Reengineering the Booch Component Library*

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. Component-based software development heavily relies on the ability of reusing components from a library with as little effort as possible. Among others, valuable features for reusing from a component library are: adaptability to many contexts, extensibility, abstraction and high level of robustness with respect to changes in some of their components. In this paper we study one of the most widely used component library for Ada 95, the *Grady Booch's* one, mainly in relation to these features. Our study focuses on the *Container-classes* family, which present some drawbacks mainly due to the fact that some parent-classes depend on the concrete form of their children-classes. We propose a solution centred on changing the *Containers* base class. This new version of the *Containers* class offers a new concept, namely shortcut, that allows not only to avoid the dependencies between parent-classes and their children classes, but also offers some additional advantages, remarkably improving the efficiency of components.

1 Introduction

Component-based software development [Jaz95,Sit+94] heavily relies on the ability of reusing components from a library with as little effort as possible. Reusing from a library can only take place when the library fulfils some nice properties. To name a few:

- It should be versatile, offering a wide range of components with different functionalities, and also with the same functionality but different implementation strategies (each one suited for particular efficiency requirements).
- It should be open, in the sense that its users should be able to add easily other components to adapt the library to their own context.
- It should be stable, to avoid new future releases forcing changes on existing software.
- It should be abstract enough, to allow easy use.
- And of course, it should be correct and properly documented.

* This work has been partially supported by the Spanish research programme CICYT under contract TIC97-1158.

Marco, J.; Franch, X. Reengineering the booch component library. A: International Conference on Reliable Software Technologies. "Reliable Software Technologies Ada-Europe 2000: 5th Ada-Europe International Conference: Potsdam, Germany, June 26-30, 2000: proceedings". Berlín: Springer, 2000, p. 96-111.
The final authenticated version is available online at https://doi.org/10.1007/10722060_11

One of the best well-known component libraries for Ada 95 [Ada95] is the Grady Booch's one. This library was originally created for Ada 83 [Boo87] and reengineered first for C++ [BV90] and later on for Ada 95 [BWW99]. This library fulfills many of the properties listed above, and it has proven to be very useful in the development of component-based software. However, it also presents some drawbacks that makes it not as powerful as it could be with respect to the following criteria:

- **Versatility.** It offers a wide range of components, but it is not as versatile as one could expect, because all its components have just a single implementation.
- **Efficiency.** Not only does the lack of many implementations provoke a loss of efficiency in some contexts, but some of these implementations are not efficient enough, because component interfaces do not allow direct access to certain parts of the implementation.
- **Openess.** New components and implementations for components could be added, but implementations for already existing components should look very similar to the existing ones, due to the internal structure of the library.
- **Stability.** Some feasible changes on implementations can damage existing programs, because component definitions make use in some places of features that are inherent of proper implementations.
- **Usage.** The mixture of specification and implementation characteristics in component definitions interferes with the easy understanding and use of the components therein.

The purpose of this paper is to make reengineering on the Booch library to solve these problems. The proposal is based mainly on changing a particular base class, *Containers*, in such a way that all the dependencies on a concrete implementation disappear. This new class allows easy integration in the library of new components and also of new arbitrary implementations for every component, even the already existing ones. Moreover, the new *Containers* class offers an enlargement in its interface that make its derived classes more efficient in a wider range of contexts. All the changes in the library do not affect existing programs which are using it; it only needs a recompilation step. However, existing programs could be improved with this new version of *Containers* to make them more efficient.

The rest of the paper is organised as follows. First, in Sect. 2, we analyse the Booch component library for Ada 95 describing the advantages and drawbacks that it presents. Then, in Sect. 3, we introduce the main features of a new *Containers* class that allows solving the problems that the *Booch* library presents. Section 4 explains the implementation details of the new *Containers* class. Next, in Sect. 5, we show the necessary modifications of the *Containers* children-classes and an example. Finally, in Sect. 6, we give the conclusions and future work.

2 Analysing the Booch Component Library for ADA 95

The Booch component library is one of the best-known component libraries for Ada 95 [Ada95]. This library was originally created for Ada 83 [Boo87] and reengineered first for C++ [BV90] and later on for Ada 95 [BWW99]. The Ada 95 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 1 shows the main hierarchy of these components; their code is available at [BWW99].

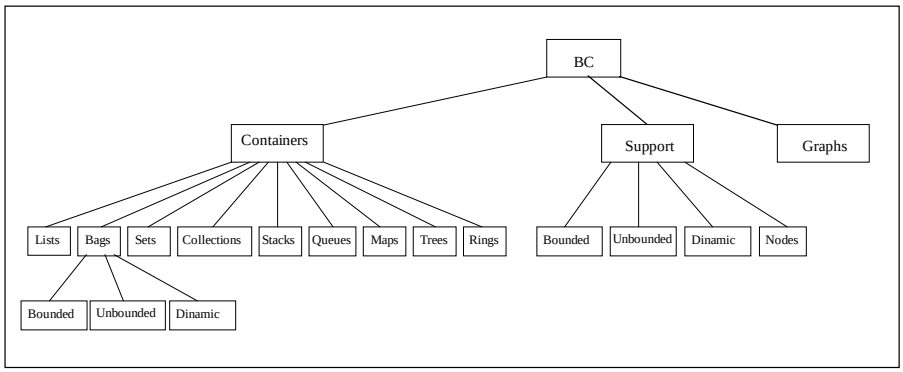

Fig. 1. Hierarchy of the Booch component classes (excerpt)

Our study is centred on the *Container* family of classes, which 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 clearer, Fig. 2 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*[1] 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.

[1] An iterator is a separate abstract data type, found in most of the existing comercial libraries, that allows iterating through all the items in a structure. Its existence and good behaviour is critical for a library being competitive, and this is why all of the most important component libraries offer them in one way or another.

```
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. 2. Extract of the generic package *BC.Containers.Bags*

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.
- 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. 3, the *Reset* operation could have linear cost with respect to the *Number_of_Buckets*. A similar problem occurs in the *Next* operation.

```
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. 3. Reset operation's code of the generic package *BC.Containers.Bags*

– 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 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.

It can be argued that these disadvantages are strong enough to reject the use of the Booch library. However, this is not the case; this library offers several advantages that make it very useful, mainly:

– 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.

Given the above advantages we think it is worth improving this library by solving the above mentioned problems, instead of discarding it and looking for building a brand new one.

3 A New *Containers Class*

The goal of this paper is to improve the containers-class family of Booch components library to avoid the problems shown in the previous section. Our approach consists on changing the base class *Containers* in such a way that the items will be stored in a generic container. Then iterators are implemented in this generic container and so they are independent of the concrete implementation of the children-classes. This generic container will offer efficient access paths to

the items therein, which we call *shortcuts*. The *Containers* children-classes must store the shortcuts (that allow access to items) instead of the items themselves. In addition, the users can obtain an alternative, efficient and abstract way to access to the items stored in the structure.

The children classes of the new *Containers* class inherit from it the shortcuts and the operations. Therefore, the items stored therein can be accessed not only by using the operations given in the former specification, but also by means of other new ones that use the efficient paths. This results very useful whenever the operations are considered too expensive; on the other hand, since the addition of shortcuts does not affect the former behaviour of the existing operations, they can be used in the other cases as well.

Shortcuts solve the problems mentioned in the previous section:

- The items in the container can be accessed without knowing how they are stored in the container and, therefore, without knowing the underlying representation of the container (with arrays, pointers, linked, in tree-form, ...). Therefore, many implementations can exist together for the same type of container.
- The access to the items in a container by means of a shortcut is achieved in constant time, making it possible to reuse containers even with high efficiency constraints.
- The addition of shortcuts to a container does not modify its functional behaviour. This is assured by incorporating the concept of shortcut into the formal specification of the container (see [FM00] for more details). Preservation of behaviour makes possible the substitution of old components by new ones.

Shortcuts are created and destroyed dynamically as items are inserted to and removed from a container; every item is bound to one (and only one) item. The shortcut bound to an item remains the same while it is inside the container, even if the underlying representation requires rearrangements. The access to the items by means of shortcuts is safe because meaningless access to them is avoided: our approach avoids dangling shortcuts or out-of-date ones.

4 Implementing the New *Containers* Class with Shortcuts

The essential point consists in implementing the generic container with a mapping from shortcuts to items. At the same time, items in the children components must be substituted by the shortcut that identifies them. The mapping from shortcuts may be implemented both using dynamic storage or an array; shortcuts are implemented then as pointers or cursors (i.e., array positions), respectively. Fields to obtain (in constant time) the shortcuts to the last item stored and to the *first* and the *last* item in the iterator order are necessary too; the field *first* to reset an iterator and the field *last* to add an item after the previous last item in the iterator order (which is not the same that the last item stored because it can have been added anywhere).

Released shortcuts must be available somehow to allow further reassignment, provided that there are not extra copies of it. When an item is removed from the container, it is marked as deleted. The corresponding shortcut can only be reused when there are no references to it. Therefore, memory management should be incorporated in the implementation. In this paper, shortcuts are implemented by pointers reusing a particular memory manager offered by the Booch's library itself.

To assure efficient and independent iterators we need to create a double-linked list of shortcuts, which means that we need $2 * N * space(\text{pointer}) + N * space(\text{shortcut})$ extra space (where N is the number of items in the container). Then, the iterators are implemented as shortcuts. The waste of this extra space offers a lot of benefits: the iterators are independent of the concrete form of the container, the efficiency of all the iterator operations is constant even in the worst case, and even this waste of space will generate a later saving, when shortcuts substitute identifiers (generally strings, which require more space than pointers or integers) in outer references from programs or other data structures.

Let's now establish formally the equation that assures the saving of space. Let N be the total number of items in the container and R the total number of external references. Since generally,

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

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

and since $space(\text{shortcut}) = 2 * space(\text{pointer})$ (see below) space is really saved when the relationship

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

holds, which is satisfied when the following relationship holds:

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

Figure 4 shows the implementation of the types of the new class *BC.Containers*. The *Shortcut* type is implemented with a record with two fields: the first one is an access to the *Container* to which the shortcut is associated and the last one is a pointer (reused from *BC.Smart*) to the node where the item is stored. The *Node* type is a record with four fields: the first one is the item itself, the second and the third ones are pointers to maintain doubled-linked list of items and the last one is a boolean to mark the item as deleted when it has been logically removed but there are still some shortcuts that refer to it. The *Container* type is a record that maintains the corresponding fields for the last, first and last added items, and an additional field to obtain the number of items therein in constant time.

```
with BC.Smart;
generic
  type Item is private;
  with function "=" (L, R : Item) return Boolean is <>;
package BC.Containers is
  type Container is new Ada.Finalization.Controlled with private;
  type Shortcut is private;
  type Iterator is private;
  function "=" (L, R : Shortcut) return Boolean;
...
private
  type Node;
  type Access_Node is access Node'Class;
  type Access_Node_P is access Access_Node;
  package SP is new BC.Smart (T => Access_Node, P => Access_Node_P);
  type Shortcut_Pointer is new SP.Pointer;
  function Create_Node(C : Container) return Access_Node;
  type Access_Container is access all Container;
  type Shortcut is record
      Position: Shortcut_Pointer;
      For_The_Container: Access_Container := null;
    end record;
  type Node is tagged record
      Elem: Item;
      Next: Shortcut_Pointer;
      Previous: Shortcut_Pointer;
      Deleted: Boolean := FALSE;
    end record;
  type Iterator is new Shortcut;
  type Container is new Ada.Finalization.Controlled with record
      Cardinality: Natural := 0;
      First_Item: Shortcut_Pointer;     -- For access to the first and
      Last_Item: Shortcut_Pointer;      -- the last items by shortcuts.
      Last_Item_Added: Shortcut_Pointer;-- To obtain the shortcut bound
    end record;                         -- to the last item added.
...
end BC.Containers;
```

Fig. 4. New *BC.Containers* class

This scheme also works, without further considerations, in the case of the linear containers, such as stacks, lists, and so on. Figure 5, on the left, shows this situation with stacks. Now, the stack contains just shortcuts, and the objects are stored in a generic container, directly accessible by means of the shortcut. In the case of types accessed by some kind of key, like bags, maps, etc., we need to define the functions required for the concrete implementation over the shortcuts (for instance, the equality function and, in case of a hashing table, the hash function). These new functions will consist in applying the original function to the item associated to the shortcut. The scheme is shown in Fig. 5, on the right, and the Ada 95 code can be found in [MF99].

The new *Containers* class includes, in addition to the operations offered by the old one, seven new ones shown below. The first three ones are public while the others are private and, as a consequence, can only be used by the children

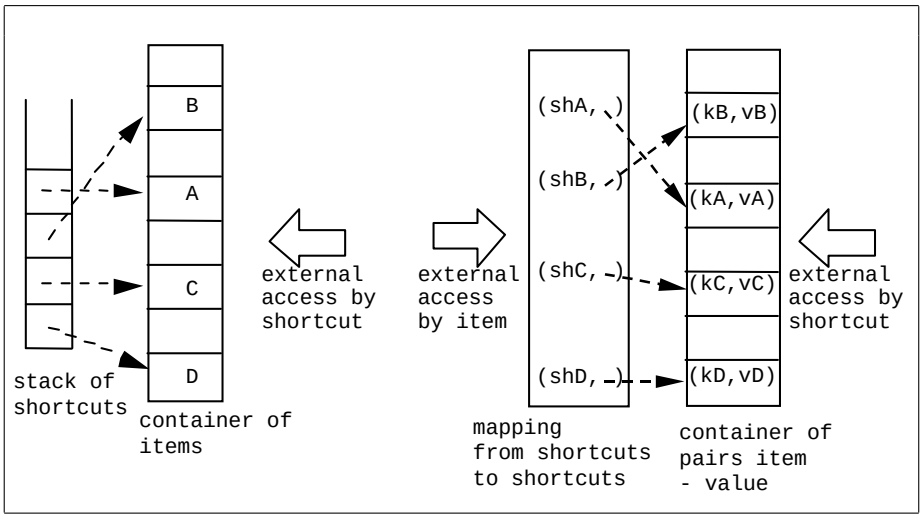

Fig. 5. Adding shortcuts to linear structures (left) and to mappings (right)

classes (Fig. 6 shows the signature of these operations; see [MF99] for a complete implementation). Their behaviour is as follows:

- *Add* adds the given item into the container in the last position (with respect to the iterator order). To be used in concrete containers implementations that do not require a special traversing order.
- *Add_After* adds the given item after the one bound to the given shortcut (with respect to the iterator order). If the shortcut is undefined, an exception arises. To use in concrete containers implementations that require a special traversing order.
- *Shortcut_To_The_Last_Item_Added* returns the shortcut that allows accessing to the last item added (using *Add* or *Add_After*) into the container. If there are no items in the container, it returns an undefined shortcut. This operation is required to obtain the shortcuts from the data structure; once obtained, it can be stored in other data structures and then coupling of structures without duplication of items is possible.
- *Item_Of* returns the item bound to the given shortcut. If the shortcut is undefined, an exception arises.
- *Defined* returns true if there is an item associated to the given shortcut, otherwise returns false.
- *Remove* removes from the container the item associated to the given shortcut. If the shortcut is undefined, an exception arises.
- *Node_Of* returns an access to the node where the item associated to the given shortcut is stored. If the shortcut is undefined, an exception arises. This operation is required in order to access to the values that have been stored together with the item, if it is the case. In the children classes, public efficient operations to access to these values by shortcuts can be defined. To allow this, the *Node* type is tagged and the children classes can define a new *Node* type with the necessary extra fields; in this case, the *Create_Node* function (see Fig. 4) has to be overridden.

```
...
generic
...
package BC.Containers is
...
  function Shortcut_To_The_Last_Item_Added
                     (In_The_Container : Container) return Shortcut;
  function Item_Of (The_Shortcut : Shortcut) return Item;
  function Defined (The_Shortcut: Shortcut) return Boolean;
...
private
...
  procedure Add (In_The_Container:in out Container'Class; Elem:Item);
  procedure Add_After (The_Shortcut: Shortcut;
                       In_The_Container : in out Container;
                       Elem: Item);
  procedure Remove (The_Shortcut : Shortcut);
  function Node_Of(S : Shortcut) return Access_Node;
...
end BC.Containers;
```

Fig. 6. Signature of the new operations of the *BC.Containers* class

We remark that the cost of all these operations as well as all the iterator operations is constant even in the worst case. Another important feature is that the operations *Shortcut_To_The_Last_Item_Added*, *Item_Of* and *Defined* are public and they are inherited by the children-classes of Containers. In consequence, all the children classes offer shortcuts and their users can take profit of them to improve the efficiency of their programs.The complete implementation can be found not only in [MF99] but also in `ftp://ftp.lsi.upc.es/pub/users/jmarco/`.

5 Using the New *Containers* Class

In this section, we explain the steps required to modify the *Containers* children-classes to adapt them to the new *Containers* class. First, we explain the general changes required and then we applied the required changes to a concrete example, the class *BC.Containers.Bags.Bounded* and its ancestors.

In the general case, the changes required in the *Containers* children-classes are:

- Remove the specification and the implementation of the concrete iterators (i.e., the new *Actual_Iterator*).
- Modify the operations that take into account features that are tied to the concrete implementation of the new *Actual_Iterator*.
- In the class where the container is implemented we must:
  - Replace the items by shortcuts.
  - If there are elements to store together with the item (as in the example below), then define a new *Node* type and override the function *Create_Node* to initialise these elements.
  - If the existing implementation needs some specific functions (e.g., a hash function in the case of a container implemented by hashing), then redefine them working on shortcuts instead of items.

- For every operation that works with items, add the indirection required to work with shortcuts instead of items.

We present below an example of the (few) changes to be done in the *Containers* children-classes. More precisely, we explain the required changes in the class *BC.Containers.Bags* and *BC.Containers.Bags.Bounded*, respectively. Similar changes should be done to the rest of children-classes.

The only required change in the package specification *BC.Containers.Bags* is to remove the iterators therein, i.e. the new *Actual_Iterator* type as well as its corresponding operations (Fig. 7 shows the part of the specification that must be removed).

```
...
generic
...
package BC.Containers.Bags is
...
private
...
  type Bag_Iterator (B : access Bag'Class)
  is new Actual_Iterator (B) with record
    Bucket_Index : Natural := 0;
    Index : Natural := 0;
  end record;
  procedure Initialize (It : in out Bag_Iterator);
  -- Overriding primitive supbrograms of the concrete actual Iterator.
  procedure Reset (It : in out Bag_Iterator);
  procedure Next (It : in out Bag_Iterator);
  function Is_Done (It : Bag_Iterator) return Boolean;
  function Current_Item (It : Bag_Iterator) return Item;
  function Current_Item (It : Bag_Iterator) return Item_Ptr;
end BC.Containers.Bags;
```

Fig. 7. Types and operations of the *BC.Containers.Bags* class that must be removed

The changes required in the package body are: removing the body of all the iterator operations and modifying the *Intersection* operation, because this operation takes into account that the elements in the hashing table can change their position when removals take place. Therefore the iterator to the next element remains the same. Figure 8 presents the new *Intersection* operation (the changes are marked with a comment IS NEW).

```
procedure Intersection (B : in out Bag'Class; O : Bag'Class) is
...
        if not Exists (O, This_Item) then
            Aux := It;                  -- IS NEW
            Next(Aux);                  -- IS NEW
            Detach (B, This_Item);
            It := Aux;                  -- IS NEW
        else
...
  end Intersection;
```

Fig. 8. New version of *Intersection* operation of the *BC.Containers.Bag*

The required changes of the *Containers.Bags.Bounded* are to create the hash table with shortcuts as items and as values (see Fig. 9) and to remove the specification of the functions *Item_At* and *Value_At* and of the type *Bounded_Bag_Iterator* in the specification package. Notice that we need to store the number of copies of an item (because a bag actually owns only one copy of each item and it counts the duplicates). We must then define a new *Node* type to store the number of occurrences together with the item. In consequence, the function *Create_Node* (introduced in Sect. 5) must be overridden. We must also define the function "=" and the function *New_Hash* to take into account the necessary indirection provoked by storing the shortcuts instead of the items.

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

  function "=" (L, R : Shortcut) return Boolean;

  type Shortcut_Ptr is access all Shortcut;

  package IC is new BC.Support.Bounded (Item => Shortcut,
                                        Item_Ptr => Shortcut_Ptr,
                                        Maximum_Size => Size);
  use IC;

  package VC is new BC.Support.Bounded (Item => Shortcut,
                                        Item_Ptr => Shortcut_Ptr,
                                        Maximum_Size => Size);
  use VC;

  type Node_Bag is new Node with
    record
      Value : Positive;
    end record;
  function Create_Node(B: Bounded_Bag) return Access_Node;

  function New_Hash(S : Shortcut) return Positive;

  package Tables is new BC.Support.Hash_Tables
     (Item => Shortcut,
      Hash => New_Hash,
      Value => Shortcut,
      Value_Ptr => Shortcut_Ptr,
      Buckets => Buckets,
      Item_Container => IC.Bnd_Node,
      Item_Container_Ptr => IC.Bnd_Node_Ref,
      Value_Container => VC.Bnd_Node,
      Value_Container_Ptr => VC.Bnd_Node_Ref);

  type Bounded_Bag is new Bag with record
    Rep : Tables.Table;
  end record;
...
end BC.Containers.Bags.Bounded;
```

Fig. 9. Changes in the package specification *BC.Containers.Bags.Bounded*

In the package body, we must add for every operation the corresponding indirection to access to the item besides of the corresponding implementation of *Create_Node*, "=" and *New_Hash*. Figure 10 shows the implementation of these three functions and, as an example of how indirections are added, the implementation of the operation *Add*. In [MF99], we present the whole new *Containers* class and the classes *Containers.Bags* and *Containers.Bags.Bounded* with all the required modifications.

```
...
package body BC.Containers.Bags.Bounded is
...
function Create_Node(B : Bounded_Bag) return Access_Node is
  A : Access_Node;
  begin
    A := new Node_Bag;
    Node_Bag(A.all).Value := 1;
    return A;
  end Create_Node;
function "=" (L, R : Shortcut) return Boolean is
begin
  return Item_Of(L) = Item_Of(R);
end "=";
function New_Hash (S : Shortcut) return Positive is
begin
  return Hash(Item_Of(S));
end New_Hash;
...
procedure Add (B:in out Bounded_Bag; I:Item; Added:out Boolean) is
  sh : Shortcut;
  A : Access_Node;
  begin
    Containers.Add(B,I);
    sh := Shortcut_To_The_Last_Item_Added(B);
    if Tables.Is_Bound(B.Rep,sh) then
      sh := Tables.Value_Of (B.Rep, sh);
      A := Node_Of(sh);
      Node_Bag(A.all).Value := Node_Bag(A.all).Value + 1;
      Added := False;
      Remove(Shortcut_To_The_Last_Item_Added(B));
      B.Last_Item_Added := sh.Position;
    else
      Tables.Bind (B.Rep, sh, sh);
      Added := True;
    end if;
  end Add;
...
end BC.Containers.Bags.Bounded;
```

Fig. 10. Implementation of the functions *Create_Node*, =, *New_Hash* and of the operation *Add* of *BC.Containers.Bags.Bounded*

In the operation *Add*, we associate first a shortcut to the item, and then we ask if it is already bound to some item in the hash table. If it is already bound, we must increment the number of occurrences using the previous shortcut associated to the item and remove the temporal one.

6 Conclusions

Component libraries play currently a crucial role on software development. The existence of a wide variety of such libraries, both general-purpose and also field-specific, is becoming essential and its importance will increase in the future. However, nowadays a twofold phenomena occurs. On the one hand, most of the existing libraries present some features that can be improved, concerning quality factors as efficiency, usability, functionality, etc. On the other hand, potential clients of the library tend to simply discard it if it does not fit completely into their context. Our point of view is that both phenomena will be corrected little by little, when maturity in the field will be reached.

Our paper tries to be a contribution to this madurity process. We have presented here a case study of reengineering a good library, the Booch one, that presents a few drawbacks. First, we have enumerated the advantages and disadvantages it presents and then we have proposed a solution to all the drawbacks. The solution is based on designing a new *Containers* base class that avoids the problems and also offers a new type that implement the concept of *shortcut*, as an alternative access path to items in the container. Shortcuts are interesting because, besides of assuring fast access time to items in the container, they are abstract (independent of the implementation of the component), persistent (movements inside the data structure do not affect them), secure (meaningless accesses are not possible) and they preserve behaviour (the new component behaves as the old one). Therefore, both existing and new software can benefit from the nice properties that present the concept of shortcut. We have also shown the few necessary changes in the *Containers* children-classes to use the new *Containers* base class. The paper focuses in the technical details to show the feasibility of the reengineering process and to make explicit the cost of this process.

The library with the modifications presented in this paper can be found in `ftp://ftp.lsi.upc.es/pub/users/jmarco/`.

We would like to point out the benefits of our approach:

- The Booch library is improved from many points of view: versatility, persistence, openess, efficiency and ease of use.
- This improvement has been made in a very comfortable way; with only few changes needed. The core data structures and algorithms are the same without any modification at all.
- The concept of shortcut is general enough to be exported to other libraries. In fact, most of the main existing component libraries offer it in many different ways [MN99,MS96]. The advantage of our proposal is that it provides a systematic way of adding shortcuts to existing libraries, instead of putting them in the library from the very beginning.
- 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 *bag_test* provided by the original Booch library without any modification.

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.

[FM00] X. Franch and J. Marco. Adding Alternative Access Paths to Abstract Data Types. To appear in *Proceedings of 2000 Information Resources Management Association International Conference (IRMA)*. To celebrate in May 2000.

[Jaz95] M. Jazayeri. Component Programming – a fresh look at software components. In *Proceedings of 5th European Software Engineering Conference (ESEC)*, volume 989 of *LNCS*, pages 457–478. Springer–Verlag, 1995.

[MF99] J. Marco and X. Franch. Reengineering the Booch Component Library (extended version). Technical Report 1999–46–R, Departament de Llenguatges i Sistemes Informàtics. Universitat Politècnica de Catalunya, 1999.

[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.

[Sit+94] M. Sitaraman (coordinator). Special Feature: Component–Based Software Using RESOLVE. *ACM Software Engineering Notes*, 19(4):21–67, October 1994.