values are computed using some metric. Throughout this article, we refer to characteristics, subcharacteristics, and attributes as quality entities, and we define quality requirements as restrictions over the quality model.

Table 1 shows the six quality characteristics defined in the 9126-1 quality standard and their decomposition into subcharacteristics.

Building ISO/IEC 9126-1 quality models

Our methodology comprises six steps (see Figure 2). Also, we consider a preliminary activity (Step 0) for analyzing the package domain. Although we present these steps as sequential, they can be intertwined or repeated.

Step 0: Defining the domain

First, you must carefully examine and describe the domain of interest, with the help of experts. To describe the domain, we recommend using conceptual modeling to keep track of relevant concepts.

One of the biggest problems is the lack of standard terminology among a domain's software packages. Different vendors refer to the same concept by different names, or even worse, the same name might denote different concepts in different packages. Discovering all these conflicts during this preliminary step is essential to avoid semantic mismatches throughout the software selection process.

The e-learning domain provides more than one example. Consider the term *virtual classroom*. Some packages use this term and *course* as synonymous, referring to the contents of

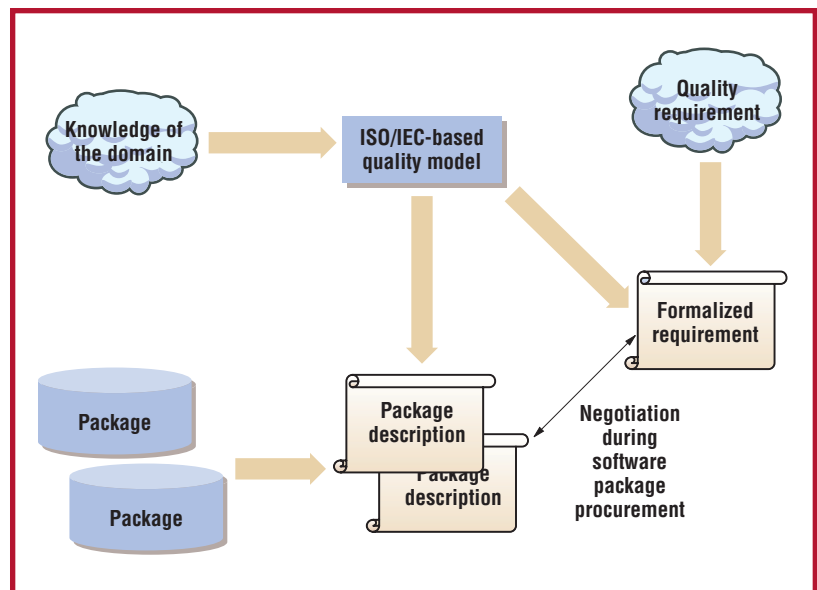


Figure 1. Using a quality model in software procurement.

Table 1

ISO/IEC 9126-1 characteristics and subcharacteristics

Characteristics	Subcharacteristics
Functionality	Suitability Accuracy Interoperability Security Functionality compliance
Reliability	Maturity Fault tolerance Recoverability Reliability compliance
Usability	Understandability Learnability Operability Attractiveness Usability compliance
Efficiency	Time behavior Resource utilization Efficiency compliance
Maintainability	Analyzability Changeability Stability Testability Maintainability compliance
Portability	Adaptability Installability Coexistence Replaceability Portability compliance

particular subjects. Other packages use different semantics for *virtual classroom*—meaning, for instance, the interface (such as the base

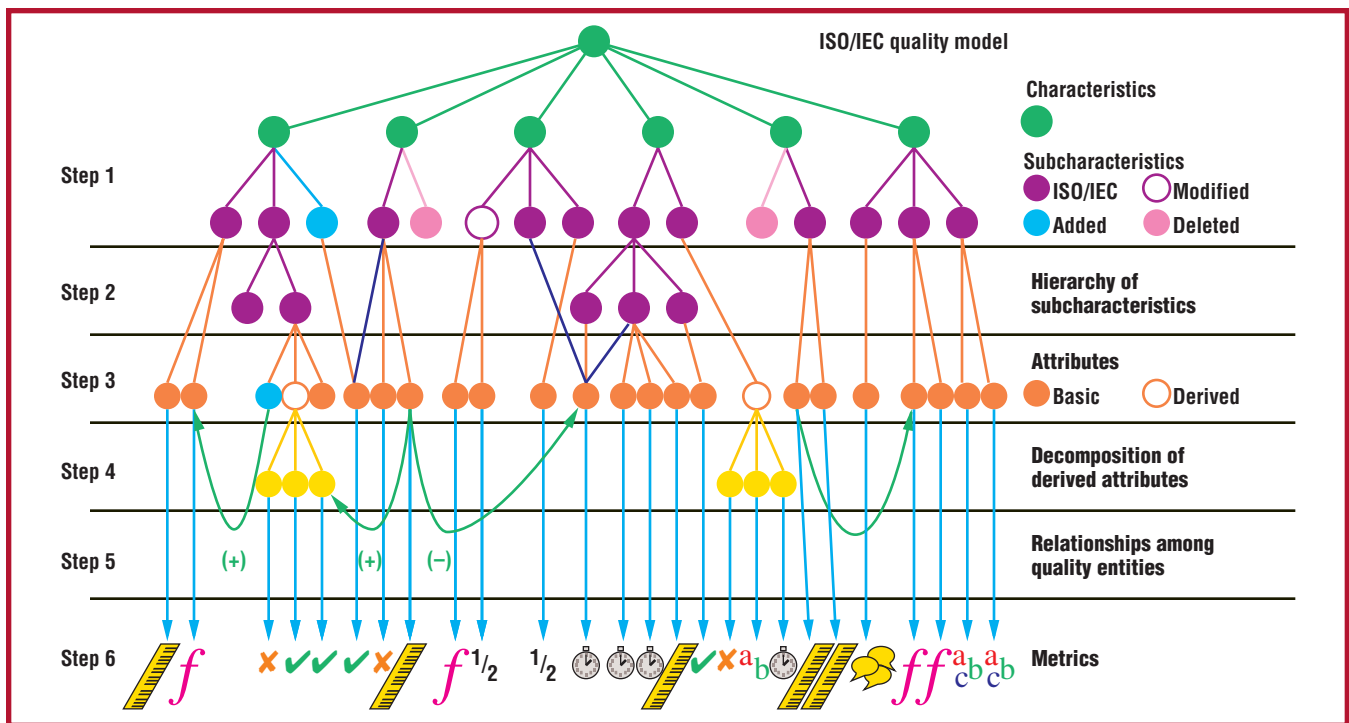


Figure 2. Our six-step methodology.

URL where the contents of a course are to be placed) or the list of tasks to be performed during the course.

Step 1: Determining quality subcharacteristics

The decomposition of characteristics into subcharacteristics that appear in the standard is quite reasonable and should be used unless good reasons for not doing so come out during domain analysis. In these cases, you might add new subcharacteristics specific to the domain, refine the definition of existing ones, or even eliminate some. For example,

- In the domain of ERP systems, you might add Functionality subcharacteristics for tracking the company areas covered (such as finance or staff).
- In the domain of data structure libraries, you might refine the Time Behavior subcharacteristic as “execution time of the methods provided by the classes inside the library.”
- In domains that are purely pieces of software to be integrated in a system, you might omit the Attractiveness subcharacteristic as defined in the standard (which keeps track of quality attributes such as color use or graphical appearance) because it does not apply.

Step 2: Defining a hierarchy of subcharacteristics

Typically, you can further decompose sub-

characteristics with respect to some factors, yielding a hierarchy.

A frequent situation appears in the Suitability subcharacteristic. Successful software packages tend to include applications that were not originally related to them. A usual reason for this is that product suppliers often try to include features to differentiate their products from their competitors’. These added applications are usually not shipped in the original packages; they are offered separately, as extensions. But, they often are referenced as essential to the functions the package provides (although additional software is often required). As a result, we can split the Suitability characteristic into two subcharacteristics, Basic Suitability and Added Suitability, keeping track of each 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, we must decompose 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 might include the Quality of the Product’s Graphical Interface, the Number of Languages Supported, and the Quality of the Available Documentation. We must define attributes precisely to

Related Work

Other authors have proposed quality models as a basis for software evaluation, but most of these proposals deal with measuring custom software instead of selecting software packages. As a result, they focus mainly on internal attributes and their implications instead of external attributes, as is our aim.

However, similarities exist. Geoff Dromey suggests the use of ISO/IEC 9126 standards, although he does not require it.¹ In fact, he uses different models for parts of his proposal, making model reusability more difficult. There are two other fundamental differences between our and Dromey's work: he does not include metrics as part of the model, and his concept of quality is context-free, whereas we introduce the idea of context-dependent quality attributes as a means to measure quality, taking into account the package's context of use.

The Squid method focuses on quality for in-house software.² The aim of Squid is wider than our approach; it covers quality planning, control, and evaluation as well as modeling. Concerning quality models, Squid is closer to our proposal than is Dromey's: it takes external attributes as a primary concern (mapping external attributes onto internal ones to be observed in the product), it considers quality models to be context-dependent, and both methods search for reusability by identifying robust attributes. On the other hand, Squid does not propose any particular procedure for building the quality model, nor

does it require the ISO/IEC standards. Perhaps the most significant difference is that Squid does not distinguish clearly between the quality model itself and its context of use (for example, the model includes upper target values for attributes), compromising model reuse in other contexts.

Our approach is also related to goal-oriented modeling.³ Typical goal-oriented analysis focuses on modeling a given, ongoing software system, making explicit the participating agents and the goals and their decomposition. In contrast, our quality model approach is not bound to any particular system. It is as context-free as possible, without any assumptions about the model's context of use. Nevertheless, we are currently aiming at using goal-oriented models for modeling the rationale behind the construction process.⁴

References

1. R.G. Dromey, "Corning the Chimera," *IEEE Software*, vol. 20, no. 1, Jan./Feb. 1996, pp. 33-43.
2. J. Bøegh et al., "A Method for Software Quality Planning, Control, and Evaluation," *IEEE Software*, vol. 23, no. 2, Mar./Apr. 1999, pp. 69-77.
3. E. Yu, "Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering," *Proc. 3rd IEEE Int'l Symp. Requirements Eng. (ISRE)*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 226-235.
4. B. Kitchenham and S.L. Pfleeger, "Software Quality: The Elusive Target," *IEEE Software*, vol. 20, no. 1, Jan./Feb. 1996, pp. 12-21.

clarify the underlying quality concepts that they represent and to link them with the appropriate subcharacteristics.

Because we focus on the framework of COTS package selection and package suppliers rarely give access to the package code, we are interested in external rather than internal attributes.

As the standard itself mentions, it is not possible from a practical point of view to measure all the subcharacteristics for all parts of a large software product, but it is certainly feasible to create a complete list with the most relevant ones. Concepts are the key elements when selecting quality attributes; the goal is to define a general framework for many applications of the same brand, not for a particular product.

When you decompose, some attributes are suited for more than one subcharacteristic. For instance, an attribute for the Fault Tolerance subcharacteristic in the domain of data structure libraries might be the Type of Error Recovery Mechanism. But in fact, this attribute might also be a constituent of the Testability subcharacteristic (a powerful error recovery mechanism makes testing the library easier). Therefore, we do not force attributes to appear in a single subcharacteristic; they can be part of many of them. The ISO/IEC standard allows this situation to occur.

Step 4: Decomposing derived attributes into basic ones

Some of the attributes emerging in Step 3 (for example, the Number of Languages Supported) can be directly measurable given a particular product, but others might still be abstract enough to require further decomposition. This is the case with the Quality of Graphical Interface attribute mentioned earlier; quality might depend on such factors as user friendliness, depth of the longest path in a browsing process, and types of interface supported. Therefore, we distinguish between *derived* and *basic* attributes. Derived attributes should be decomposed until they are completely expressed in terms of basic ones.

We can completely define derived attributes in terms of their components. However, in some situations, giving a concrete definition of the quality interface attribute could be considered harmful, because it would force us always to use the same definition without considering the requirements of a particular context.⁷ Sometimes, requirements give more importance to the user friendliness factor (for example, for nonskilled users), sometimes to its type (for interoperability purposes), and so on. In this case, the definition of the derived attribute is delayed until a particular selection process takes place. The first case of

To obtain a really complete quality model, you also must explicitly state the relationships between quality entities.

derived attributes is context free, the second is context dependent.

Step 5: Stating relationships between quality entities

To obtain a really complete quality model, you also must explicitly state the relationships between quality entities. The model becomes more exhaustive, and, as an additional benefit, the implications of quality user requirements might become clearer.

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

- *Collaboration.* Growing A implies growing B. For instance, the Security subcharacteristic collaborates with the Maturity one.
- *Damage.* Growing A implies decreasing B. For instance, the Type of Error Recovery Mechanism attribute collides with the Time Behavior subcharacteristic: the more powerful the mechanism is, the more slowly the program runs.
- *Dependency.* Some values of A require B fulfilling some conditions. For instance, having an exception-based error recovery mechanism requires that the programming language offer exception constructs.

In addition, you might build more elaborated types of these relationships.⁸

Step 6: Determining metrics for attributes

You must not only identify the attributes but also select metrics for all the basic attributes as well as metrics for those derived context-free attributes. You can use the general theory of metrics for this purpose. Also, ISO/IEC is currently working on writing the 9126-2 external metrics standard.⁹

Metrics for basic attributes are quantitative—for example, existence of some kind of data encryption, depth of the longest path in a browsing process, supported protocols for data transmission, and so on. Derived context-free attributes might be either quantitative or qualitative, with explicit formulas computing their value from their component attributes.

Some attributes require a more complex representation, yielding to structured metrics. Examples are sets (for example, a set of labels for the languages supported by the interface) and functions. Functions are especially useful for attributes that depend on the underlying platform. For instance, many attributes re-

lated to the Time Behavior subcharacteristic might fall into this category.

Metrics for some quality attributes can be difficult to define. However, as the standard states, having rigorous metrics is the only way to obtain a quality model useful for doing reliable comparisons.

A case study: Mail servers

As electronic mail services have grown in importance, companies are increasingly using them to improve inside and outside communication and coordination. An overwhelming number of mail-related products are 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. Selection relies on manufacturing documentation, public evaluation reports, others' experience, and hands-on experimentation. These information sources can be inaccurate, not fully trustable, and costly. For all these reasons, having a good quality model is especially useful in the domain of mail server packages, so we will use it to illustrate our general methodology (we will skip the preliminary step). The model is available at www.lsi.upc.es/dept/techreps/html/R02-36.html.

Step 1: Determining quality subcharacteristics

The subcharacteristics suggested in the 9126-1 standard are complete enough to be used as a starting point. We have adopted them with some minor modifications in their definition.

Step 2: Defining a hierarchy of subcharacteristics

According to the criteria mentioned earlier, we split the Suitability subcharacteristic into Mail Server Suitability and Additional Suitability subcharacteristics. Some examples of applications included in mail servers are chat, instant messaging, whiteboarding, videoconference, and workflow project management tools.

You might be tempted to apply this decomposition principle as often as possible, but you must do so carefully. Consider the following situation. In some contexts, you could consider the attributes categorized under the Operability subcharacteristic of Usability from two different viewpoints: the general user's and the administrator's. Because this is the

Table 2**Some relationships among quality attributes**

Characteristics			Functionality	Efficiency
	Subcharacteristics	Attributes	Security	Time behavior
			Secure email protocols	Average response time
Functionality	Security	Certification system	Depend on	Collide with
		Encryption algorithm	Depend on	Collide with
Reliability	Recoverability	Online incremental backup		Collide with
		Single-mailbox backup and recovery		Collaborate with
		Online restore		Collide with
		Dynamic log rotation		Collide with
		Event logging		Collide with
Efficiency	Resource behavior	Concurrent mail users per server		Collide with
		Number of active Web mail clients		Collide with
		Management of quotas on message and mail file size		Collaborate with
		Single copy store		Collaborate with

case for mail server products, we considered whether it was convenient to divide this subcharacteristic into two. But we observed that general user operability on mail servers depends on the mail client and the privileges given by the administrator. We did not find attributes related to clients that were independent of those related to administrators, so we decided to keep only one subcharacteristic.

Step 3: Decomposing subcharacteristics into attributes

We had to research the domain extensively to identify the attributes; then, we had to assign them to subcharacteristics. Contrary to what you might expect, this process is neither simple nor mechanical.¹⁰ Here are some of the problems you can run into:

- The number of elements can get very high, making handling them difficult.
- In some cases, the values of attributes can be confused with the attributes themselves.
- Sometimes attributes represent more than one concept and must be split. For example, we finally split the former Average Response Time attribute into two, the Average Response Time itself and Message Throughput.
- As mentioned earlier, some attributes are suited for more than one characteristic. For instance, Message Tracking and Monitoring might be seen as a functional attribute that grants Accuracy, or as an analyzability attribute of the Maintenance characteristic.

Hands-on experimentation is necessary to obtain really independent information. For in-

stance, the documentation of almost every product mentioned attributes such as administrative or expert analysis tools but gave very vague descriptions of them. We had some specific hands-on experience to better understand these concepts. These experiences turned out to be valuable to validate some results obtained in this step.

Step 4: Decomposing derived attributes into basic ones

We have identified several decomposable attributes. For instance, we decomposed the Resource Administration attribute (characteristic Usability, subcharacteristic Operability) into the following basic attributes: Maximum Storage Time of Mail Messages, Maximum Time of Life for Inactive Accounts, Mailbox Quotes, Mail File Sizes, and Management of Groups of Servers.

Step 5: Stating relationships between quality entities

With about 160 attributes, it is quite natural that there are a lot of relationships between them. Table 2 presents some dependencies, collaborations, and damages—the attributes in the rows *depend on*, *collaborate with*, or *collide with* attributes in the columns. For instance,

- If you select a certification system, you must also use an encryption algorithm, because it is needed to grant confidentiality.
- The SMTP (Simple Mail Transfer Protocol) requires the MIME (Multipurpose Internet Mail Extensions) Support attribute to be true when sending multimedia attachments.

Table 3**Example quality requirements**

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	Transmission time less than 1 minute for messages without attachments, and no more than 5 minutes per Mbyte for those with attachments

Step 6: Determining metrics for attributes

We determined metrics for all the basic and context-free attributes in the model, following the guidelines given earlier. We can evaluate some attributes by simple observation, such as Maximum Account Size and Default Folders Provided. Others are difficult to define. For example, Thwarting Spammers and Handling Bulk Junk Mail depend on the support of filters for incoming messages. Average Response Time and Message Throughput depend on hardware platform as well as other attributes such as Number of Concurrent Users or Message Sizes. In these cases, we must define the metrics as functions.

Package and requirement descriptions

Once we build the quality model for a package domain, we can describe packages in this domain and express the quality requirements for modeling a company's package procurement needs.

When we try to describe package quality characteristics, it turns out to be very difficult to find complete and reliable information on them. Manufacturers tend to give a partial view of their products. Either they put so much emphasis on their product's benefits, without mentioning weaknesses, or they give only part of the truth, making the product seem capable of more than it can really do. Some third-party reports look independent, but they have been refuted for the parties involved. Other noncommercial articles compare features but are often based on the evaluators' limited knowledge of the tools and their particular tastes, more than on serious technical tests.

The quality model can be used for describing quality requirements in a structured manner. In the mail server domain, we have introduced complete sets of quality requirements that have appeared in real mail server selec-

tion processes with very different characteristics (ranging from a public institution serving 50,000 people to a small software consultant and Internet service provider company).

Table 3 presents some requirements that illustrate typical situations we found when expressing requirements in terms of the quality model. Requirements such as 1 or 2 can be directly mapped into a single attribute of the model. The only difference is that Requirement 2 demands expert assessment to do the mapping from the expression "most commonly used certification standard" to a concrete value of the corresponding attribute, namely "X.509."

Requirements 3 and 4 are too general (what do "other applications" and "other risks" mean?). A more detailed specification is necessary to better classify them.

Requirement 5 either requires or implies a mixture of functionalities, which involve several attributes. Although we might need further feedback to better classify this kind of requirement, we succeeded in classifying this particular one.


Some requirements were originally expressed incorrectly but somehow were understandable. This was the case with Requirement 6, which was not accurate because the one-minute limit for messages without attachments could be unfeasible when they include a lot of data inline (for example, annual company reports). So, we reformulated it using the attributes' average response time and throughput, resulting in a requirement that could be satisfied.

Once you have incorporated all the requirements for a particular company into the model (after completing, discarding, and reformulating them), you can compare them extensively with respect to available package descriptions. This lets you detect differences between products as well as determine to what extent they cover the expressed needs, thereby facilitating the package procurement process. Once you have expressed all the requirements using the model, we recommend gathering feedback to refine and extend them.

Reliable processing of quality requirements demands a proper quality model to be used as a reference, especially in the context of software package se-

lection. Although building a quality model is complex,^{10,11} our methodology shows many advantages over ad hoc package evaluation:

- **Confidence.** Uniform, vendor-independent descriptions of numerous software packages facilitate package comparison. Also, rewriting quality requirements in terms of a model's concepts helps us discover ambiguities and incompleteness that, once solved, let us more easily compare requirements with package descriptions. Lastly, quality models obtained with our methodology can be expressed in a component description language,¹² making tool support for package selection feasible.
- **Productivity.** Consider the amount of repeated work that takes place in the mail server domain. Many organizations have faced exactly the same problems and have repeated the same processes, wasting human resources and money. The existence of a quality model of reference for this domain simplifies mail server procurement, once an organization's quality requirements have been expressed in terms of the reference model.
- **Experience and reusability.** Repeatedly using the same methodology and quality standard increases our model-building skills. Also, reusing parts of existing models in new domains becomes feasible, both for high-level subcharacteristics and for low-level attributes. We have confirmed this during the different experiences we have had in the domains of mail servers, ERP systems, e-learning tools, some component libraries, and so on.

Our work is compliant with the 9126-1 standard; we will aim at 9126-2 when that version is final. Also, our proposal can be used to support the evaluation and acquisition process defined in the 14598 standard, namely in the steps "Establish evaluation requirements" and "Specify the evaluation." 

Acknowledgments

This work is partially supported by the Spanish Ministry of Science and Technology under contract TIC2001-2165. Juan P. Carvallo's work has been supported by an Agencia Española de Cooperación Internacional (AECI) grant.

References

1. *Proc. 1st Int'l Conf. COTS-Based Software Systems (ICCBSS)*, Lecture Notes in Computer Science, no. 2255, Springer-Verlag, Berlin, 2002.
2. A. Finkelstein, G. Spanoudakis, and M. Ryan, "Software Package Requirements and Procurement," *Proc. 8th IEEE Int'l Workshop Software Specification & Design (IWSSD)*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 141–145.
3. J. Kontyo, "A Case Study in Applying a Systematic Method for COTS Selection," *Proc. 18th IEEE Int'l Conf. Software Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1996, pp. 201–209.
4. N. Maiden and C. Ncube, "Acquiring Requirements for COTS Selection," *IEEE Software*, vol. 15, no. 2, Mar./Apr. 1998, pp. 46–56.
5. X. Burgués et al., "Combined Selection of COTS Components," *Proc. 1st Int'l Conf. COTS-Based Software Systems (ICCBSS)*, Lecture Notes in Computer Science no. 2255, Springer-Verlag, Berlin, 2002, pp. 54–64.
6. *ISO/IEC Standard 9126-1 Software Engineering—Product Quality—Part 1: Quality Model*, ISO Copyright Office, Geneva, June 2001.
7. J. Bögh et al., "A Method for Software Quality Planning, Control, and Evaluation," *IEEE Software*, vol. 23, no. 2, Mar./Apr. 1999, pp. 69–77.
8. L. Chung et al., *Non-functional Requirements in Software Engineering*, Kluwer Academic Publishers, Dordrecht, Netherlands, 2000.
9. N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed., Int'l Thomson Computer Press, London, 1998.
10. R.G. Dromey, "Cornering the Chimera," *IEEE Software*, vol. 20, no. 1, Jan./Feb. 1996, pp. 33–43.
11. B. Kitchenham and S.L. Pfleeger, "Software Quality: The Elusive Target," *IEEE Software*, vol. 20, no. 1, Jan./Feb. 1996, pp. 12–21.
12. X. Franch, "Systematic Formulation of Non-functional Characteristics of Software," *Proc. 3rd IEEE Int'l Conf. Requirements Eng. (ICRE)*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 174–181.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.

About the Authors



Xavier Franch is an associate professor in the Software Department at the Universitat Politècnica de Catalunya in Barcelona, Spain. His interests are COTS component selection and evaluation, software quality, software process modeling and OO component libraries. He received his BSc and PhD in Informatics from the UPC. Contact him at Universitat Politècnica de Catalunya, c/ Jordi Girona 1-3 (Campus Nord, C6), E-08034 Barcelona, Spain; franch@lsi.upc.es; www.lsi.upc.es/~gessi.

Juan Pablo Carvallo is a doctoral candidate at the Universitat Politècnica de Catalunya. His interests are COTS-based systems development and COTS selection, evaluation, and certification. He received his degree in computer science from the Universidad de Cuenca, Ecuador. Contact him at Universitat Politècnica de Catalunya, c/ Jordi Girona 1-3 (Campus Nord, C6), E-08034 Barcelona, Spain; carvallo@lsi.upc.es.

