Shortcuts for the ADA Standard Container Library

Jordi Marco and Xavier Franch
Dept. Llenguatges i Sistemes Informàtics (UPC)
c/ Jordi Girona 1-3. E-08034 Barcelona, (Spain)
jmarco@lsi.upc.es,franch@lsi.upc.es

1. MISSION STATEMENT

To provide a generic and versatile feature aimed at providing efficient and secure access to elements stored in any container, while keeping other quality criteria of the library remarkably reusability and extendibility.

Our proposal has been applied to the Booch Component Library improving it from many points of view. This work was presented at the Ada-Europe 2000 Conference [1] and it is referenced, as a contribution, at the Booch Component page: `http://www.pogner.demon.co.uk/components/bc/contrib/`. The Ada 95 code of the resulting library can be found at `ftp://ftp.lsi.upc.es/pub/users/jmarco/`.

2. INTRODUCTION

In this position paper our main objective is not to propose a concrete collection of abstract data types 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 [2]. This framework allows solving the majority of drawbacks that are present in the most widespread container libraries (see [2, Sect. 2]). This is achieved by means of the *Shortcut* concept that we propose at the core of our framework. As a result, container libraries developed using our framework not only improve reusability but also other quality criteria including efficiency (see Sect. 5).

3. 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 are $O(1)$) 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 unreliable proposals (e.g., iterator in STL, references in JCF, item in LEDA, 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.

4. THE SHORTCUT-BASED FRAMEWORK

In this section we outline the main features of the *Shorcut-based Framework* (SBF), implementation details can be found in [2]. The key point consists on storing the objets 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. We first define the Shortcut-based Framework hierarchy for container libraries. This hierarchy has been built borrowing the main ideas from the different hierarchies of some widespread container libraries [2, Sect. 2]. In fact, 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), but its general layout. A complete hierarchy of a container library must 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 Shortcut-based Framework.

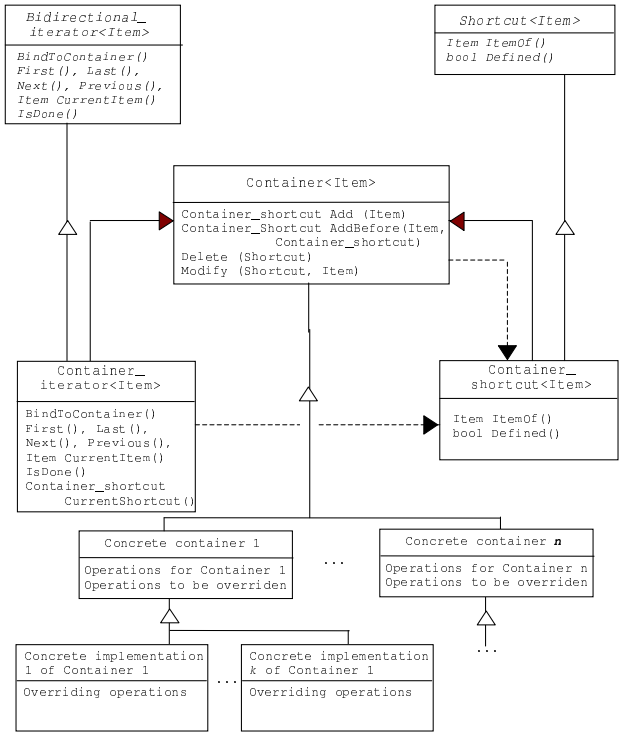

Figure 1: The Shortcut-based Framework

The classes involved in this hierarchy are:

- *Bidirectional_iterator*. An abstract class that provides the interface of this kind of iterator, i.e. iterators that support forward and backward traversal of a container.
- Container_iterator. An efficient implementation of Bidi-

rectional_iterator enlarged 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 must be $O(1)$.

- *Shortcut.* Defines the interface of the new design pattern that introduces the concept of shortcut as the location or position of objects.
- Container_shortcut. An efficient (i.e., $O(1)$ time) and safe implementation of the Shortcut interface. Container_shortcut 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 deferred operations that appear as a result of applying the Template Method design pattern (that we call *concrete interface*) and implement a (in some cases non-efficient) version of them using the Container interface and shortcuts. We want to remark that this implementation strategy uses the base class Container as a black box and, at the same time, makes the concrete container 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 fulfil efficiency requirements.

5. SBF BENEFITS

- Step-by-step implementation. Each SBF class implements all the common capabilities that the class offers (those ones identified in the current step), so it is an implementation class. The full implementation of each class is carried out using its parent class (implemented in previous steps) as a black box (i.e., using only its interface) and without making any assumption about the implementation of the classes appearing in the next step.
- Implementation with reuse. The most common capabilities (remarkably iterators and shortcuts) are implemented only once in the most efficient way (all the operations of shortcuts and iterators are $O(1)$).
- Implementation for reuse: generality. The generality of a Shortcut-based container library depends on the number of concrete containers and concrete implementations it offers. As far as the implementation of the hierarchy does not make any assumption about the form or behaviour of its components, the library may contain an utterly rich variety of containers and implementations. Last, a lot of generic algorithms working over the container base interface or over the container iterators can be offered.
- Implementation for reuse: extendibility. As many concrete containers as needed may be added; in addition, there is no restriction on the number and variety of container implementations that can appear, and new implementations may be added to existing concrete containers as required. These extensions are easy to carry out because they can reuse the most common capabilities (iterators and shortcuts); furthermore, as a consequence, the implementation of these classes is not restricted for the efficiency requirements of these capabilities.
- Functionality. In addition to offering the common features of the container libraries, secure updating of the container during iteration is supported.
- Integrity. Iterators not become out-of-date when a new object is inserted to, or an existing one is removed from, the container. The SBF offer operations to know if an iterator is still valid or not and if a shortcut is bound to an item in the container or not.
- Time efficiency. Shortcuts allow highly efficient access to elements without interfering in a significant manner with the efficiency of the other operations. Moreover, iterators are highly efficient (all of the iterator operations are $O(1)$ in the worst case). Complete computational results corresponding to compare the time efficiency of some operations of the original version of the Booch Component library and of the shortcut version of this library can be found at `http://www.lsi.upc.es/~jmarco/testing.html`.
- It is possible to design generic algorithms using the iterators and shortcuts provided by the container base class (like those in the C++ STL). These algorithms could be applied equally well to any data structure without any performance difference between them.

6. REFERENCES

[1] J. Marco and X. Franch. Reengineering the Booch Component Library In *Reliable Software Technologies Ada-Europe 2000*, volume 1845 of *Lecture Notes in Computer Science*, pages 96-111. Springer-Verlag, 2000.

[2] J. Marco and X. Franch. Improving Design and Implementation of OO Container-like Components Libraries. Technical Report, Departament de Llenguatges i Sistemes Informàtics. Universitat Politècnica de Catalunya, 2001. `http://www.lsi.upc.es/~jmarco/TR125.pdf`.