A Framework for Designing and Implementing the Ada Standard Container Library

Jordi Marco
jmarco@lsi.upc.es

Xavier Franch
franch@lsi.upc.es

Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
c/ Jordi Girona 1-3 (Campus Nord, C6)
E-08034 Barcelona (Catalunya, Spain)

ABSTRACT
An open issue of the Ada language is the definition of a standard container library. Containers in this library (e.g., sets, maps and lists) shall offer some core functionalities that characterise their behaviour (i.e., different strategies for managing the elements stored therein) as well as other general functionalities. Among these general functionalities, we are interested in alternative ways for accessing the containers, namely direct access by position and traversals using iterators. In this paper, we present the Shortcut-Based Framework (SBF), a framework aimed at providing suitable, uniform, accurate and secure access by position and iterators, while keeping other nice properties such as comprehensibility and changeability. The SBF should be considered as a baseline upon which the Ada standard container library can be built. We assess the feasibility of our proposal defining a quality model for container libraries and evaluating the SBF using some metrics defined with the Goal-Question-Metric approach.

Categories and Subject Descriptors
D.2.2 [Software Engineering]: Design Tools and Techniques—Software libraries, Object-oriented design methods; D.2.13 [Software Engineering]: Reusable Software—Reusable libraries; D.3.3 [Programming Languages]: Language Constructs and Features—Frameworks, Data types and structures; D.2.8 [Software Engineering]: Metrics—Product metrics.

General Terms
Design, Measurement.

Keywords
Container libraries, Iterators, Access by position, Quality models.

1. INTRODUCTION
Most important object-oriented (OO) programming languages include some standard libraries of reusable components as part of their definition. One kind of such libraries are container libraries. A container (also known as collection) may be defined as an object that contains (i.e., stores) other objects; examples of containers are sets, maps and sequences. Some of the most well-known container libraries in the OO world are: the Java Collections Framework (JCF) [1] for Java; the Standard Template Library (STL) [18] for C++; and the Eiffel Base Library [13] for Eiffel.

Unfortunately, the Ada language does not provide such a standard container library, in spite of various attempts and claims in this direction, among which we mention:

• Some existing widespread container libraries, as the Charles Container Library [5], Booch Components [4], etc.

• Some events, for instance the Standard Container Library for Ada workshop held during the Ada Europe 2002 conference.

• Some wide initiatives, e.g. the Application Standard Components Library [2] or some action items issued by the Ada Conformity Assessment Authority (remarkably the action item AI-302).

• Some opinions and claims, like those in the discussion list at comp.lang.ada 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 artefacts and the effectiveness of the software development process itself. For this reason, having a comprehensive, structured and precise framework for designing and implementing this library becomes utterly important, and this is the objective of the present paper.

We propose in this article a framework [6, 12] (see Sect. 3) named Shortcut-Based Framework (SBF) to drive the design and implementation of the prospective Ada Standard Container library. The SBF is not intended to be a closed proposal of the contents of the library; instead, it has been defined as a baseline upon which a high-quality Ada Standard Container library can be developed. Software quality

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGAda'03, December 7–11, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-476-2/03/0012 ...$5.00.
is always important, but it is even more crucial in standard packages, intended to be used without changes once deployed for a long period of time. In order to measure the quality of the resulting library, we have already defined elsewhere [8] a quality model for this kind of libraries built upon the ISO/IEC 9126-1 Quality Standard [11], summarized in Sect. 2. The SBE-based Ada Standard Container library we are proposing is evaluated with respect to this quality model in Sect. 4.

2. A QUALITY MODEL FOR THE ADA STANDARD CONTAINER LIBRARY

In this section we summarise an ISO-based quality model for the domain of container libraries. First we present the ISO/IEC Quality Standard 9126-1 [11] on which this quality model is based and then a brief description of some relevant quality attributes that are of interest in our proposal. A more exhaustive description of the model has been presented at Ada-Europe 2003 [8].

2.1 The ISO/IEC 9126-1 Quality Standard

The ISO/IEC Quality Standard 9126-1 [11] provides an appropriate and widespread 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 measurable 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.

2.2 An ISO/IEC-based Quality Model for Container Libraries

Functionality is probably the most relevant quality characteristic in the domain of container libraries. Success of the prospective 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.

Table 2 shows the attributes (name, definition and examples) that play a part on some functionality subcharacteristics. It is worth to remark that the Suitability subcharacteristic has been decomposed into two groups of attributes, i.e., two new subcharacteristics:

- 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.

<table>
<thead>
<tr>
<th>Table 1: ISO/IEC 9126-1 quality standard</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functionality</strong></td>
</tr>
<tr>
<td>suitability</td>
</tr>
<tr>
<td>accuracy</td>
</tr>
<tr>
<td>interoperability</td>
</tr>
<tr>
<td>security</td>
</tr>
<tr>
<td>compliance</td>
</tr>
<tr>
<td><strong>Reliability</strong></td>
</tr>
<tr>
<td>maturity</td>
</tr>
<tr>
<td>fault tolerance</td>
</tr>
<tr>
<td>recoverability</td>
</tr>
<tr>
<td>compliance</td>
</tr>
<tr>
<td><strong>Usability</strong></td>
</tr>
<tr>
<td>understandability</td>
</tr>
<tr>
<td>learnability</td>
</tr>
<tr>
<td>operability</td>
</tr>
<tr>
<td>attractiveness</td>
</tr>
<tr>
<td>compliance</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
</tr>
<tr>
<td>time behavior</td>
</tr>
<tr>
<td>resource utilization</td>
</tr>
<tr>
<td>compliance</td>
</tr>
<tr>
<td><strong>Maintainability</strong></td>
</tr>
<tr>
<td>analysability</td>
</tr>
<tr>
<td>changeability</td>
</tr>
<tr>
<td>stability</td>
</tr>
<tr>
<td>testability</td>
</tr>
<tr>
<td>compliance</td>
</tr>
<tr>
<td><strong>Portability</strong></td>
</tr>
<tr>
<td>adaptability</td>
</tr>
<tr>
<td>installability</td>
</tr>
<tr>
<td>co-existence</td>
</tr>
<tr>
<td>replaceability</td>
</tr>
<tr>
<td>compliance</td>
</tr>
</tbody>
</table>

- General Suitability. Keeps track of additional functionalities offered by (most of) the containers of the library, such as support for concurrent access or iterators.

For the other subcharacteristics of functionality, we focus on accuracy and security of access by position and access by
Table 2: Quality Attributes for some Functionality Subcharacteristics

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category variety</td>
<td>Range of different categories of containers offered by the library</td>
<td>Sequences, maps, sets, trees, graphs</td>
</tr>
<tr>
<td>Container variety</td>
<td>Range of different containers provided by every category</td>
<td>For sequences: stacks, queues, lists</td>
</tr>
<tr>
<td>Implementation variety</td>
<td>Range of different implementations provided by every category</td>
<td>For maps: closed hashing, red-black trees</td>
</tr>
<tr>
<td>Operation variety</td>
<td>Range of different operations provided by every container</td>
<td>For stacks: empty, push, pop, top, isEmpty</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct access by</td>
<td>Types and operations for supporting direct access to elements in containers</td>
<td>Type position; operation for deletion by position</td>
</tr>
<tr>
<td>position</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterators</td>
<td>Types and operations for supporting traversal of containers</td>
<td>Bidirectional, unidirectional; read, read/write</td>
</tr>
<tr>
<td>Algorithmic variety</td>
<td>Range of generic algorithms present in the library or in particular containers</td>
<td>Sorting, merging. For arrays: binary search</td>
</tr>
</tbody>
</table>

3. THE SHORTCUT-BASED FRAMEWORK

As mentioned in the introduction our main objective is not to propose a concrete collection of containers to be included in the Standard Container Library for Ada. Rather we propose the use of a concrete framework for the design of this library: the Shortcut-Based Framework [16]. This framework allows solving the majority of drawbacks (with respect to the quality criteria presented in Sect. 2) that are present in the most widespread container libraries [15]. This is achieved by means of the Shortcut concept that we propose at the core of our framework.

3.1 The Shortcut Concept

Shortcuts encapsulate the feature of location or position of an object in a container. Shortcuts provide an abstract, reliable and efficient (all the operations have $O(1)$ as order of complexity) alternative access path to the elements stored in the container. Most of the existing container libraries recognise the need for such a kind of alternative access method and thus they have similar mechanisms but they are ad-hoc implementation-dependent and not totally reliable proposals (e.g., iterator in STL [18], references in JCF [1], item in LEDA [17], location in JDSL [9], etc.). Instead, we provide an implementation-independent approach based on the use of shortcuts to implement a generic container which acts as a base class of the rest of concrete containers. Shortcuts allow implementing only once, in the base class, the most common capabilities (e.g., iterators) in a highly efficient and reliable way. The implementation of both the shortcuts and the common capabilities are decoupled from the details of the implementation of its inheritors.

3.2 The Shortcut-Based Framework

In this section we outline the main features of the Shortcut-Based Framework (SBF) together with some implementation details. The key point consists on storing the objects of any concrete container in a Container class, while keeping in the concrete container only the shortcuts bound to them. We show throughout this section the complete development of this idea.

First, we define the SBF hierarchy for container libraries. This hierarchy has been built borrowing the main ideas from the different hierarchies of some widespread container libraries [4, 18, 17]. Since, we are not interested in fixing all the details of the hierarchy (i.e., which concrete containers, and which concrete operations in them, do exist) the
Table 3: Other Important Quality Attributes

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Changability</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal reusability</td>
<td>Degree of reusability of the code inside the library</td>
<td>One package for container type</td>
<td></td>
</tr>
<tr>
<td>Programming practices</td>
<td>Adoption of best programming practices in-the-small</td>
<td>Use of abstract classes</td>
<td></td>
</tr>
<tr>
<td>Decoupling practices</td>
<td>Independence of the different packages that are in the library</td>
<td>Avoid global variables; adopt name conventions</td>
<td></td>
</tr>
<tr>
<td>Quality of design</td>
<td>Quality of the design of the library</td>
<td>Use the Template Method pattern</td>
<td></td>
</tr>
<tr>
<td>Complexity</td>
<td>Difficulty of analysing the internal structure of the library</td>
<td>Intensive use of object-oriented features</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Time behavior</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order of magnitude</td>
<td>Worst-case execution time of the operations of an implementation as the size of their input grows</td>
<td>Constant time on insertion, $O(1)$, linear time on removal, $O(n)$</td>
<td></td>
</tr>
<tr>
<td>Real time efficiency</td>
<td>Execution elapsed time of (a pattern of) the operations of an implementation with selected benchmarks</td>
<td>Number of ms. for list traversal with 20,000 elements in a given platform</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Resource utilization</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data structure order of magnitude</td>
<td>Worst-case amount of storage required by the data structure of an implementation as the number of elements grows</td>
<td>Linear space for a hashing strategy, $O(n)$</td>
<td></td>
</tr>
<tr>
<td>Data structure real space efficiency</td>
<td>Number of bytes required by the data structure of an implementation given the space required to represent the elemental data types</td>
<td>$n \times$ the space of a pointer</td>
<td></td>
</tr>
<tr>
<td>Order of magnitude of the operations</td>
<td>Worst-case amount of auxiliary storage required by the operations of an implementation as the number of elements in its data structure grows</td>
<td>Linear space for a sorting algorithm</td>
<td></td>
</tr>
</tbody>
</table>

framework should include just its general layout. A complete hierarchy of a container library shall provide: a container base class (where common capabilities are offered); a hierarchy of iterators; locations for accessing and modifying containers; concrete containers classes; and different implementation strategies for each concrete container. Our objective is to reuse in the concrete containers classes the common capabilities of the (fully-implemented) container base class. Figure 1 shows the hierarchy we have chosen for the SBF.

The classes involved in this hierarchy are:

- **BidirectionalIterator.** An abstract class that provides the interface of this kind of iterator, i.e. iterators that support forward and backward traversal of a container.
- **ContainerIterator.** An efficient implementation of BidirectionalIterator enriched with a new method that returns the shortcut bound to the current item of the iterator. This class is implemented over the base class Container; as a consequence, it is fully independent of the specific kind of container. All operations of this class shall be $O(1)$ to guarantee highly-efficient access by iterator.
- **Shortcut.** Defines the interface of the the concept of shortcut which encapsulates the feature of location or position of objects.
- **ContainerShortcut.** An efficient (i.e., $O(1)$ time) and secure implementation of the Shortcut interface. ContainerShortcut is implemented over the base class Container; as a consequence, it is fully independent of the specific kind of container but can be used for access to items them store.
- **Container.** This base class acts as a common parent class for all kinds of containers. It provides the interface and implementation of the most common capabilities of container libraries.
- **Concrete containers.** Children classes of Container that are not leaves, which represent different types of containers (list, map, etc.). Each of them adds the interface and implementation of its specific functionalities to the ones inherited from the Container class. The strategy chosen to implement these classes consists on storing the items in the base class Container and the shortcuts bound to them in a concrete implementation (an array, dynamic storage, ...). In order to do this, specific operations of concrete containers are implemented using (if it is necessary) an operation implemented by their subclasses (i.e., using the Template Method design pattern). Concrete containers also define as protected the interface of the de-
3.3 Implementation Details

The essential point consists in maintaining an efficient mapping from shortcuts to items in the Container base class. There are three possibilities to implement this mapping depending on the underlying memory management scheme:

1. Using dynamic storage without garbage collection. The smart pointer shall be maintained in the base class Container as a black box for its children classes. Moreover, all the operations of the container class are O(1). Last but not least, each concrete container is an implementation (non-abstract) class.

   • Concrete implementations. Children classes of concrete containers that are leaves. These classes implement the concrete interface by means of data structures. They inherit all the functionalities of the concrete container and as a consequence their implementation can be made avoiding iterators and locations. On the other hand, inherited implementations may remain if they already fulfill efficiency requirements.

Picture 1: The Shortcut-Based Framework

As a sample of these three schemes, Figure 2 shows the array-based one.

![Figure 2: Array-based implementation scheme](image)

It must be remarked that shortcut assignment is properly managed in both schemes. Using dynamic storage, the class defining smart pointers redefines the assignment to detect that a new shortcut has come up. Using an array, normal behaviour of assignment is sufficient.

It can be observed that shortcuts require some extra time and space in order to get all its benefits. We will assess the behaviour of assignment is sufficient.

2. Using dynamic storage with garbage collection. The smart pointer shall be substituted by a regular pointer. There are no more changes with respect the previous approach.

3. Using an array. In this case shortcuts shall be implemented as tuples of an index to the array position and an object identifier which allows check if the object corresponding to the index position of the array has changed. Then the Container base class shall be implemented as an array of tuples which contain the object, a item identifier (counts the updates of the array position) and two indexes to the next and previous tuples linking the elements in the iterator ordering. As released shortcuts (the deleted array positions) shall be available somehow to allow further reassignment, we shall link them too. Additional index members corresponding to the position of the first object, the position of the last object and the first free position shall be maintained in the Container base class.

Examples of defined Shortcuts

<table>
<thead>
<tr>
<th>Index: 0</th>
<th>Index: 3</th>
<th>Index: 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjectId: 0</td>
<td>ObjectId: 0</td>
<td>ObjectId: 0</td>
</tr>
<tr>
<td>N-1</td>
<td>N-1</td>
<td>N-1</td>
</tr>
<tr>
<td>Prev: N-1</td>
<td>Prev: N-1</td>
<td>Prev: N-1</td>
</tr>
</tbody>
</table>

Examples of not defined Shortcuts

<table>
<thead>
<tr>
<th>Index: 0</th>
<th>Index: 3</th>
<th>Index: 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjectId: 0</td>
<td>ObjectId: 0</td>
<td>ObjectId: 0</td>
</tr>
<tr>
<td>N-1</td>
<td>N-1</td>
<td>N-1</td>
</tr>
<tr>
<td>Prev: N-1</td>
<td>Prev: N-1</td>
<td>Prev: N-1</td>
</tr>
</tbody>
</table>

†Defined Shortcuts are those which ObjectId is equal to the ObjectId of the corresponding array position

![The complete code can be found at www.lsi.upc.es/~jmarco/SBFAdaImplementation.zip](image)
remain valid even in data structures that make rearrangement of elements. The specifications and implementations of the ContainerShortcut and the ContainerIterator are in the containers specification and body packages. The implementation of shortcuts is made using dynamic memory.

Package specification of the Container class (in file containers.ads). This package contains the specification of shortcuts, iterators and containers. We use a Pointer class from the SmartPointer package, which encapsulates the definition of smart pointers. The implementation of the Shortcut and Iterator classes is the same, it stores a smart pointer Ptr to the container node where the item bound to the shortcut or to the iterator is, along with an access to the Container. Notice that in the Delete operation of the Container class, the Shortcut parameter is in out in order to set its smart pointer to NULL for decrementing the number of references to the element removed.

with Ada.Finalization; 
with SmartPointer; 
withSmartPointerTo; 
with Bidirectional_Iterators; 
generic 
type Item is private; 
package Containers is 
type Container is tagged private; 
function Item_Of (Sh: Shortcut) return Item is 
begin 
if IsNull(Sh.Ptr) then return false; 
else return not SP.Value(Sh.Ptr).Deleted_Flag; 
end if; 
end Item_Of; 
function Defined (Sh : Shortcut) return Boolean is 
begin 
if Defined(Sh.Ptr) then return true; 
else return not SP.Value(Sh.Ptr).Deleted_Flag; 
end if; 
end Defined; 

Operations of the ContainerShortcut class. This code is in the Containers package body. We do not present here the "=" function which implementation is straightforward. Notice that both operations below have constant time cost.

function Item_Of (Sh : Shortcut) return Item is 
begin 
if not Defined(Sh.Ptr) then 
raise Undefined_Shortcut; 
elsif Sh.Container = NULL then return Item; 
else return not SP.Value(Sh.Ptr).Deleted_Flag; 
end if; 
end Item_Of; 
function Defined (Sh : Shortcut) return Boolean is 
begin 
if IsNull(Sh.Ptr) then return false; 
elsif Sh.Container = NULL then return false; 
else return not SP.Value(Sh.Ptr).Deleted_Flag; 
end if; 
end Defined; 

Operations of the ContainerIterator class. This code is in the Containers package body too. We only show the Swap procedure which interchange the iterator order of the items associated to the two iterators, and the CurrentShortcut function which returns the Shortcut associated to the current item. Smart pointer assignment inside this method keeps track of the existence of a new shortcut to the involved element (this also happens in iterator assignment); also, the container bound to the shortcut is the container bound to the iterator through BindToContainer. The rest of operations are the usual ones for iterators and are implemented straightforwardly. As in the case of the Shortcut class, all the operations need O(1) time in the worst case.

function CurrentShortcut (It : Iterator'Class) return Shortcut is 
begin 
if IsDone(It) then 
raise Iterator_Out_Of_range; 
end if; 
return 'CurrentShortcut'(Ada.Finalization.Controlled with Container => It.Container, Ptr => It.Ptr); 
end CurrentShortcut; 
procedure Swap(It1, It2 : in out Iterator) is 
PtnAux : SmartPointer.Pointer; 
begin 
if not IsDone(It1) and not IsDone(It2) then 
if Value(It1.Ptr).Next /= It2.Ptr and 
Value(It2.Ptr).Next /= It1_Ptr then 
raise Undefined_Shortcut; 
elsif Value(It1.Ptr).Next = It2.Ptr and 
Value(It2.Ptr).Next = It1_Ptr then 
raise Undefined_Shortcut; 
else 
Swap(It1.Ptr).Next := It2.Ptr; 
Swap(It2.Ptr).Next := It1.Ptr; 
PtnAux := Value(It1.Ptr).Next; 
AccessValue(PtnAux).Next := Value(It2.Ptr).Next; 
end if; 
end if; 
end Swap;
AccessValue(It2_Ptr).Next := It1_Ptr;
AccessValue(It1_Ptr).Previous := It2_Ptr;
else
 AccessValue(Value(It2_Ptr).Previous).Previous :=
 Value(Value(It2_Ptr).Previous).Next := It2_Ptr;
end if;

begin

procedure Add (In_The_Container : in out Container;
Elem: Item) is
Ptr := Create;
AccessValue(Ptr).Elem := Elem;
AccessValue(Ptr).Previous := In_The_Container.Dummy;
In_The_Container.Cardinality := In_The_Container.Cardinality+1;
In_The_Container.Last_Item_Added := Ptr;
end Add;

procedure Delete (In_The_Container : in out Container;
The_Shortcut: in out Shortcut'Class) is
begin
if Defined(The_Shortcut) then
if The_Shortcut.Container = In_The_Container then
 AccessValue(The_Shortcut.Ptr).Deleted_Flag := true;
 SetNull(AccessValue(The_Shortcut.Ptr).Next);
 SetNull(AccessValue(The_Shortcut.Ptr).Previous);
end if;
end if;
end Delete;

The Delete map operation uses Dispatching operations in order to call the concrete operations of the implementation class that has called Delete.

procedure Dispatching_Delete (In_The_Container : in out Map'Class; The_Key: Key) is
begin
Con_Delete(In_The_Container, The_Key);
end Dispatching_Delete;

procedure Delete (In_The_Container : in out Map;
The_Key: Key) is
begin
Sh := Dispatching_Get(In_The_Container, The_Key);
Delete(In_The_Container, Sh);
end Delete;

Con_Error : exception;
Not_Existing_Key : exception;

The use of parent and children classes as black boxes.

Now, we consider a possible concrete implementation, named MapArray for the concrete container Map. We have chosen this example because it shows the main points of the SBF:

• The application of the Template Method design pattern.

• The possibility of defining generic algorithms that work over any kind of containers.
implementation of the `Map` includes the code for the deferred concrete operation `ConDelete`. It is worth to remark that this implementation changes positions of elements in the array; in despite of these rearrangements, all the existing shortcuts and iterators keep being valid.

```ada
procedure Con_Delete(In_The_Container: in out MapArray; The_Key: Con_Key) is
begin
  if not Con_Exist(In_The_Container, The_key) then
    raise Not_Existing_Key;
  end if;
  In_The_Container.FirstFree :=
    In_The_Container.FirstFree - 1;
  for i in In_The_Container.Cache ..
    In_The_Container.FirstFree-1 loop
    In_The_Container.MapA(i) :=
      In_The_Container.MapA(i+1);
  end loop;
  SmartPointer.SetNull(
    In_The_Container.MapA(In_The_Container.FirstFree).Ptr);
end Con_Delete;
```

### 3.4 Use of the SBF

We illustrate here how to use a SBF-based Ada Container Library by presenting two generic algorithms over it: `Min_In_Range` and `Sort`. The `Min_In_Range` algorithm returns the minimum element found between two iterators in a given order. The `Sort` algorithm sorts the elements of a container (with respect to the iteration order). Fig. 3 shows an example of the last one in which can be observed that after sorting the container all the shortcuts refer to the right object. We remark that these generic algorithms work for any kind of container with the same time efficiency (because iterators are independent of the concrete container).

#### The `Min_In_Range` generic algorithm

```ada
generic
  type Item is private;
  with function '<' (L,R:Item) return boolean is '<';
  with package BI is new Bidirectional_Iterators(Item);
  use BI;

  function Min_In_Range(FirstIt,LastIt:
    BI.Bidirectional_Iterator'Class) return BI.Bidirectional_Iterator'Class is
    ItMin, It: BI.Bidirectional_Iterator'Class:= FirstIt;
    begin
      while not IsDone(It) and It /= LastIt loop
        if CurrentItem(It) < CurrentItem(ItMin) then
          ItMin := It;
        end if;
        Next(It);
      end loop;
      return ItMin;
    end Min_In_Range;
```

#### The `Sort` generic algorithm

```ada
generic
  type Item is private;
  with function '<' (L,R:Item) return boolean is '<';
  with package C is new Containers(Item);
  use C;

  procedure Sort (C1 : in out C.Container'Class) is
    function Min_In_Range is new
      GenericAlgorithms.Min_In_Range(Item => Item,
        '<' => '<', BI => C.Container_Iterators);
    It1, It2, ItMin: C.Iterator;
    begin
      if Cardinality(C1) /= 0 then
        Bind_To_Container(It1,C1);
        Bind_To_Container(It2,C1);
        Last(It2);
        Next(It2);
        while not IsDone(It1) loop
          ItMin:=Iterator(Min_In_Range(It1,It2));
          Swap(It1,ItMin);
        end loop;
      end if;
      end Sort;
```

4. EVALUATING THE SHORTCUT-BASED FRAMEWORK

In this section we carry out the assessment of the SBF introduced in the last section. We proceed as follows. First we define some metrics for reasoning about the quality of a SBF-based Ada container library, using the ISO/IEC-9126 quality model for container libraries presented in Sect. 2 and then we reason about how the SBF affects the value of these metrics. Remarkably, this study is carried out without making any assumption about the library other than it is built on top of our framework.

There are many proposals for defining metrics. We have adopted one of the most widespread approaches, the Goal-Question-Metric (GQM) [3]. In GQM, goals of the product under measurement are identified, and then some questions are raised to characterize the way the assessment of a specific goal is going to be performed. Last, a set of metrics is associated with every question in order to answer it in a quantitative way; metrics can be objective or subjective. The final result of the GQM approach is a hierarchical structure in graph-like form, since metrics may influence in more than one question, and questions may be related to more than one goal. Goals are composed of four elements: purpose, issue, object and viewpoint. In our framework, these elements take the following form:

- **Purpose.** Presence or absence of a particular feature or characteristic in the library.
- **Issue.** The GQM recommends to identify quality goals; then, we define one issue for each attribute of the ISO/IEC-based quality model. As a consequence, we have as many goals as quality attributes.
- **Object.** Always the SBF-based Ada Standard Container Library.
The SBF allows implementations with rearrangement of elements in its data structure after modifications (e.g., open addressing hashing).

We characterize content quality by robustness, basically presence of unnecessary or conflicting elements in the library. Again, our proposal does not interfere with the robustness of the library: shortcuts do not improve or damage the robustness of the core suitability of the library.

### 4.2 Assessment of other Functionality Attributes

Table 5 shows an excerpt of the GQM model for other functionality attributes. Some explanations follow:

- Access by position is characterized by the set of operations that use shortcuts to access the elements in the container: lookup, removal and modification. We measure the percentage of containers that provide each of these operations. The SBF provides 100% coverage.

- For iterators, the questions elucidate which types exist (see for instance [18] for a summary of types of iterators). We show as example the questions for two of those types. We use the same metric as before with the same 100% result.

- Concerning algorithmic variety, we distinguish among the algorithms provided by the library and the possibility of defining new ones\(^3\). In both cases we focus on algorithms using positions or iterators. The SBF is well-suited for both categories, because it does not restrict the type of algorithms that can be designed (see Sect. 3.4 for examples). Remarkably, generic algorithms that traverse the container using iterators are allowed to modify the container during iteration.

\(^2\)It is not a goal of the paper to give detailed arguments supporting this particular model; in fact, this would require to present a whole metrics plan as defined in [19].

\(^3\)This last question is also relevant for changeability; we remark that the GQM approach allows questions to be bound to more than one goal.
Table 5: GQM Model for other Functionality Attributes

<table>
<thead>
<tr>
<th>Goal</th>
<th>Question</th>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose: Have</td>
<td>Does the library provide an operation for retrieving by position?</td>
<td>% of containers that provide it</td>
<td>100 %</td>
</tr>
<tr>
<td>Issue: Access by position</td>
<td>Does the library provide an operation for removing by position?</td>
<td>% of containers that provide it</td>
<td>100 %</td>
</tr>
<tr>
<td>Viewpoint: Ada community</td>
<td>Does the library provide an operation for modifying by position?</td>
<td>% of containers that provide it</td>
<td>100 %</td>
</tr>
<tr>
<td>Purpose: Have</td>
<td>Does the library allow removing during iteration?</td>
<td>% of containers that provide it</td>
<td>100 %</td>
</tr>
<tr>
<td>Issue: Access by Iterators</td>
<td>Does the library allow modifying during iteration?</td>
<td>% of containers that provide it</td>
<td>100 %</td>
</tr>
<tr>
<td>Viewpoint: Ada community</td>
<td>Has the library the most frequently used algorithms?</td>
<td>% of basic generic algorithms</td>
<td>N.A.</td>
</tr>
<tr>
<td>Purpose: Have</td>
<td>Does the library allow defining new generic algorithms that work with all containers?</td>
<td>% of containers for which is possible</td>
<td>100 %</td>
</tr>
<tr>
<td>Issue: Algorithmic variety</td>
<td>Is it impossible to access to an item other than the original one?</td>
<td>100% − % of cases that is possible</td>
<td>100 %</td>
</tr>
<tr>
<td>Viewpoint: Ada community</td>
<td>Is it possible to use implementation strategies that make rearrangements without making positions out-of-date?</td>
<td>% of cases that is possible</td>
<td>100 %</td>
</tr>
<tr>
<td>Purpose: Have</td>
<td>Are positions remain valid when insertions or deletions take place?</td>
<td>% of cases that remains valid</td>
<td>100 %</td>
</tr>
<tr>
<td>Issue: Accurate Access by position</td>
<td>Is it impossible to traverse the data structure beyond the last element?</td>
<td>100% − % of cases that is possible</td>
<td>100 %</td>
</tr>
<tr>
<td>Viewpoint: Ada community</td>
<td>Does the iterator remain valid when insertions or deletions take place?</td>
<td>% of cases that remains valid</td>
<td>90 %</td>
</tr>
<tr>
<td>Purpose: Have</td>
<td>Is it impossible to use a position to access to a wrong container?</td>
<td>100% − % of cases that is possible</td>
<td>100 %</td>
</tr>
<tr>
<td>Issue: Secure Access by position</td>
<td>Is an exception raised if a position which does not have associated an object is used?</td>
<td>% of cases that is raised</td>
<td>100 %</td>
</tr>
<tr>
<td>Viewpoint: Ada community</td>
<td>Is it possible to know if a position is bound to an element in a container?</td>
<td>% of cases that it is possible</td>
<td>100 %</td>
</tr>
<tr>
<td>Purpose: Have</td>
<td>Is it possible to know if an iterator is still valid or not?</td>
<td>% of cases that is possible</td>
<td>100 %</td>
</tr>
<tr>
<td>Issue: Secure Access by iterator</td>
<td>Is it impossible to use operations not valid given the type of the iterator?</td>
<td>100% − % of cases that remains valid</td>
<td>100 %</td>
</tr>
<tr>
<td>Viewpoint: Ada community</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N.A.: Not Affected

- In accuracy and security goals, we have included questions for identifying those situations that can yield to erroneous behaviour. We include some examples in the table addressing to scenarios that typically fail to behave accurately in other proposals. For instance, the SBF ensures accuracy of results even for those strategies that rearrange the elements in the data structure. Metrics just count the percentage of failure cases. We remark that the SBF suffers just a restriction: removal of the current element during an iterator makes the iterator undefined.

4.3 Assessment of Understandability and Changeability

Briefly, we summarize in this section the points that show how the SBF supports these attributes. First, the concept of shortcut is in fact a kind of container-oriented design pattern; in fact, we have formulated the SBF in terms of design patterns [16] using the notation of [10]. The use of design patterns has been mentioned in Sect. 2 as a technique supporting quality of design, and also enhances quality of documentation and uniformity, since all the concepts are applied the same way to all kinds of containers.

The internal structure of the SBF exhibits also other properties that have been identified as quality attributes. First, access by position and iterators rely on a clear distinction among specification and implementation of containers; furthermore, shortcuts and iterators follow this distinction, i.e. they are formulated in terms of the specification and do not impose any restriction on the implementations of the containers. Second, complexity is kept up to a minimum level: for instance, the set of operations to manage shortcuts and iterators is small and always the same, shortcuts are obtained as a result of the usual insertion operation without any further action, etc. Next, other design properties such as modularity and loose coupling are fulfilled as a consequence of our proposal. Last, internal reusability is favoured due to the organization of the class hierarchy in the SBF; this property makes library customization and extension easier.
4.4 Assessment of Efficiency

The assessment of the SBF concerning efficiency turns out to be a crucial point. It is clear that there must be a price to pay for having access by position and iterators fulfilling suitability, accuracy, security, understandability and changeability as explained above, and this price is efficiency. Nevertheless, we have checked that efficiency is not seriously damaged by the SBF. We summarize our study in terms of the efficiency attributes identified in Sect. 2.2.

4.4.1 Assessment of Time Behavior

Order of magnitude. It is worth to remark two fundamental things. First, all the operations that involve shortcuts and iterators are as efficient as they can be, which means that access by shortcut is \( O(1) \), and traversals using iterators are \( O(n) \), being \( n \) the number of elements in the container. Second, the order of magnitude of the core operations of the containers (i.e., those using neither shortcuts nor iterators) remains the same; in other words, shortcut management can be done without penalising the order of magnitude of the operations.

Real time efficiency. We have carried out some experiments for assessing this quality attribute. We have developed a SBF-based version of the Booch Components library (see [14] for details) and then we have compared some execution results on a large set of instances with the original version of the library. As an example, we have measured the results for three representative processes (insertion, lookup and iteration) in a particular type of container, namely a Bag implemented with a hashing table. In order to cover a reasonable sample of representative scenarios we have considered the following three criteria: container’s element size, percentage of occupation of the Bag and number of collisions by bucket produced by the hash function. For each scenario, different bag sizes and different number of insertions (depending of the number of occurrences of each item) have been tested.

From the results obtained, we have observed that the execution time of iteration and the execution time of lookup (with respect to the number of items) are independent of the scenario in the case of the shortcut version, while in the case of the original version the execution time increases proportionally to the size of the items and in some cases (instances are generated randomly) inversely proportional to the percentage of occupation of the Bag and number of collisions by bucket produced by the hash function. For each scenario, different bag sizes and different number of insertions (depending of the number of occurrences of each item) have been tested.

Figure 4 shows the number of iterations required to amortise the insertion overhead in the best case (B.C. curves) and the worst case (W.C. curves) for some representative scenarios. The overhead of insertions is amortised with 2 and 8 iterations respectively. In general, time efficiency of insertion with shortcut gets closer to the one of the original version as size of elements increases; we remark that with strings of 300 characters, there are cases where one single iteration makes our proposal better than the original one.

All the test programs and the Ada-95 packages needed to generate these programs together with the shortcut version can be found at wwwlsi.upc.es/~jmarco/testing.html and the original library at www.pogner.demon.co.uk/components/bc.

4.4.2 Assessment of Resource Utilization

Data structure real space efficiency. Being \( N \) the number of elements in the container and assuming a pointer-based implementation of shortcuts without garbage collection, which is the worst case concerning space efficiency, the total amount of extra space required for shortcut management is:

\[
2 \cdot N \cdot \text{space}(\text{pointer}) + N \cdot \text{space}(\text{shortcut}) + N \cdot \text{space}(\text{bool}) + N \cdot \text{space}(\text{integer})
\]

The first operand comes from the double linked list\(^5\), the second one from the shortcuts stored in the implementation of a concrete abstraction and the last two operands from the deleted flag and the reference counter. We would like to remark that this waste of space will usually generate a later saving, when shortcuts substitute identifiers (generally strings, which require more space than shortcuts) in references from other containers. Identifiers would be required if the Ada Standard Container Library would not provide access by position, or if this access by position were inaccurate or unsecure and so rejected in critical systems. The relationship between these two factors may be formally established.

Let \( N \) be the total number of objects in the container and \( R \) the total number of external references to objects in the container. Since generally,

\[
\text{space}(\text{identifier}) \geq \text{space}(\text{pointer})
\]

then \( \exists k \geq 1 \) s.t. \( (k+1) \cdot \text{space}(\text{pointer}) > \text{space}(\text{identifier}) \geq k \cdot \text{space}(\text{pointer}) \) and since \( \text{space}(\text{shortcut}) = 2 \cdot \text{space}(\text{pointer}) \) (because we use two pointers for assuring that a particular shortcut is bound to a particular container) and assuming the worst case \( \text{space}(\text{bool}) = \text{space}(\text{integer}) = \text{space}(\text{pointer}) \), space is really saved when the relationship

\[
R \cdot \text{space}(\text{identifier}) \geq (6 \cdot N + 2 \cdot R) \cdot \text{space}(\text{pointer})
\]

\(^5\)The space of a smart pointer is the same as a regular one.
holds, which is satisfied when the following condition holds:

\[ R \cdot k \geq 6 \cdot N + 2 \cdot R \]  \hspace{1cm} (3)

For example, if identifiers were strings of average size 40 (i.e., \( k = 10 \) with pointers requiring 4 bytes) and there is an external reference to each of them \( (N = R) \), then by substituting from (3) we obtain:

\[ 10 \cdot R \geq 8 \cdot R \]

that obviously holds.

**Order of magnitude of the data structure.** From the result above, we can verify that the addition of shortcut management does not increase the order of magnitude of the data structure used in the container implementation.

**Order of magnitude of the auxiliary space in operations.** Any operation, either related to core or general suitability, suffers from more than an \( O(1) \) increment of auxiliary space.

## 5. CONCLUSIONS

In this paper we have presented the Shortcut-Based Framework (SBF) as the basis for designing and implementing a prospective Ada Standard Container Library. The SBF aims at supporting access by position and iterators suitable for containers in the library. The proposal is built around the concept of shortcut. We have arranged a class hierarchy that includes base classes for containers, shortcuts and iterators, and specializations for types of containers and implementations. We have assessed the quality of the SBF-based container library using an ISO/IEC-based quality model, together with some metrics defined with the Goal-Question-Metric approach.

We think that the main contributions of our work are:

- The SBF provides high-quality access by position and iterators. Unlike other approaches for handling this kind of access, the SBF provides rich suitability and ensures accuracy and security of their use. Also, our approach provides uniformity and supports understand-ability and changeability of the library, among other advantages. We have quantified the cost of the approach regarding efficiency, which turns out to be not severe; in fact, both time and space overhead may disappear under some circumstances.

- The SBF provides absolute freedom for the core suit-ability of the library, i.e., types of containers, implementa-tion strategies and operations offered. For this reason, our approach does not interfere with the discussion about whether an existing Ada library (Charles, Booch Components, etc.) can be used as the basis for the standard library, or else if this library shall be de-sign and implemented from the scratch.

- Our proposal has been assessed thoroughly using an exhaustive quality model and a set of metrics. It should remain clear that in this paper we have not presented neither the whole quality model nor the whole set of metrics, for the sake of brevity; we have focused on those parts of the model and those metrics that are more relevant for our proposal.

To be more precise, the SBF avoids the following problems, which are very usual in other proposals for dealing with access by position and iterators:

- Out-of-date positions or iterators. Shortcuts are de-coupled from physical positions (either memory ad-dresses or array indexes).

- Restrictions on the use of iterators. Remarkably it is allowed to update the container during a traversal.

- Restrictions on the implementation strategy. Remark-ably, implementations in which the elements may change their position are allowed. Also, the SBF may be tai-lored to any kind of memory management scheme.

- Lack of uniformity. All the containers offer the same operations for iterating and accessing by position.

- Lack of internal reuse. The cost of extending the li-brary with new types of containers or new implementa-tion strategies is diminished due to the reuse of code for shortcut management.

## 6. REFERENCES


[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1996.


