Bridging the Gap Between Design and Implementation of Component Libraries

Jordi Marco and Xavier Franch

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

Abstract. Object–oriented design is usually driven by three main reusability principles: step–by–step design, design for reuse and design with reuse. However, these principles are just partially applied to the subsequent object–oriented implementation, often due to efficiency constraints, yielding to a gap between design and implementation. In this paper we provide a solution for bridging this gap for a concrete framework, the one of designing and implementing container–like component libraries, such as STL, Booch Components, etc. Our approach is based on a new design pattern together with its corresponding implementation. The proposal enhances the same principles that drive the design process: step–by–step implementation (adding just what is needed in every step), implementation with reuse (component implementations are reused while library implementation progresses and component hierarchies grow) and implementation for reuse (intermediate component implementations can be reused in many different points of the hierarchy). We use our approach in two different manners: for building a brand–new container–like component library, and for reengineering an existing one, Booch Components in Ada95.

1 Motivation

Since McIlroy proposed in 1969 the notion of software catalogue [McI69], component–based software development (CBSD) has become without any doubt one of the most important software development paradigms. The key point behind this paradigm is the process of reusing components from standard software catalogues. Component reuse provides many advantages, remarkably software production hastening, software quality improvement and software maintenance cost decrease.

One of the most valuable contributions in CBSD is object–oriented (OO) technology. Basic concepts such as inheritance, polymorphism and dynamic binding, and others built on top of them, such as design patterns [GHJ+96], had a strong impact on this paradigm. In fact, as Meyer remarks [Mey97], OO technology makes possible, for the first time, the idea of turning academic McIlroy's vision of software development into a component–based industry become real.

Unfortunately, despite of this ongoing success, we have not still reached the ultimate McIlroy's goal: making software components equivalent to other engineering component types, as chips, bricks and valves. One of the reasons behind this fact has to be with the difficulty of turning designs into implementations. The OO design process is usually driven by three main reusability principles, namely step–by–step design, design for reuse and design with reuse. However, these principles are often just partially applied to the subsequent OO implementation. Therefore, a gap between design and implementation appears with respect to these reusability principles. This fact holds even if advanced OO concepts such as design patterns are used during development: it is not enough to reuse design patterns, a good reuse policy must provide implementations of them.

But we turn back to McIlroy's software catalogues, more specifically to the OO corresponding notion of class component library. Class component libraries help to dismiss the gap mentioned above because classes implemented in the library come to light naturally when identifying classes in the new design. Hence, a good design and implementation of these library classes become crucial to support the reusability of the components stored therein.

But design and implementation of reusable, general–purpose class libraries is a hard work, because many criteria have to be taken into account:

– Adaptability. Design and implementation of general–purpose libraries have to be flexible and extensible to allow the possibility to adapt them or to extend them to a particular context.
– Reusability. General–purpose libraries should deal with as many different contexts as possible. Therefore it is utterly important to avoid assumptions about (and dependencies on) the context, which could limit the usability of the library and hence its applicability and effectiveness.
– Functionality. The success of general–purpose libraries depends on its ability to provide the most common functionalities in the domain in which they are defined.
– Efficiency. Classes in the library should incorporate algorithms and features to fulfill usual efficiency requirements.

Reconciling these criteria is a difficult task. At the design level, it is possible to define a class hierarchy satisfying the first three criteria. But when building its implementation, efficiency requirements collide often with the others, making design decisions difficult to be kept. This makes the gap between design and implementation to appear again, now in the development of the library itself. Thus, reusability can be damaged again, both when building the library (common functionalities are implemented more than once in different classes) and when using it (because library adaptability, reusability or functionality may have been partially sacrificed).

The gap may take different forms in different types of libraries. In this paper we focus on a concrete category of libraries, the one of container–like component libraries (*CLC–library* for short). Containers are objects that store collections of other objects; different types of containers may offer different functionalities, and their implementations (usually, more than one) will satisfy different efficiency requirements. Some representative CLC–libraries are the Standard Template Library (STL) [MS96], the Library of Efficient Data types and Algorithms (LEDA) [MN99] and Booch Components (BC) [BV90,BWW99].

In these kind of libraries a well–established hierarchy of components can be found at the design level. Sometimes, this hierarchy appears explicitly (for instance BC) and sometimes not (for instance, STL and LEDA). In any case, the hierarchy has no subsequent implementation in these libraries, making thus the gap appear again. Typically, three different situations exist:

– The implementation of the intermediate levels of hierarchy only provides interfaces, but not real implementations. In other words, the hierarchy implementation just preserves the layout. Code reuse does not takes place, but efficiency is optimal, because methods can use the private attributes of class implementation.
– The intermediate levels simply disappear, maybe because it can be argued that non–implemented interfaces are useless from an implementation point of view. Again, efficiency is the main motivation behind this approach. Examples of these libraries are STL and LEDA.
– The hierarchy is partially preserved in the implementation, with those modifications required to support code reuse. This solution usually dismisses the level of adaptability and reusability of the library. The Ada95 version of the BC is a representative library in this scenario.

The purpose of this work is bridging this gap between the design and the implementation of CLC–libraries. Our approach is based in the definition of a new design pattern, namely *Shortcut*, together with its corresponding implementation. This pattern describes and solves the problem of accessing objects stored in a container, in an abstract and efficient way. Most of the existing CLC–libraries try to solve this problem by means of different *ad–hoc*, implementation–dependent proposals. Instead, we use an implementation–free approach based on the use of shortcuts to implement a generic container in which the objects are really stored. Our approach yields to the same reusability principles in the implementation process that we had in design: step–by–step implementation (coding just what has been introduced or redefined in every class), implementation with reuse (class–implementations are reused while the implementation stage progresses and class–hierarchies grow) and implementation for reuse (intermediate class implementations can be reused in many different points of the hierarchy).

Our proposal is twofold in the sense that it can be applied both for developing new CLC–libraries and for improving existing ones, because the core data structures and algorithms can be reused. In the case of reengineering an existing library, since the addition of shortcuts will not affect the former behaviour of the library, those running software applications that use the previous version of the library does not need to be modified. We will show this point in a particular case, the Ada95 BC library.

The rest of the paper is organized as follows. First, we make a comparative analysis of the three CLC–libraries mentioned above highlighting their benefits and drawbacks. Next, we propose a hierarchy and a design pattern aimed at solving these drawbacks. Last, we apply the design pattern to a particular case study, the Ada95 BC library.

2 Some Representative Implementations for CLCL

In this section we analyse three of the most widely used CLC–libraries: the Standard Template Library (STL), the Booch Components (BC) and the Library of Efficient Data types and Algorithms (LEDA). For each of them we focus only on the containers subhierarchy (together with their capabilities, e.g. iterators).

2.1 The Standard Template Library

STL is a component library adopted by ANSI as a standard for C++. This library has been an important contribution to the programming methodology. STL has not an explicit hierarchy, only the leaf classes exist, but there is an implicit hierarchy that can be deduced from its organisation.

STL is organised in six kinds of components: containers, generic algorithms, iterators, function objects, adaptors and allocators. We are going to analyse containers, iterators and containers adaptors components.

Containers are divided into two categories: sequence containers and sorted associative containers.

There are three different sequence containers abstractions: vectors, deques and lists with a single one implementation. Using the containers adaptors: stack, queue, and priority queue together with the sequence containers can be obtained three different implementations for each one of these adaptors. Therefore, we can say that STL has six different sequence containers abstractions: vectors, deques, lists, stacks, queues, and priority queues, with one implementation of the first three ones and three different implementations of the three last ones.

The associative containers category offers four different containers abstractions: sets, multisets, maps and multimaps each of them with just one implementation.
All the containers provide:

– Iterators. Iterators are objects that allow to navigate through all the objects stored in a container. Iterators are divided in five categories: input iterators, ouput iterators, forward iterators, bidirectional iterators and random acces iterators among which there is the hierarchical relationship shown in Fig. 1. Only vectors and deques provide random access iterators; the other containers provide just bidirectional ones. In STL, iterators mix two different capabilities, namely, the concept of iterator and the concept of location of the objects stored in the container. This mixture of concerns provokes some drawbacks that we mention further on the paper.

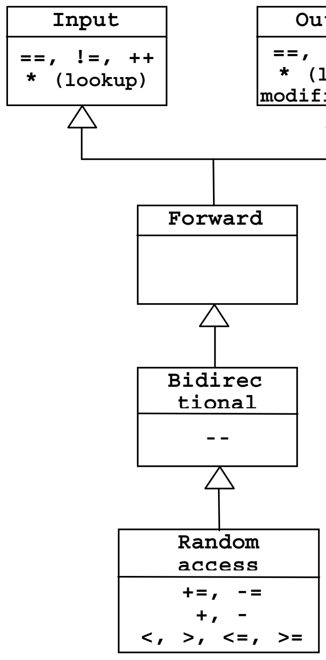


Fig. 1. Hierarchy of the STL iterators

- Insert operations. There are a variety of insert operations to insert one or more objects. The common one is the basic insert operation which consists in inserting one object; this operation returns an iterator that can be used like a reference to the new object.
- Remove operations. All the remove operations use iterators to remove the object bound to them. The common one removes one object using its iterator.
- Modify operations. In STL, the modification of an object stored in a container has to be made using iterators like pointers. The problem with this scheme is that iterators are not persistent, in other words an iterator could refer not to the original object but to other one (typically when objects can be reallocated inside the structure or when they are removed); the problem is serious because STL does not provide any functionality to check this situation.
- Efficient access operations. In STL, efficient access to objects stored in a container is made by using iterators like pointers. Therefore, the problem mentioned above appears again.

This library have a lot of advantages, mainly:

- Offers a large amount of robust, efficient and well–designed components with appropriate algorithms and data structures.

– The algorithms are structured independent, they work externally using the iterator common facility.
– For many efficient data structures access by position is important. STL uses iterator concept to cast positions into an abstract form.
– It is a standard library supported by documentation and books.
– It is freeware.

The main drawbacks that present this library are:

– Lack of internal level of reusability. All the common capabilities have to be implemented on each concrete class.
– Each container abstraction (besides the abstractions obtained using adaptors: queue, stack and priority queue) has just a single implementation. The implementation of *List* is a double linked list, the associative sorted containers are implemented using a red–black tree, etc. This makes difficult the possibility of adapting the library for concrete proposals (e.g. if a hashing map is needed to have even more efficient access by key).
– It is difficult to extend it. If we want to add a container component, it has to offer all the common capabilities but some of them restrict the possible implementations (e.g. we cannot add a new container whose objects change their position in the internal structure when a new object is inserted) and others difficult to understand and implementing.
– Non persistency of iterators. In some containers classes all the iterators are invalidated when we remove or insert an object. And in the others an iterator could be invalid if the object that it referred has been removed.
– Lack of an operation to know if an iterator is still valid or not.

2.2 The Booch Components

The Booch Components was first created for Ada 83 [Boo87] and reengineered first for C++ [BV90] and later on for Ada 95 [BWW99]. We are going to analise in detail the Ada 95 version, which is later used in the paper to show the feasibility of our approach.

This version of the Booch components is organised into three main super–classes: *Containers*, *Support* and *Graphs*, which have a common parent–class *BC*. The base class *BC* has no functionality at all, it only provides the definition of the common exceptions. The *Containers* category of classes provides a wide range of structural abstractions (lists, bags, sets, collections, etc.) using many widespread implementation techniques (chaining, hashing, search trees and so on). Figure 2 shows the main hierarchy of these components; their code is available at [BWW99].

The *Containers* class offers only the interface of the iterators. The structural abstraction classes offers the interface of these abstractions and, in an intent of reuse implementation, the implementation of the iterators, we can see later that this intent provokes several drawbacks. Finally, in the leaf classes of the hierarchy we can find the implementation of the concrete container. The unique common operation of these containers is the add operation.
This library offers several advantages that make it very useful, mainly:

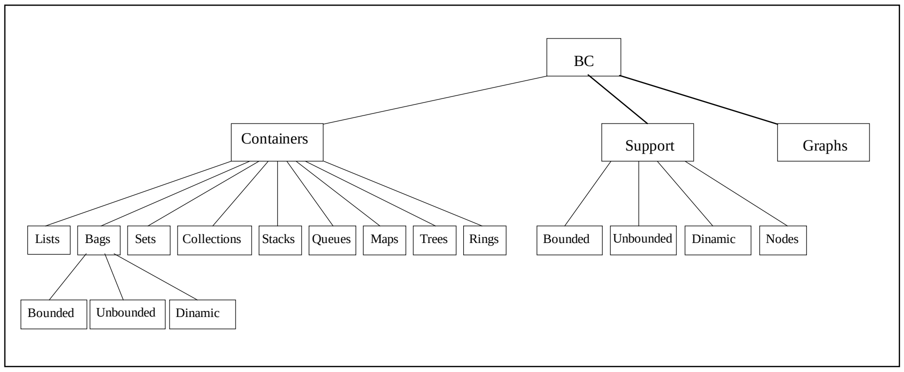


Fig. 2. Hierarchy of the Booch component classes (excerpt)

- A large amount of robust and well–designed components with appropriate algorithms and data structures.
- It is a well–known library supported by documentation and books.
- It is freeware.
- It has several and complete testing packages for its components.

The container category of this libray presents some drawbacks that decrease the potential of reusability of this component library. Most of the problems arise because some parent–classes depend on the concrete implementation of their children–classes. To make it clear, Fig. 3 shows a typical situation in this library, in which the *BC.Containers.Bags* class depends on the concrete form of its children classes *Bounded*, *Unbounded* and *Dynamic*, which are hashing tables. Notice that the type definition of *Bag_Iterator* forces all the *Bags* children to be implemented by means of a hashing table. This restriction interferes with the possibility of extending the class hierarchy or changing the concrete form of one of its children. All these dependencies exist because iterators are strongly dependent on the concrete container implementation. Similar dependencies can be also found between the classes *Maps*, *Sets*, *Queues*, etc., and their respective children. Therefore, to solve the problems we clearly need to make iterators independent of the specific container.

To sum up, the main problems in this library are:

- The hierarchy is not robust enough with respect to changes in some of their components: changes in a component require the modification of other components. This is due to the implementation dependency mentioned above. For instance, changing the current hashing implementation of the *BC.Containers.Bags.Bounded* by an implementation (bounded) with a binary search tree (for instance, because elements must be obtained in some order), requires changing the implementation of the type *Bag_Iterator* (defined in the *BC.Containers.Bags* class) and the implementation of its operations as well.

```
generic
package BC.Containers.Bags is
...
private

type Bag is abstract new Container with null record;
...
type Bag_Iterator (B : access Bag'Class)
is new Actual_Iterator (B) with record
  Bucket_Index : Natural := 0;
  Index : Natural := 0;
end record;
...
end BC.Containers.Bag;
```

Fig. 3. Extract of the generic package *BC.Containers.Bags*

- Moreover, this hierarchy restricts to a single set of possible implementations for some of the different structure abstractions. This is a serious drawback because some of the implementations provided therein can be inefficient in some contexts (we have already mentioned ordered traversal of *Bags*). For instance, it is not possible to have different implementations of the class *BC.Containers.Bags.Bounded*, because it is not implementation–independent, and hence it forces a concrete implementation strategy (hashing). This problem could be solved adding another level in the hierarchy, making the class abstract and defining their concrete children. This is not possible without changing other parts of the hierarchy, because of the implementation dependency again.
- Low level of abstraction makes the usage of the implementation harder. This happens when dealing with iterators. The iterator type and its operations are strongly dependent on the concrete implementation of the underlying structure. As a consequence, for every concrete implementation of a children–class a new *Actual_Iterator* type must be defined and its operations must be overridden. This approach, which is different from many other libraries, prevents the easy usage of the iterator facility.
- It is not only the lack of multiple implementations for components that damages efficiency, but also some of the iterator operations have lineal cost in the worst case with respect to a certain parameter (although their amortised cost is constant). For instance, as shown in Fig. 4, the *Reset* operation could have linear cost with respect to the *Number_of_Buckets*. A similar problem occurs in the *Next* operation.
- In many contexts in which components often encapsulate data structures, reusability can be damaged due to efficiency requirements: even if a component fulfils a required functionality, the time complexity of its operations may be inadequate given the context in which it should be integrated (either considering them individually or when combining them to build more complex components). The access by means of the operations offered by a

```
procedure Reset (It : in out Bag_Iterator) is
  begin
    It.Index := 0;
    if Cardinality (It.B.all) = 0 then
      It.Bucket_Index := 0;
    else
      It.Bucket_Index := 1;
      while It.Bucket_Index <= Number_Of_Buckets (It.B.all) loop
        if Length (It.B.all, It.Bucket_Index) > 0 then
          It.Index := 1;
          exit;
        end if;
        It.Bucket_Index := It.Bucket_Index + 1;
      end loop;
    end if;
end Reset;
```

Fig. 4. Reset operation's code of the generic package *BC.Containers.Bags*

component may be costly if the logical layout of the data structure is used; if fast access is required, it becomes necessary to look up the item using directly a reference to it.

2.3 The Library of Efficient Data types and Algorithms

The Library of Efficient Data types and Algorithms (LEDA) is the result of a project started in 1988. The goal of this project was design a library of data types and algorithms, mainly, for combinatorial computing area where it is necessary the use of complex data types. The actual version of this library [MN99] offers a large amount of data types and algorithms. From a container point of view LEDA offers the next generic (parameterised) data types: arrays with one and two dimensions, stacks, queues, bounded stacks, bounded queues, linear lists, single linked lists, sets, dictionaries, dictionaries with implementation parameter, sorted sequences, sorted sequences with implementation parameter, dictionary arrays, dictionary arrays with implementation parameter, hashing arrays and others variations above containers.
All the containers–like data types of this library provide:

– Items. LEDA use an item concept that play the role of locations or positions in data structures to achieve efficient access to the objects therein.
– Iteration macros. This macros can be used similarly to the C++ **for** statement to iterate over the items of a container. Iteration macros have the restriction that the object corresponding to the item can not be modified inside the body of the loop.
– Insert operations. These operations return an item that can be used in other operations which have then constant time.

– Delete operations. These operations take an item as a parameter to delete the object it refers. The item has to be in the container.
– Modify operations. These operations take an item as a parameter to modify the object it refers. The item has to be in the container.
– Fast access operations. These operations take an item as a parameter to access to the object it refers. The item has to be in the container.

The main advantages that offers this library are:

– Provides a sizeable collection of data types and algorithms in a form that allows them to be used by non–experts. In the current version, this collection includes most of the data types and algorithms described in the text books of the area.
– Gives a precise and readable specification for each of the data types and algorithms mentioned above.
– For many efficient data structures access by position is important. LEDA uses an item concept to cast positions into an abstract form.
– It is a well–known library supported by documentation and books.
– LEDA is not in the public domain, but can be used freely for academic research and teaching.

The main drawbacks that present this library are:

– Lack of internal level of reusability. All the common capabilities have to be implemented on each concrete class.
– It is difficult to extend it. If we want to add a container component, it has to offer all the common capabilities but some of them restrict the possible implementations (e.g. we cannot add a new container whose objects change their position in the internal structure when a new object is inserted) and others difficult to understand and implementing.
– Non persistency of items. An item could be invalid if the object that it referred has been removed.
– Lack of an operation to know if an item is still in a container or not.

Figure 5 summarizes the most rellevant features of these libraries.

3 A Class Hierarchy for Container–like Component Libraries

The analysis carried out in the previous section has shown some problems that appear in many representative CLC–libraries. We are specifically interested in the strategy chosen for adding efficient access to objects stored in containers, which is an essential feature for obtaining a really useful solution in the CLC–libraries domain. Usually CLC–libraries allow this kind of efficient access by having alternative paths to objects in the containers, but we have shown that the concrete proposals (e.g., items in LEDA or iterators in STL) are really

	Class Hierarchy	Internal reuse	Multiple Implemen.	Extensibility	Iterators	Positions	Variety of components
STL	No	Non existing	Mostly no. Just for adaptors	Difficult	Bidirectional. Non persistent	Non persistent	Not much
LEDA	No	Non existing	Yes	Difficult	Iteration macros. Non persistent	Non persistent	Large
BC	Yes	Low and inappropriate	Yes but limited	Difficult	Unidirectional iterators. Non persistent	Non provided	Medium

Fig. 5. Summary of CLC–libraries characteristics

implementation–dependent, restricting somehow the quality of the library with respect to reusability, adaptability or applicability,

In the rest of the paper we are going to present a more structured approach for solving this problem, based on the use of a new design pattern, the *Shortcut*. Before this, in order to make our approach as library–independent as possible, we first define a common class–hierarchy for the container family; this hierarchy has been deduced from the concrete form that take the three CLC–libraries studied in 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.

More specifically, the hierarchy includes (see Fig. 6 for a concrete example of the hierarchy):

– A hierachy of *iterators*. Based on the *Iterator* design pattern. The most complete solution offers : input iterators, output iterators, forward iterators, bidirectional iterators and random iterators.
– *Shortcuts*. A new design pattern that introduces the concept of shortcut as the location or position of objects, together with some operations involving shortcuts. We present this pattern thoroughly in Sect. 4.
– A hierarchy of containers. We distinguish:
  • A base class *Container*. This base class acts as a common parent class for all kinds of containers. It provides the following functionalities:
    ∗ *Iterators* to iterate over the objects in a *Container*. The base class does not fix any particular form of iterator. Although the best case is having random iterators, they are difficult to implement; thus, concrete containers will commonly offer bidirectional iterators.
    ∗ A new *Insert* operation for inserting objects in the container returning a *Shortcut* to the object.

  * New operations of *access*, *modify* and *delete* objects in the *Container*, which take their *shortcut* as a parameter. These operations should be implemented with O(1) time.
- *Container Abstractions.* Children classes of *Container* that are not leafs, which represent different types of containers (list, map, etc.). Each of them adds its specific functionalities to the ones inherited from the *Container* class.
- *Container Abstraction Implementations.* Children classes of *Container Abstractions* that are leafs. Every leaf class is an implementation of its container parent. As such, they inherit all the functionalities of the concrete abstraction they implement. Unlike other possibilities (e.g., considering the implementation as a parameter), this approach allows implementations of abstractions can offer new functionalities (e.g., random iterators).

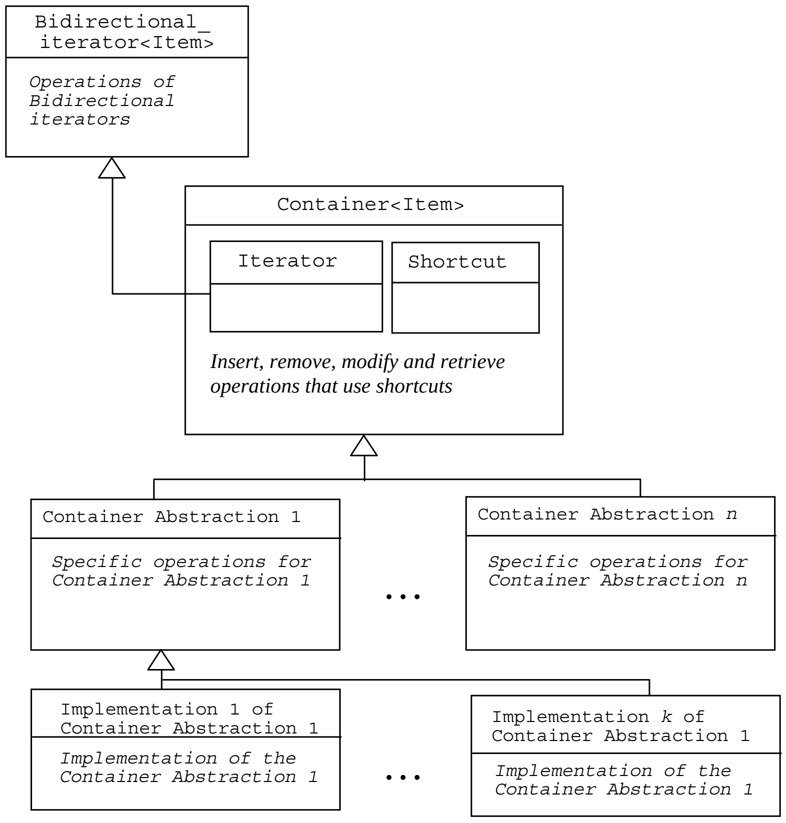

Bidirectional_ iterator<Item>
Operations of Bidirectional iterators
Container<Item>
Iterator
Shortcut
Insert, remove, modify and retrieve operations that use shortcuts
Container Abstraction 1
Specific operations for Container Abstraction 1
Container Abstraction n
Specific operations for Container Abstraction n
...
Implementation 1 of Container Abstraction 1
Implementation of the Container Abstraction 1
Implementation k of Container Abstraction 1
Implementation of the Container Abstraction 1
...

Fig. 6. Hierarchy of a common design for CLCL

4 The *Shortcut* Design Pattern

We present next the *Shortcut* design pattern mentioned in the previous sections using the format presented in [GHJ+96].

Intent

Define an object that encapsulates the concept of location or position of an object in a container. Provide an abstract, efficient and reliable way to access to the objects in a container without exposing its underlying implementation.

Also Known As

Location or position

Motivation

This pattern allows bridging the gap (presented in previous sections of this paper) between design and implementation in the domain of CLC–libraries. As a consequence, it solves the problems which damage reusability, adaptability and extensibility of CLC–libraries that are due to this gap. Moreover, this pattern provides fast access to the objects stored in a container in those contexts where efficiency is crucial. And finally, it allows to maintain robust references to objects stored in a container in other objects, which is a common practice in programming with CLC–libraries. We say that a location or position is robust if:

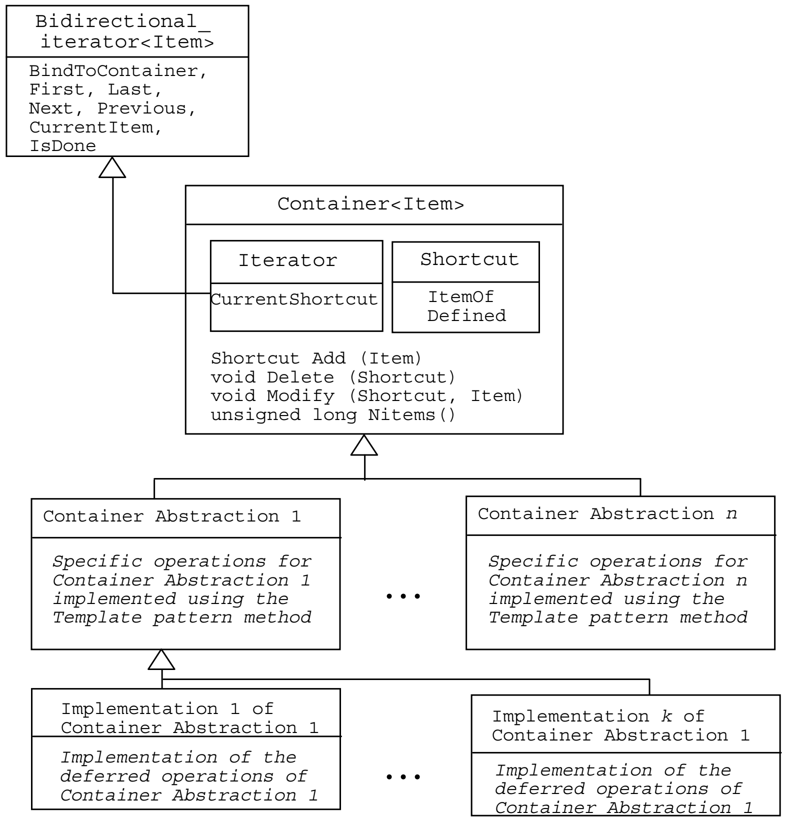
- It is bound to one and only one object in the container.
- It does not change while the object which it is bound to is inside the container, even if the underlying representation requires rearrangements.
- It is possible to know if it is bound to an object in the container or not.

Applicability

Use the *Shortcut* pattern to fulfill the following requirements altogether:

- To access to the objects stored in a container in an abstract and efficient way.
- To maintain robust references to objects stored in a container in other objects.
- To offer an uniform interface to perform efficient operations in a container.
- To increase the reusability, adaptability and extensibility of a CLC–library.
- To increase the internal level of reusability of the CLC–library itself. Remarkably, to implement iterators only in a container base class and reuse them in its children classes.

Structure


Bidirectional_ iterator<Item>
BindToContainer, First, Last, Next, Previous, CurrentItem, IsDone
Container<Item>
Iterator
CurrentShortcut
Shortcut
ItemOf
Defined
Shortcut Add (Item)
void Delete (Shortcut)
void Modify (Shortcut, Item)
unsigned long Nitems()
Container Abstraction 1
Specific operations for Container Abstraction 1 implemented using the Template pattern method
...
Container Abstraction n
Specific operations for Container Abstraction n implemented using the Template pattern method
Implementation 1 of Container Abstraction 1
Implementation of the deferred operations of Container Abstraction 1
...
Implementation k of Container Abstraction 1
Implementation of the deferred operations of Container Abstraction 1

Participants

– **Shortcut**
  - defines the Shortcut interface for fast access to the objects stored in a container.
  - implements the Shortcut interface.
  - keeps track of a persistent and safe location or position of an object stored in a container.

– **Container**
  - defines the interface of those container operations that involve shortcuts, i.e. the *add*, *delete*, *modify* and *retrieve* operations.
  - implements in an efficient manner the *add* operation in a way that returns a shortcut to the new object.
  - defines the interface of an operation to add an object after (in the iterator order) the object bound to a Shortcut, and implement it. This is required when iterations in orderings other than the implicit add–ordering are required.

  - implements in an efficient manner the *delete*, *modify* and *retrieve* operations by means of its shortcut.
- **Iterator**
  - defines the interface of bidirectional iterators including a new operation namely *CurrentShortcut* which returns the shortcut bound to the current item.
  - implements the interface using shortcuts.
- **Container Abstraction**
  - defines the interface of a container abstraction.
  - defines the objects (i.e., shortcuts) to be stored in every shortcut–based implementation of a container abstraction.
  - implements its interface using the Template Method pattern[1].
  - defines as protected the interface of the deferred operations (that we call concrete interface) and implement a (in some cases non–efficient) version of them using the container interface and shortcuts. This concrete interface consists in a concrete operation for each of the operations in the container abstraction interface.
- **Implementation of a Container Abstraction**
  - implements the concrete interface of a container abstraction without any assumption. Neither fast access operations nor iteration have to be explicitly added; they are inherited from the abstract class. Therefore there is any external restriction over the data structure to be used to implement the Container Abstraction.

Collaborations

- Shortcut keeps track if an object is still in the container and if it is the case the Shortcut allows accessing it.
- Container is responsible of notifying deletion of objects to the corresponding shortcuts.

Consequences

The Shortcut pattern offers several benefits:

1. *Abstraction.* The objects stored in a container can be accessed without knowing how they are stored therein and, therefore, without knowing the underlying representation of the container (with arrays, pointers, linked, in tree–form, ...).
2. *Efficiency.* The access to the objects in a container by means of a shortcut is done in constant time (see Implementation section below), making it possible to reuse containers even in those contexts with high efficiency constraints. Moreover, the efficiency of the iterator operations is $O(1)$ in the worst case[2].

[1] This pattern defines just the skeleton of an algorithm in an operation deferring some steps to subclasses of the class where the Template Method is used.

[2] We would like to remark that this low cost cannot be assured with implementation–dependent iterators, if these implementation is not well–suited for this purpose (e.g., a hashing implementation).

3. *Security.* The access to the objects by means of shortcuts is safe because meaningless access to them is avoided. In particular, the following situations are avoided: dangling shortcuts (references without bound objects) or out–of–date ones (shortcuts bound to objects which are not the original ones).
4. *Improving existing CLC–libraries.* As the addition of shortcuts to a concrete container does not modify its functional behaviour, preservation of behaviour supports both the reengineering process of existing CLC–libraries and the use of the new version in running applications without any modification (and the old library can be thus discarded). The preservation of behaviour is assured by incorporating the concept of shortcut into the formal specification of the container (see [FM00] for more details).
5. *External reusability.* The children classes of *Container* inherit its definition of shortcuts and the corresponding operations. Therefore, the objects stored in the *Container* specialisations can be accessed by using either the operations that characterise every particular type of container or the shortcut operations.
6. *Internal reusability.* The CLC–library is developed following step–by–step design and implementation; design and implementation for reuse; and design and implementation with reuse.
7. *Avoid commit iterations to a specific container implementation.* Iterations are made over the Container base class and as a consequence we can iterate over a container without commiting to an specific implementation[3].

Implementation

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

1. *Using dynamic storage.* In this case the Shortcut pattern is implemented with an *smart pointer*[4] [Ede92] to a tuple which contains the object and a deleted flag. On the other hand the Container base class is implemented with a double linked list of these tuples[5].
2. *Using an array.* In this case the *Shortcut* pattern is implemented as an index to the array position. Then the *Container* base class is implemented as an array of tuples which contain the object, a deleted flag, a reference counter (being every reference a shortcut) and two indexes to the next and previous tuples in the iterator ordering. As released shortcuts (the deleted array positions) must be available somehow to allow further reassignment we must 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 have to be maintained in the *Container* base class.

[3] Polymorphic iterators solve this problem too, but they introduce other drawbacks.
[4] Being its main characteristic that the deletion of allocated objects does not take place until there are no shortcuts bound to them.
[5] In order to have efficient bidirectional iterators.

A particular issue has to be taken into consideration. In order to get all the benefits that the *Shortcut* pattern offers we need some extra space. More precisely, being N the number of objects in the container, the total amount of extra space is:

$$2 * N * space(\text{pointer}) + N * space(\text{shortcut})$$

(we are assuming a pointer–based implementation of shortcuts, which is the worst case concerning space efficiency). The first operand comes from the double linked list and the second one from the shortcut stored in the implementation of a concrete abstraction. However, 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 objects. 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 references. Since generally,

$$space(\text{identifier}) \geq space(\text{pointer})$$

then $\exists k \geq 1$ s.t. $space(\text{identifier}) \geq k * space(\text{pointer})$,

and since $space(\text{shortcut}) = 2 * space(\text{pointer})$ (because we use two pointers for assuring that a particular shortcut is bound to a particular container) space is really saved when the relationship

$$R * space(\text{identifier}) \geq 4 * N * space(\text{pointer})$$

holds, which is satisfied when the following relationship holds:

$$R * k \geq 4 * N \ .$$

Sample code

We show here a C++ implementation of three participants in the pattern, Shortcut, Iterator and Container, together with the implementation of the Container Abstraction *Map* and a Concrete Array based Implementation of it. We present an example of use, too. The implementation of the pattern uses dynamic memory. The example of use illustrates how iterators are not bound to a concrete implementation.

1. *Implementation of the Shortcut class.* This class is in the interface of Container. This is the reason for which it is not presented as a template class. The Node and pointer classes are not included.

```
class Shortcut
{
    protected:
        pointer<Node> ptr;
        Container* ptrContainer;
        friend class Container<Item>;
        friend class Iterator;
    public:
        Shortcut() :ptrContainer(NULL) {}
        Shortcut(const Shortcut& shortcut) :ptr(shortcut.ptr),
                        ptrContainer(shortcut.ptrContainer) {}
        const Item& ItemOf()
        {
            if (!Defined()) throw UndefinedShortcut;
            return ptr.value().item;
        }
        bool Defined()
        {
            if (ptr.IsNull()) return false;
            if(ptr.value().deleted_flag)
            {
                ptr.SetNull(); return false;
            }
            return  true;
        }
        void operator =(const Shortcut &shortcut)
        {
            ptr = shortcut.ptr;
            ptrContainer = shortcut.ptrContainer;
        }
        ~Shortcut(){}
    protected:
        Shortcut (const Item& item)
        {
            ptr = pointer<Node>(Node());
            ptr.value().item = item;
        }
};
```

The implementation of this class is straightforward. It stores the `Container` along with a smart pointer `ptr` to the container node where is the item bound to the shortcut. The `Defined` operation returns true if there is a non deleted object bound to the shortcut and false otherwise. Notice, that the operation `ItemOf` and `Defined` are not declared as `const` because if the object associated to the shortcut is marked as deleted we set the pointer as `NULL` to dismiss the number of references. We can observe that all the operations are $O(1)$ in the worst case.

2. *Implementation of the Iterator class.* This class is in the interface of Container too. The iterator class is a subclass of a bidirectional_iterator abstract class.

```
class Iterator:public bidirectional_iterator<Item>
{
    protected:
        pointer<Node> ptr;
        Container<Item>* ptrContainer;
```

```
public:
    Iterator() :ptrContainer(NULL) {}
    Iterator(const Iterator& iterator) :ptr(iterator.ptr),
                    ptrContainer(iterator.ptrContainer) {}
    ~Iterator(){}
    virtual void BindToContainer(Container<Item>* C)
    {
        ptr = C->_container.value().next;
        ptrContainer = C;
    }
    virtual void First()
    {
        if(ptrContainer != NULL)
            ptr = ptrContainer->_container.value().next;
    }
    virtual void Last()
    {
        if(ptrContainer != NULL)
            ptr = ptrContainer->_container.value().previous;
    }
    virtual void Next()
    {
        if(!IsDone()) ptr = ptr.value().next;
    }
    virtual void Previous()
    {
        if(!IsDone()) ptr = ptr.value().previous;
    }
    virtual const Item& CurrentItem()
    {
        if(IsDone())
            throw IteratorIsDone;
        return ptr.value().item;
    }
    virtual bool IsDone()
    {
        if(ptrContainer == NULL) return true;
        if(ptr.IsNull()) return true;
        if(ptr.value().deleted_flag)
        {
            ptr.SetNull(); return true;
        }
        if(ptr == ptrContainer->_container) return true;
        return false;
    }
    virtual void operator =(const Iterator &iterator)
    {
        ptr = iterator.ptr;
        ptrContainer = iterator.ptrContainer;
    }
```

```
            virtual Shortcut CurrentShortcut()
            {
                Shortcut sh;
                if(IsDone()) throw IteratorIsDone;
                sh.ptr = ptr;
                sh.ptrContainer = ptrContainer;
                return sh;
            }
    };
```

This class offers a new operation `CurrentShortcut` that returs the Shortcut associated to the `CurrentItem`. As in the case of the class shortcut, all the operations have constant cost in the worst case.

3. *Implementation of the class Container.* The classes Shortcut and Iterator shown before are included in the public Container interface.

```
    template<class Item>
    class Container{
            // Definition of the class Node.
            // ...
            class Iterator; // It's necessary because the class
                            // Shortcut declares it as friend
        public:
            // Definition of the class Shortcut
            // ...
            // Definition of the class Iterator
            // ...
        protected:
            friend class Iterator;
            friend class Shortcut;
            pointer<Node> _container;
            unsigned long count;
        public:
            Container();
            ~Container();
            virtual Shortcut Add(const Item &item);
            virtual void Delete(Shortcut &shortcut);
            virtual void Modify(Shortcut& shortcut, const Item& item);
            virtual unsigned long NItems();
    };
```

Constructor and destructor operations.

```
    template <class Item>
    Container<Item>::Container() :_container(Node()), count(0)
    {
        _container.value().next = _container;
        _container.value().previous = _container;
    }

    template <class Item>
```

```
Container<Item>::~Container()
{
    pointer<Node> p,a;
    a = _container.value().next;
    while(a !=_container)
    {
        p = a.value().next;
        a.value().deleted_flag = true;
        a.value().next.SetNull();
        a.value().previous.SetNull();
        a = p;
    }
    _container.value().deleted_flag = true;
    _container.value().next.SetNull();
    _container.value().previous.SetNull();
}
```

`Add` adds the item in the container and return the shortcut that allows fast access to it.

```
template <class Item>
Container<Item>::Shortcut Container<Item>::Add(const Item &item)
{
    Shortcut sh(item);
    _container.value().previous.value().next = sh.ptr;
    sh.ptr.value().previous = _container.value().previous;
    sh.ptr.value().next = _container;
    _container.value().previous = sh.ptr;
    count++;
    sh.ptrContainer = this;
    return sh;
}
```

`Delete` implements the deletion in a double link list. It marks as deleted the item associated to the shortcut, if any, set as `NULL` the shortcut members `ptr` and `ptrContainer`, and the `Node` members pointers `next` and `previous`.

```
template <class Item>
void Container<Item>::Delete(Shortcut &shortcut)
{
    if (shortcut.Defined())
    {
        if(shortcut.ptrContainer != this)
            throw ShortcutIsNotBoundToThisContainer;
        shortcut.ptr.value().deleted_flag = true;
        shortcut.ptr.value().previous.value().next =
                            shortcut.ptr.value().next;
        shortcut.ptr.value().next.value().previous =
                        shortcut.ptr.value().previous;
        shortcut.ptr.value().next.SetNull();
        shortcut.ptr.value().previous.SetNull();
```

```
            shortcut.ptr.SetNull();
            count--;
        }
    }
```

Modify changes the item associated to the shortcut, if any, by the new one.

```
    template <class Item>
    void Container<Item>::Modify(Shortcut& shortcut, const Item& item)
    {
        if(shortcut.Defined())
        {
            if(shortcut.ptrContainer != this)
                throw ShortcutIsNotBoundToThisContainer;
            shortcut.ptr.value().item = item;
        }
    }
```

NItems is straightforward. It returns the value of the internal counter.

```
    template <class Item>
    unsigned long Container<Item>::NItems()
    {return count;}
```

4. *Map Container Abstraction.* We show here the interface and implemention of this class. This class defines the type of the objects to be stored in its concrete subclasses (in this case pairs of Shortcuts) and defines a new compare class which allows to compare keys with shortcuts (comparing the key with the key associated to the shortcut). The implementation of this class uses the Template Method to implement its public interface. Therefore, its children classes have to only implement the operations deferred in the implementation of this public interface (concrete operations) which are defined in the protected part. On the other hand, this class provides an implementation of this protected interface that makes it a non–abstract class.

```
    template <class Key, class Value, class Less=less<Key> >
    class Map:public Container< KeyValue<Key,Value> >
    {
        protected:
            class ConLess
            {
                protected:
                    Less _less;
                public:
                    ConLess(Less l):_less(l){}
                    bool operator()(const Key& key, Shortcut& sh)
                    {
                        return _less(key,sh.ItemOf().key);
                    }
                    bool operator()(Shortcut& sh, const Key& key)
                    {
                        return _less(sh.ItemOf().key,key);
                }
            };
            typedef Key ConKey;
            typedef Shortcut ConValue;
            typedef KeyValue<Shortcut,ConValue> ConItem;
            Less less;
            ConLess conless;
            Iterator cache;
        public:
            typedef KeyValue<Key,Value> Item;
            Map():conless(less) {cache.BindToContainer(this);}
            ~Map(){}
            Shortcut Add(const Item& item);
            void Delete(Shortcut& shortcut);
            void Modify(Shortcut& shortcut, const Item& item);
            Value Delete(const Key& key);
            const Value& Get(const Key& key);
            bool Exist(const Key& key);
        protected:
            virtual void ConAdd(const ConItem& item){}
            virtual ConValue ConDelete(const ConKey& key);
            virtual ConValue ConGet(const ConKey& key);
            virtual bool ConExist(const ConKey& key);
    };
```

`Add` operation of the class Map. This operation deferres the operations `ConExist`, `ConGet` and `ConAdd` to the concrete implementation and adds or modifies the item in the container base object.

```
    template <class Key, class Value, class Less=less<Key> >
    Container< KeyValue<Key,Value> >::Shortcut Map<Key,Value,Less>
    ::Add(const Item& item)
    {
        Shortcut sh;
        if(ConExist(item.key))
        {
            sh = ConGet(item.key);
            Container<Item>::Modify(sh,item);
        }
        else
        {
            sh = Container<Item>::Add(item);
            ConItem itemcon(sh,sh);
            ConAdd(itemcon);
        }
        return sh;
    }
```

`Delete` (by shortcut) operation of class Map. This operation deletes the item associated to the shortcut using its key.

```
    template <class Key, class Value, class Less=less<Key> >
```

```
void Map<Key,Value,Less>::Delete(Shortcut& shortcut)
{
    if(shortcut.Defined())
        Delete(shortcut.ItemOf().key);
}
```

`Modify` operation of class Map. In this Container Abstraction this operation have to be overrided to allow only modifing the value and not the key.

```
template <class Key, class Value, class Less=less<Key> >
void Map<Key,Value,Less>
::Modify(Shortcut& shortcut, const Item& item)
{
  if(shortcut.Defined())
    if(!conless(shortcut,item.key) && !conless(item.key,shortcut))
           Container<Item>::Modify(shortcut,item);
}
```

`Delete` (by key) operation of class Map. This operation deletes the pair of shortcuts stored in the concrete implementation of the container abstraction using the `ConDelete` operation and deletes the item from the Container base object using the shortcut.

```
template <class Key, class Value, class Less=less<Key> >
Value Map<Key,Value,Less>::Delete(const Key& key)
{
    Shortcut sh;
    Value v;
    sh = ConDelete(key);
    v = sh.ItemOf().value;
    Container<Item>::Delete(sh);
    return v;
}
```

`Get` operation of class Map. This operation uses the `ConGet` to get the shortcut of the key and returns the value of the item associated to this shortcut.

```
template <class Key, class Value, class Less=less<Key> >
const Value& Map<Key,Value,Less>::Get(const Key& key)
{
    Shortcut sh;
    sh = ConGet(key);
    return sh.ItemOf().value;
}
```

`Exist` operation of class Map.

```
template <class Key, class Value, class Less=less<Key> >
bool Map<Key,Value,Less>::Exist(const Key& key)
{
    return ConExist(key);
}
```

Following we give a version of the concrete operations of class Map using the the container interface and shortcuts.
`ConDelete` concrete delete operation of class Map.

```
template <class Key, class Value, class Less=less<Key> >
Map<Key,Value,Less>::ConValue Map<Key,Value,Less>
::ConDelete(const ConKey& key)
{
    ConValue v;
    if(ConExist(key))
    {
        v = cache.CurrentShortcut();
        cache.Previous();
        return v;
    }
    throw NotExistingKey;
    return ConValue();
}
```

`ConGet` concrete get operation of class Map.

```
template <class Key, class Value, class Less=less<Key> >
Map<Key,Value,Less>::ConValue Map<Key,Value,Less>
::ConGet(const ConKey& key)
{
    if(ConExist(key))
        return cache.CurrentShortcut();
    throw NotExistingKey;
    return ConValue();
}
```

`ConExist` concrete exist operation of class Map.

```
template <class Key, class Value, class Less=less<Key> >
bool Map<Key,Value,Less>::ConExist(const ConKey& key)
{
    Iterator it;
    if(!cache.IsDone())
        if(!less(key,cache.CurrentItem().key) &&
            !less(cache.CurrentItem().key,key))
            return true;
    it.First(this);
    while(!it.IsDone())
    {
        if(!less(it.CurrentItem().key,key) &&
            !less(key,it.CurrentItem().key))
        {
            cache = it; return true;
        }
        it.Next();
    }
    return false;
}
```

5. *An Implementation of the Container Abstraction: Map.* We show here an implementation namely `MapArray`, that makes rearrangements of its elements when there occurs a deletion. Note that, in spite of this, we can use shortcuts because they are persistent. This class only must implement the concrete deferred operations `ConAdd`, `ConDelete`, `ConExist` and `ConGet` which work with the concrete types: `ConItem`, `ConValue` and `ConKey` and with the concrete comparison operation `ConLess`, offered by its parent class `Map`.

```
template <class Key, class Value, class Less=less<Key> >
class MapArray:public Map<Key,Value,Less>
{
    protected:
        ConItem mapArray[1000];
        int pos;
        int cache;
    public:
        MapArray() :pos(0),cache(-1) {}
    protected:
        virtual void ConAdd(const ConItem &it);
        virtual ConValue ConDelete(const ConKey& key);
        virtual bool ConExist(const ConKey& key);
        virtual ConValue ConGet(const ConKey& key);
};

template <class Key, class Value, class Less=less<Key> >
void MapArray<Key,Value,Less>::ConAdd(const ConItem &it)
{
    mapArray[pos] = it;
    cache = pos;
    pos++;
}

template <class Key, class Value, class Less=less<Key> >
MapArray<Key,Value,Less>::ConValue MapArray<Key,Value,Less>
::ConDelete(const ConKey& key)
{
    ConValue v;
    int i;
    if(ConExist(key))
    {
        v = mapArray[cache].value;
        pos--;
        for(i=cache; i < pos; i++)
            mapArray[i] = mapArray[i+1];
        cache--;
        return v;
    }
    throw NotExistingKey;
    return v;
}
```

```
template <class Key, class Value, class Less=less<Key> >
bool MapArray<Key,Value,Less>::ConExist(const ConKey& key)
{
    int i;
    if(cache != -1)
        if(!conless(key,mapArray[cache].key) &&
            !conless(mapArray[cache].key,key))
            return true;
    for(i=0;i<pos;i++)
        if(!conless(key,mapArray[i].key) &&
           !conless(mapArray[i].key,key))
        {
            cache = i;
            return true;
        }
    return false;
}

template <class Key, class Value, class Less=less<Key> >
MapArray<Key,Value,Less>::ConValue MapArray<Key,Value,Less>
::ConGet(const ConKey& key)
{
    if(ConExist(key))
        return mapArray[cache].value;
    throw NotExistingKey;
    return ConValue();
}
```

6. *An example of using iterators without binding them to a concrete container.* We can define a generic function to print the collection of objects of a container that works for any kind of container.

```
template <class Item>
void print(Container<Item> *C)
{
  Container<Item>::Iterator it;
  cout <<"Printing the " << C->NItems() <<" items in container: ";
  it.BindToContainer(C);
  it.First();
  while(!it.IsDone())
  {
      cout << it.CurrentItem() << ' ';
      it.Next();
  }
  cout << ":Last\n";
}
```

The next piece of code show its use applied to a `MapArray` implementation of the Map container.

```
MapArray<int,int> C;
// code to insert elements
print(&C);
```

Known Uses

The Shortcut pattern is widely used to keep track of references to objects stored in a container, in other objects. A typical example is to maintain in an object that is in a container a reference to other object in the same container (e.g., a parent–children relationship in a list of people).

Another common use is to combine diferent data structures in an efficient manner to obtain new efficient ones. An example is the case of a symbol table in the compilation field. Although a symbol table can be described and understood in terms of classical containers (stacks, list and map), normally they are not really reused; instead, the data structure is represented directly in terms of arrays and pointers. We can find an example in [WM95].

Related Patterns

Iterator: Shortcuts allow avoid make differents implementations of Iterators for each of the different Containers Abstractions.

Template Method: The implementation of the Container Abstractions use this pattern.

5 A Case Study: Reengineering the BC Library

In this section we summarize the main results of applying our approach to the Ada 95 version of the Booch Components presented in Sec. 2. The technical details can be found in [MF00].

The unique difference with the pattern presented in this paper is that in order to preserve the interface of the original version of the library, this interface has been enlarged by a new operation namely `Shortcut_To_The_Last_Item_Added`. This operation returns the shortcut associated to the last item added to a container. Then we have to use the next piece of code:

```
Containers.Add(C,I);
sh := Containers.Shortcut_To_The_Last_Item_Added(C);
```

instead of

```
sh := Containers.Add(C,I);
```

in order to obtain the corresponding shortcut (because the original `Add` operation is not a function).

The main results of applying the Shortcut approach to the Booch Components library are summarized next:

– The Booch library have been improved from many points of view:
  - Adaptability. The original library is not extensible enough. Its hierarchy restricts to a single set of possible implementations for some of the different structure abstractions. Therefore, its adaptability is damaged. It is not the case of the new version.
  - Applicability. Not only the posibility of extending the new version but also the efficiency that shortcuts provide makes the library to be reusable in more different contexts.
  - Functionality. In addition to the new functionalities of shortcuts, the new iterators are bidirectional while the old ones were not.
  - Efficiency. In the old version, not only the lack of multiple implementations for components damages efficiency, but also some of the iterator operations have lineal cost in the worst case with respect to a certain parameter (although their amortised cost is constant). Our version solve these problems and offers the efficient operations that use shortcuts.
  - Robustness. The original library is not robust enough with respect to changes in some of their components. The new version avoids this problem.

– This improvement has been made in a very comfortable way; just a few changes are needed. The core data structures and algorithms are the same without any modification at all.

– The reengineering process does not interfere with the previous behaviour of the library (both for functionality and for efficiency) and, in consequence, existing software that use this library does not need to be modified; only recompilation is needed.

– But existing software could be modified in a methodical way (basically, changing the way of accessing to the structure) to take profit of the new version of the library. New software, of course, will be built in general using the new layout of the structure, making intensive use of shortcuts.

– The new library has been tested using the *test packages* provided by the original Booch library without any modification.

In our opinion, this reengineering case study has been a good example of how our approach can be used not only to design and implement new CLC–libraries but also to improve existing ones.

6 Conclusions

In this paper, we have presented the *Shortcut* design pattern as a mean for bridging the gap between design and implementation in the container–like component libraries domain. The design pattern enhances the quality of the internal structure of the library while supporting also efficient access to the objects stored in the containers.

We think that the most rellevant contributions of our work are the following:

- Separation of concerns. Independent definition of the functionality offered by the containers, and the operations for the efficient access to objects, make libraries well–structured and supports their better understanding.
- Usability. As a result of this separation of concerns while keeping the containers efficient, libraries can be used in a large number of contexts.
- Structural quality. Applying our approach has some interesting benefits which are rarely offered altogether in the existing libraries: access by position is implementation–independent; there are no restrictions on the number and characteristics of container implementations; libraries can be customised (by means of specializations of existing containers) and enlarged (creating new types of containers or implementations of them).
- Standarisation. The proposal has been introduced using a widely accepted design tool, a design pattern. As a result, it can be easily understood and integrated with other similar concepts, as the one of iterator.
- Compatibility. As shown in section 5, applying the shortcut design pattern to existing libraries should not be difficult at all. The quality of the reengineered library is improved from many points of views and the new version is compatible with the old one, avoiding then modification of running programs that use the library and making unnecessary the existence of the two versions at the same time.

References

[Ada95] S. Tucker Taft and R.A. Duff (Eds.). *Ada 95 Reference Manual.* Lecture Notes in Computer Science 1246, Springer–Verlag, 1995.

[Boo87] G. Booch. *Software Components with Ada.* The Benjamin/Cummings Publishing Company, 2nd edition, 1987.

[BV90] G. Booch and M. Vilot. The Design of the C++ Booch Components. In *Proceedings of Conference on Object Oriented–Programming: Systems, Languages and Applications (OOPSLA)*, volume 25 of *SIGPLAN Notices*, pages 1–11. ACM, 1990.

[BWW99] 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.

[Ede92] D.R. Edelson. Smart pointrs: They're smart, but they're not pointers. In *Proceedings of the 1992 USENIX C++ Conference*, pages 1–19. USENIX Association, 1990.

[FM00] 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.

[GHJ+96] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns. Elements of Reusable Object–Oriented Software.* Addison–Wesley, 1996.

[Mey97] B. Meyer. *Object–Oriented Software Construction, Second Edition.* Prentice Hall, 1997.

[McI69] M. McIlroy Mass Produced Software Engineering In *Software Engineering Concepts and Techniques.* NATO Conference on System Sciences, 1969.

[MF00] 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.

[MN99] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[MS96] D.R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison–Wesley, 1996.

[Str91] B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, 2$^{\text{nd}}$ edition, 1991.

[WM95] R. Wilhem and D. Maurer. *Compiler Design*. Addison–Wesley, 1995.