A Quality Model for the Ada Standard Container Library*

Xavier Franch and Jordi Marco

Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya
c/ Jordi Girona 1-3 (Campus Nord, C6) E-08034 Barcelona (Catalunya, Spain)
{franch,jmarco}@lsi.upc.es

Abstract. The existence of a standard container library has been largely recognized as a key feature for improving the quality and effectiveness of Ada programming. In this paper, we aim at providing a quality model for making explicit the quality features (those concerning functionality, suitability, etc.) that determine the form that such a library might take. Quality features are arranged hierarchically according to the ISO/IEC quality standard. We tailor this standard to the specific context of container libraries, by identifying their observable attributes and establishing some tradeoffs among them. Afterwards, we apply the resulting model to a pair of existing container libraries. As main contribution of our proposal, we may say that the resulting quality model provides a structured framework for (1) discussing and evaluating the capabilities that the prospective Ada Standard Container Library might offer, and (2) analyzing the consequences of the decisions taken during its design.

1 Introduction

Most important object-oriented programming languages include some standard libraries of reusable components as part of their definition. Among them, we focus on container libraries. A *container* (also known as *collection*) may be defined as an object that contains (i.e., stores) other objects. Some examples of containers are sets, maps and sequences. Object-oriented programming languages that offer container libraries are: Java, with the Java Collections Framework (JCF) [1]; C++, with the Standard Template Library (STL) [14]; and Eiffel, with the Eiffel Base Library [10].

Unfortunately, this is not the case for the Ada language, in spite of various attempts and claims in this direction, among which we mention:

- Some existing widespread container libraries, such as the Booch Components [3] and the Charles Container Library [4].
- Some events, such as the *Standard Container Library for Ada* workshop held during the Ada Europe 2002 conference.
- Wide initiatives, such as the Application Standard Components Library [2] or the prospective Working Group mentioned during the workshop above.

* Work partially supported by the Spanish research project CICYT TIC2001-2165.

Franch, X.; Marco, J. A quality model for the Ada standard container library. A: International Conference on Reliable Software Technologies. "Reliable Software Technologies: Ada-Europe 2003: 8th Ada-Europe International Conference on Reliable Software Technologies: Toulouse, France, June 16-20, 2003: proceedings". Berlín: Springer, 2003, p. 283-296.
The final authenticated version is available online at https://doi.org/10.1007/3-540-44947-7_21

- Some opinions and claims, such as those in the `comp.lang.ada` discussion list or the *ACM SIGAda* Chair Message for March 2002 *Ada Letters*: "Such a [Container] Library could be an excellent addition to the Ada International Standard".

Needless to say, the existence of a standard container library for Ada would clearly contribute to the quality of the final Ada artifacts and the effectiveness of the software development process itself. For this reason, and also because once deployed, its modification would be certainly difficult and should be avoided whenever possible, having a comprehensive, structured and precise framework for assessing the design and implementation of this library becomes utterly important.

With this objective in mind, we present in this paper a quality model aimed at making explicit all the quality criteria (e.g., reliability, functionality, efficiency etc.) that should be considered when building the Ada Standard Container Library. We use the ISO/IEC 9126-1 quality standard [9] as initial point. It is a very general standard, and so the main goal of the paper is tailoring the quality model proposed therein to the specific context of container libraries and also showing its use by means of the evaluation of two existing container libraries.

The rest of the paper is structured as follows. Section 2 presents the ISO/IEC 9126-1 quality standard, which introduces six groups of software characteristics that will drive the discussion. Sections 3 and 4 analyze some of the form that these characteristics take in the domain of container libraries, and Sect. 5 discuss their relationships. Section 6 applies the resulting quality model to two existing container libraries, an Ada one (Booch Components) and a standard library from other programming language (Java Collection Framework). Finally, Sect. 7 provides the main conclusions of our work.

2 The ISO/IEC 9126-1 Quality Standard

The ISO/IEC Quality Standard 9126-1 [9] provides a good framework for determining a quality model for a given domain of software components. An ISO/IEC-9126-1-based *quality model* is defined by means of general *characteristics* of software, which are further refined into *subcharacteristics*, which in turn are decomposed into *attributes*. Attributes collect the properties that software components exhibit. Intermediate hierarchies of subcharacteristics and attributes may appear making thus the model highly structured.

The ISO/IEC 9126-1 standard fixes six top level characteristics: functionality, reliability, usability, efficiency, maintainability and portability (see Table 1). It also fixes their further refinement into subcharacteristics but does not elaborate the quality model below this level, making thus the model flexible. The model is to be completed based on the exploration of the particular software domain and its application context; because of this, we may say that the standard is very versatile and may be tailored to domains of different nature, such as the one of container libraries.

Table 1. ISO/IEC 9126-1 characteristics and subcharacteristics

Functionality	
suitability	presence and appropriateness of a set of functions for specified tasks
accuracy	provision of right or agreed results or effects
interoperability	capability of the software product to interact with specified systems
security	prevention to (accidental or deliberate) unauthorized access to data
compliance	adherence to functionality-related standards or conventions
Reliability	
maturity	capacity to avoid failure as a result of faults in the software
fault tolerance	ability to maintain a specified level of performance in case of faults
recoverability	capability of reestablish level of performance after faults
compliance	adherence to reliability related standards or conventions
Usability	
understandability	effort for recognizing the logical concept and its applicability
learnability	effort for learning software application
operability	effort for operation and operation control
attractiveness	capability of the product to be attractive to the user
compliance	adherence to usability related standards or conventions
Efficiency	
time behavior	response and processing times; throughput rates
resource utilization	amount of resources used and the duration of such use
compliance	adherence to efficiency related standards or conventions
Maintainability	
analysability	identification of deficiencies, failure causes, parts to be modified, etc.
changeability	capability to enable a specified modification to be implemented
stability	capability to avoid unexpected effects from modifications
testability	capability to enable for validating the modified software
compliance	adherence to maintainability related standards or conventions
Portability	
adaptability	opportunity for adaptation to different environments
installability	effort needed to install the software in a specified environment
co-existence	capability to co-exist with other independent software in a common environment sharing common resources
replaceability	opportunity and effort of using software in the place of other software
compliance	adherence to portability related standards or conventions

In our paper, we are interested in considering the model as a means for obtaining quantitative measures of the attributes of the final product. Qualitative metrics would be defined later, when considering how the different attributes should be weighted for taking into account the requirements over the Ada Standard Container Library.

In the next sections, we adapt this quality model to the domain of container libraries. To do so, we apply a methodology presented elsewhere [6] that extends the hierarchy and makes explicit the relationships between the different quality features.

3 Quality Attributes for Functionality

Functionality is probably the most relevant quality characteristic in the domain of container libraries. Success of the Ada Standard Container Library requires exhibiting the appropriate functionality once considered its design requirements. It should be noted that "appropriate" does not necessarily mean "exhaustive", because an excess of functionality would impact negatively in other criteria such as usability or operability. Tradeoffs among quality factors, which are part of the quality model, would help to make this clear, see Sect. 5.

This section is structured into five subsections, one for each functionality subcharacteristic. For each subcharacteristic, we present a table with the attributes that play a part on them, together with some explanation.

3.1 Suitability

Not only in the domain of container libraries but also in others we have analyzed before [6, 5], *Suitability* is perhaps the largest and more complex subcharacteristic. For this reason, it is worth to decompose it into groups of attributes, i.e., new subcharacteristics. In our case, we identify two of them:

- *Core Suitability*. Addresses the types of containers offered and their implementations. These types are mainly characterised by the operations for adding, removing, modifying and searching elements.
- *General Suitability*. Keeps track of additional functionalities offered by (most of) the containers of the library, such as support for concurrent access or iterators.

Core Suitability. Table 2 introduces the attributes for *Core Suitability*. Some comments follow:

- The concept of *category* stands for huge groups of behaviour-related containers. Some categories such as sequences or maps will surely be present somehow in the Ada Standard Container Library, while others such as graphs may be a matter of discussion (in fact, most of the standard container libraries present in other languages do not offer this container category).
- Containers and their implementations are kept separated from the very beginning. A container represents an abstract data type, which may have (and probably will have) some different implementations in the library.
- The concept of *operation* should be viewed from the abstract-data-type point of view. In addition to operations, there could be generic algorithms that use those operations (container traversal, merging, etc.). These algorithms are represented by an attribute in the *General Suitability* subcharacteristic.
- It is important to include in the model itself the different types of container elements that are present in the library, and also their relationships, which are not shown in the table for lack of space.
- The second, the third and the fourth attributes are in fact families of attributes, one for each type of category or container.

Table 2. Attributes for *Core Suitability*

Attribute	Definition	Examples
Category variety	Range of different categories of containers offered by the library	Sequences, maps, sets, trees, graphs
Container variety	Range of different containers provided by every category	For sequences: stacks, queues, lists
Implementation variety	Range of different implementations provided by every category	For maps: closed hashing, red-black trees
Operation variety	Range of different operations provided by every container	For stacks: empty, push, pop, top, isEmpty
Element variety	Types of container elements	Universal, comparable

General Suitability. Table 3 lists the attributes for *General Suitability*. It should be mentioned that other attributes could also be incorporated, but we focused on the most usual ones:

- *Position* is a direct access path for elements in the container. *Iterators* provide a means to obtain and possibly manipulate the elements in the container. These two features are present in most widespread container libraries, with these names or others (e.g., iterators in STL, items in LEDA [13]).
- As some libraries do (e.g., STL and LEDA), positions and iterators may be implemented using the same mechanism. But it is important to keep both attributes separate, since their semantics are different.
- It is expected that these attributes will behave uniformly in the whole library, e.g., the same error management mechanisms used throughout the library (see 4.1).

Table 3. Attributes for *General Suitability*

Attribute	Definition	Examples
Direct access by position	Types and operations for supporting direct access to elements in containers	Type *position*; operation for deletion by position
Iterators	Types and operations for supporting traversal of containers	Bidirectional, unidirectional; read, read/write
Concurrent access	Mechanisms for managing concurrent access to containers	Semaphores, synchronization
Persistency	Mechanisms for storing container elements in a persistent manner	Serialization; operations for writing to disk; file types
Algorithmic variety	Range of generic algorithms present in the library or in particular containers	Sorting, merging For arrays: binary search
Error management	Mechanisms available for error management	Use of contracts, exceptions, messages
Sizeability	Strategies supported for managing the size of the container	Bounded, unbounded and resizeable containers

- An exception to the last rule is the *Sizeability* attribute, because some implementation strategies may prevent the use of some kind of sizeability strategies (e.g., heap implementations for resizeable containers).

3.2 Accuracy

In Table 4 we present the attributes for *Accuracy*. We highlight the following:

- It is necessary to distinguish among the *Error Management* mechanisms available (part of *General Suitability*) and how they are used in container operations. The first attribute in Table 4 addresses the last topic.
- Absence of ambiguity is analyzed at the specification level, not at the implementation one. For instance, it is important to know the policy that the container follows when elements with the same value are found in an ordered traversal. But on the other hand, it is not relevant to know the detailed behaviour of a concrete implementation (e.g., how a hashing strategy handle collisions), provided that it keeps the intended specification.
- Access by position and iterators must be well defined. A survey of these mechanisms in some widespread libraries shows that there are certain conditions that may affect the accuracy of the results and may compromise the integrity of the container. This is the reason why these new attributes have been introduced in this subcharacteristic.

Table 4. Attributes for *Accuracy*

Attribute	Definition	Examples
Trusted operations	Policies that ensure right results when executing operations	Deleting non-existing elements will not harm the state of the container
Absence of ambiguity	Certainty about the behaviour of a container	Priority queue: FIFO ordering of elements with the same priority
Accurate access by position	Policies and artifacts that ensure right results when accessing by position	Operation for knowing if a position is bound to the right element
Accurate access by iterator	Policies and artifacts that ensure right results when accessing by iterator	Operation for knowing if the current element during traversal has changed

3.3 Interoperability

Table 5 introduces the attributes for *Interoperability*. The second attribute refers to the possibility of using a container in a distributed system with some kind of middleware. To do so, some actions must be taken; e.g., the interface of the container should be defined in some specific Interface Description Language (IDL). Although a great deal of applications would avoid the remote use of containers for efficiency reasons, we think that the attribute must be included in the model for its analysis when designing the Ada Container Standard Library.

Table 5. Attributes for *Interoperability*

Attribute	Definition	Examples
Language interoperability	Ability to be invoked by programs written in a language different from Ada	Invocation from C++ and Eiffel
Component interoperability	Ability to be integrated in heterogeneous systems	CORBA, DCOM, RMI middleware

3.4 Security

Table 6 presents the attributes for *Security*. We remark that:

- *Concurrent Access Security* is closely related to the *Concurrent Access* attribute which belongs to the *General Suitability* characteristic. In other words, as it happened with error management in 3.2, it is important to distinguish among the available policies and how they are implemented from the security point of view.
- Since positions and iterators provide additional access mechanisms to containers (additional with respect to the access schemes bound to the type of container), it is important to establish their conditions of correct behaviour. Obviously, the two resulting attributes are closely related to the ones appearing in *Accuracy* but it is necessary to distinguish among accuracy of results and data security.

Table 6. Attributes for *Security*

Attribute	Definition	Examples
Concurrent access security	Policies and mechanisms that ensure safe concurrent access to elements in the container	Facilities for duplicating parts of the container; blocking during an iterator traversal
Direct access by position security	Policies and artifacts that ensure safe use of positions when accessing the container	A position bound to an element is the same while it is in the container
Iterator security	Policies and artifacts that ensure safe use of iterators when accessing the container	Read-only iterators may not be used in an odd manner

3.5 Functionality Compliance

Table 7 introduces the attributes for *Functional Compliance*. In general, the concept of compliance comes from two different sources. The first one are regulations clearly stated in a document, such as the reference manual of the Ada programming language. The second one stems from the current practices of the community, which have lead to a common foundation widely accepted. Some

different communities must be taken into account, mainly the Ada community (e.g., use the term *package* instead of *module*), the software libraries community (e.g., use the expression *browsing* when performing a tool-supported search in the library) and the data structure community (e.g., use names such as *stack* and *hashing*).

Table 7. Attributes for *Functionality Compliance*

Attribute	Definition	Examples
Domain compliance	Adherence to conventions of names of containers, operations and other artifacts	Function-like containers should be named something close to *map* or *table*
Language compliance	Adherence to conventions in the language community	In Ada: *To_List* for transforming a set into a list

4 Other Quality Attributes in Container Libraries

In the previous section we have analysed all the subcharacteristics belonging to the *Functionality* characteristic. The same should be done with the rest of characteristics of the ISO/IEC 9126-1 quality standard, but this is not possible in this paper for lack of space. So, we have focused on two other subcharacteristics that would play also an important part in the success of the Ada Standard Container Library.

4.1 Understandability

Table 8 introduces the attributes for *Understandability*. Some comments follow:

- A clear distinction among types of containers and their implementations is crucial for supporting understandability. Information hiding is the principle that should rule the design of the library. Therefore, no assumptions about implementation policies should appear when defining the type of container.
- Uniformity could be defined as a family of attributes, one for each type of concept in the library. Thus, we may talk about uniformity of error management, uniformity on the way of using generic algorithms, and so on.
- The third attribute is different from the compliance ones (see 3.5), although some relationships exist.
- Documentation has to be considered here from the user's point of view. This means that we are not addressing project documentation, such as code comments, but documentation for understanding the product.
- The quality of the design affects the understandability of a library. Bad designs may hide concepts and may place features in the wrong place.
- Complexity is a concept that has to be mainly with two factors: size (number of packages, number of methods, etc.) and conceptual difficulty of the implementations, algorithms, strategies, etc.

Table 8. Attributes for *Understandability*

Attribute	Definition	Examples
Separation between type of container - implementation	Degree of distinction among the semantics of a container type and its available implementations	No assumptions on the available implementations
Uniformity	Same strategies and level of detail when dealing with the same concept in different parts of the library	Access by position available to all types of containers
Name appropriateness	Behaviour of library features accordingly to their name	The *getCurrent* operation of an iterator does not change the current element
Quality of documentation	Appropriateness and comprehension of the documentation to make easy the use of the library	UML diagrams for describing the packages; browsing capabilities
Quality of design	Quality of the design of the library	Use of design patterns
Complexity	Size of the library and conceptual difficulty of the offered features	Use of advanced implementation techniques

4.2 Changeability

Table 9 presents the attributes for *Changeability*. Issues worth to remark:

- Changeability may be seen from different points of view, namely: extension of the library with new types of containers, implementations, generic algorithms and so on; specialization of existing types of containers with new features, new specific algorithms, etc.; modification of existing features. We could then split the subcharacteristic into three. However, for the sake of brevity, we do not proceed this way.
- Modularity and internal reusability are perhaps the key two factors in this subcharacteristic. Of course, whatever final form the library takes, some degree of modularity and reusability will exist.
- The last two attributes were previously introduced in the *Understandability* subcharacteristic. This situation illustrates the fact that the hierarchy has a graph-like form. However, it should be remarked that the focus of the attribute vary depending on the subcharacteristic. For instance, complexity in *Understandability* has been considered from the user point of view (basically, number of concepts to be understood) while complexity in *Changeability* is considered from the developer point of view (basically, how easy is the design and the code of the library to be modified). In other words, metrics for this two attributes would be definitively different.

5 Stating Relationships among Quality Attributes

We have already mentioned that a fundamental point when building a quality model is making explicit the tradeoffs among the different quality factors that

Table 9. Attributes for *Changeability*

Attribute	Definition	Examples
Modularity	Extent of the decomposition of the library into modules	One package for container type
Internal reusability	Degree of reusability of the code inside the library	Use of abstract classes
Programming practices	Adoption of best programming practices in-the-small	Avoid global variables; adopt name conventions
Decoupling	Independence of the different packages that are in the library	Use the *Template Method* pattern
Quality of design	Quality of the design of the library	Use of design patterns
Complexity	Difficulty of analysing the internal structure of the library	Intensive use of object-oriented features

have been identified. The purpose of this section is to identify and characterize the types of relationships that appear in the model.

Relationships may be classified according to two different concepts:

- Which are the types of quality factors related. The most usual case is relating a couple of attributes, but also subcharacteristics may be related. Also, a subcharacteristic may affect an attribute, or the other way round.
- Which is the kind of relationship. We distinguish among three types:
  - Dependency. If A depends on B, then A is an attribute that makes sense only if B satisfies some condition on its value.
  - Collaboration. If A collaborates with B, growing of A (from the metrics point of view) implies growing of B. Often, collaboration is symmetric.
  - Damage. If A damages B, growing of A (from the metrics point of view) implies shrinking of B.

As an example Table 10 enumerates some representative relationships among the quality factors presented in Sects. 3 and 4. Some explanation is given below:

- The first relationship is a dependency among two attributes. It states that talking about accuracy of iterators is useless when there are no iterators.
- The second one is a damage from one subcharacteristic to another. It reflects that the more suitable the library is, the more difficult to understand.
- The third relationship is a symmetric collaboration among two attributes. Modularity of the library clearly supports separation of concerns, while having a conceptual distinction among type of container and implementation will favour the ability to structure the library in a modular manner.
- Finally, the fourth relationship is a collaboration from one attribute to another. Internal reusability clearly has a positive influence on uniformity of the library, since inherited features will appear in the heirs without any difference.

Table 10. Relationships among quality factors

Dependee	Depender	Type of relationship
Iterators	Accurate access by iterator	Dependency
Suitability	Understandability	Damage
Modularity	Separation type of container – implementation	Symmetric collaboration
Internal reusability	Uniformity	Collaboration

It should be remarked that the analysis of the relationships among quality attributes requires adopting a qualitative point of view of the quality model. As stated in the introduction, qualitative measures are supposed to be introduced when considering a specific context for the library, after analysis of the requirements for its design and implementation. The relationships stated here could be then a great help when considering the implications of some requirements over the quality of the library.

6 Applying the Quality Model

In this section we apply the resulting quality model to two existing container libraries: an Ada one, Booch Components (BC) and the standard library from Java, the Java Collections Framework (JCF). Due to lack of space, we do not analyze other interesting Ada libraries (such as Charles) or standard libraries of other languages (e.g., STL).

Table 11 summarizes the evaluation of the attributes in both libraries. The first column identifies the subcharacteristic, the second one the attribute, and the third and fourth the values in the libraries. The table is commented in the rest of the section.

Core suitability. It can be observed that BC provides a richer variety of categories and containers, but there is not a significant difference in implementation strategies. Operations on containers are usual ones: insertions, deletions, etc. Some containers and implementation strategies require comparisons of elements and so both libraries offer comparable elements.

General suitability. Both libraries are not well-suited for access by position. BC does not provide this kind of access, while JCF just in lists. Also, iterators are not as powerful as in other widespread libraries. In fact, we have a proposal [11] enriching BC with access by position and more powerful iterators.

It must be also remarked that the direct access provided by JCF by means of list iterators is in fact the position on the list, so in case of *LinkedList* implementation it is highly inefficient. This property would have appeared if efficiency were included in the part of the quality model developed in this paper. On the other hand, the list iterator are bidirectional and read/write.

Other general suitability attributes are more or less covered by both libraries.

Table 11. Applying the quality model to the BC and JCF libraries

	Attribute	BC	JCF
Understandability	Separation type of container – implem.	Yes	Yes
	Uniformity	Yes	Not entirely
	Name appropriateness	Yes	Not entirely
	Quality of documentation	Good	Good
	Quality of design	Yes	Not entirely
	Complexity	Low	High
Changeability	Modularity	Yes	Yes
	Internal reusability	Yes, but causes coupling	Yes, but causes inefficiency
	Programming practices	Good	Good
	Decoupling	No	Yes
	Quality of design	Coupling causes lost of quality	Good
	Complexity	High	High
Core Suitability	Category variety	Sequences, maps, trees, sets, graphs	Sequences, maps, sets
	Container variety	Sequences: collections, lists, stacks, rings, queues; Maps: maps; etc. up to 12 varieties	Sequences: lists; Maps: maps, sorted maps; Sets: sets, sorted sets
	Implementation variety	One strategy for each container	Lists amb maps, two strategies; others, one
	Operation variety	Most common operations for each container	Most common operations for each container
	Element variety	Universal, comparable	Universal, comparable
General Suitability	Access by position	Not supported	Only in lists, using iterators
	Iterators	Unidirectional, read-only	Unidirectional (except for lists), read-and-remove
	Concurrent access	Guarded and synchronized	Not specific mechanisms
	Persistency	Not supported	Serialization
	Algorithmic variety	No generic algorithms	No generic algorithms
	Error management	Exception mechanism	Exception mechanism
	Sizeability	Bounded, unbounded and resizeable	Unbounded, resizeable
Accuracy	Trusted operations	Yes, using exceptions	Yes, using exceptions
	Absence of ambiguity	Yes	Yes
	Accurate direct access by position	Direct access not supported	Lists invalidate direct access as soon as the container is modified
	Accurate access by iterator	Yes, using concurrent access mechanisms	An iterator is invalidated when the container is modified by other means than the iterator itself
Interopera.	Language interoperability	No language interoperability	No language interoperability
	Component interoperability	No component interoperability	No component interoperability
Security	Concurrent access security	Yes, using concurrent access mechanisms	Controlled by exception mechanisms
	Direct access by position security	Direct access not supported	Positions provided by lists not persistent
	Iterator security	Safe read-only iterators	Control of modifications
F.C.[1]	Domain compliance	Good enough	Good enough
	Language compliance	Good enough	Good enough

[1] Functionality Compliance

Accuracy. Exception handling is widely used for assuring correct results. Concerning iterators and direct access, JCF has a safe although somehow restrictive policy of invalidating iterators when other operations interfere.

Interoperability. None of the libraries offer interoperability capabilities. It could be thought of building wrappers, using either the *Export* pragma and *Remote Call Interface* in the case of Ada, or the CORBA and RMI standards in the case of Java.

Security. In both cases, the security aspects are well covered.

Functionality compliance. In both cases, the libraries are compliant enough with respect to functionality, although from time to time some odd terms appear.

Understandability. It is worth to remark some lack of uniformity in the JCF library. For instance, JCF bidirectional iterators and access by position are only available for lists. Also, some of the JCF method names are not according to their functionality (e.g. the *next* iterator method return the current element and then advances to the next).

The design of the JCF library is not as clear as it should be. For instance, there are two separate hierarchies; this makes more difficult to get the general picture.

With respect to complexity there are different tradeoffs between the two libraries. Basically, BC offers few methods that can be combined to obtain more functionalities while JCF has a lot of different methods some of them with similar functionalities and so the library becomes more difficult to understand.

Changeability. Both libraries present internal reusability up to some extent, but their solutions are not optimal. In the case of BC, some dependencies among packages arise and makes changes more difficult to implement (see [11] for details). This coupling also impacts on other attributes. In the case of JCF, some methods of its abstract classes are implemented using only the interface of the class (avoiding coupling) but as a consequence they are not as efficient as possible, and the size of each category abstract class grows.

7 Conclusions

In this paper, we have proposed a framework based on the concept of quality model for assessing the quality of the prospective Ada Standard Container Library. The quality model is based on the ISO/IEC 9126-1 quality standard which fits well with the fact that Ada is an ISO standard too. The use of such a quality standard to design an Ada Standard Container Library is an important aspect to make easier the standardization of the new Ada language extensions and evolutions. In the paper, we have tailored the highly abstract quality model proposed in this standard to our specific needs, by adding some quality factors and exploring their relationships. Also, we have checked the applicability of the model to a pair of existing libraries. For the sake of brevity, we have not presented the whole quality model, but a (representative enough) part.

Our paper tries to be a contribution in the design and implementation of the Ada Standard Container Library. It is our believe that the success of this potential library requires as a first step a deep knowledge of the features that must be considered for their inclusion, the different ways on offering these features, and also the consequences of including them. Mechanisms such as shortcuts [7], specific design patterns [8, 12] and others may be then more thoroughly evaluated. The quality model may act as a cornerstone in these activities.

But adopting a quality-model-based approach in not only useful for assessment. It provides a structured and precise way of describing the features of the library, making its use more appealing. It also can be used as a first step for getting a certification of the resulting product, issued by a third-party organization; there are currently some organizations that use the ISO/IEC quality standards for issuing such certifications. Last, a quality model supports traceability of decisions taken during the construction of the library, which is specially interesting when considering maintenance of the library.

References

1. K. Arnold, J. Gosling and D. Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2000.
2. Application Standard Components Library (ASCL). http://ascl.sourceforge.net/.
3. G. Booch, D.G. Weller and S. Wright. The Booch Library for Ada 95 (version 1999). Available at http://www.pogner.demon.co.uk/components/bc/.
4. M. Heaney. Charles Container Library. At home.earthlink.net/∼matthewjheaney/.
5. J.P. Carvallo, X. Franch and C. Quer. Defining a Quality Model for Mail Servers. In *Proceedings of the 2nd International Conference on COTS-Based Software Systems (ICCBSS)*, volume 2580 of *LNCS*. Springer-Verlag, 2003.
6. X. Franch and J.P. Carvallo. Using quality models in software package selection. *IEEE Software*, **20(1)**, 2003.
7. X. Franch and J. Marco. Adding Alternative Access Paths to Abstract Data Types. In *Challenges of Information Technology Management in the 21st Century (IRMA'2000)*, pages 283-287. Idea Group Publishing, 2000.
8. N. Gelfand, M. T. Goodrich and R. Tamassia. Teaching Data Structure Design Patterns. In *Proceedings of ACM SIGCSE '98*, 1998.
9. ISO/IEC Standards 9126-1 Software Engineering – Product Quality – Part 1: Quality Model, June 2001.
10. B. Meyer. *Reusable Software: the base object-oriented component libraries*. Prentice Hall, 1994.
11. J. Marco and X. Franch. Reengineering the Booch Component Library. In *Reliable Software Technologies Ada-Europe 2000*, volume 1845 of *LNCS*, pages 96-111. Springer-Verlag, 2000.
12. J. Marco and X. Franch. Shortcuts for the Ada Standard Container Library. Presented at the *Workshop for Standard Container Library for Ada*. Held during the Ada-Europe 2002 Conference, Wien (Österreich). Available at http://www.auto.tuwien.ac.at/AE2002/resources.html.
13. K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
14. D.R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.