Adding Efficient and Reliable Access Paths to the JCF
Marco, J. and Franch, X.
Research Report LSI-04-19-R

Departament de Llenguatges i Sistemes Informàtics

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Adding Efficient and Reliable Access Paths to the JCF

Jordi Marco
jmarco@lsi.upc.es

Xavier Franch
franch@lsi.upc.es

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

ABSTRACT

The Java Collections Framework (JCF) is the standard Java library for representing and manipulating collections (i.e., objects that represent a group of objects, such as sets, lists, etc.). Although JCF provides adequate functionality for many purposes, it does not offer any mechanism for accessing directly the objects stored in collections apart from the standard Java references. This absence is a crucial functionality exhibited by many other widespread Java and non-Java collection libraries. In this paper, we carry out a reengineering process on the JCF to add this kind of alternative access paths, which we give the name of shortcuts. This process relies on a framework called Shortcut-Based Framework, which has been defined as library-independent. We present this framework and then we show how it may be tailored to the specific case of the JCF. The resulting JCF with shortcuts library is fully compatible with the original one (i.e., programs using the original JCF are not required to be modified), and exhibits good behaviour with respect to efficiency, reliability and internal quality. As an additional benefit of the framework, we mention that it can be applied to other collection libraries, as we have done before with an Ada95 one.

1. INTRODUCTION

Component-based software development (CBSD) [2, 19] heavily relies on the ability of reusing components from software libraries. Component reuse provides many advantages, remarkably software production hastening, software quality improvement and software maintenance cost decrease. This is especially true when considering CBSD supported by object-oriented (OO) technology [12, 18]. Basic features such as inheritance, polymorphism and dynamic binding, and others built on top of them, such as design patterns [9], had a strong impact on this paradigm. In fact, as Meyer points out [12, Chap. 4], OO technology makes possible, for the first time, the idea of turning academic McIlroy's vision of software development [13] into a component-based industry.

Collection libraries are a particular domain of software component libraries. *Collections*, also known as *containers*[1], are objects that contain (i.e., store) other objects, usually known as *data items* or *data elements*. Examples of collections are sets, maps and lists. Collection libraries not only have been used through the years in different fields of computing (geometrical computing, mechanical engineering, artificial intelligence, etc.), but also they keep showing their usefulness in new domains, such as geographical information systems, bioinformatics and public key infrastructure, to name a few. Some representative collection libraries are the Java Collection Framework (JCF) [1], the Standard Template Library (STL) [17], the Library of Efficient Data types and Algorithms (LEDA) [16] and Booch Components (BC) [3, 4].

Different types of collections organize data elements in different ways, and thus they offer different functionalities, captured by their interfaces. Collection implementations (usually, more than one) will fulfill this functionality whilst exhibiting some non-functional characteristics with respect to criteria such as efficiency and security. Roughly speaking, and for the purposes of this paper, we distinguish among two different kinds of functionalities in a collection:

- *Core suitability*: functionalities related with its underlying abstract data type [6]. We find here mainly insertions, deletions, updates and retrievals, which take different forms and exhibit different properties in sets, maps, lists and so on.

- *General suitability*: additional functionalities offered by most of the collections of the library, such as persistency or concurrent access facilities. In this paper we are interested in one of those common facilities, namely providing both sequential and direct efficient access to collection elements. Sequential efficient access is usually implemented by a mechanism called *iterator* and allows traversing the elements stored in a collection in some order (usually just to retrieve the values of the elements). Direct efficient access is usually implemented by a mechanism called *location* or *position* and allows random access to collection elements. Locations exist mainly for efficiency purposes: when an element is inserted in a collection, the location in which it is stored is returned as a result of the insertion. This location is usually stored as part of the value of other element, which in its turn may be stored in another collection. Since the location is a direct effi-

[1]There is some divergence concerning terminology, depending on the particular programming language or library considered. For instance, JCF uses the term *collection*, whilst STL, Ada95 and the EiffelBase Library use *container*.

cient access to the element, it can be used for accessing the element in a highly efficient manner in other operations (typically, value retrieval, value modification or even element deletion).

All widespread collection libraries (in particular, the ones enumerated above) recognize the need for such a kind of alternative access method and therefore they provide some mechanisms for supporting it: iterators for both sequential and direct efficient access in STL [17]; items for direct access and the `for` macro for sequencial access in LEDA [16]; location and enumeration in JDSL [8]; etc. Unfortunately, this is not the case in the JCF [1], which only provides sequential access by means of iterators; even iterators are not as powerful as in other collections libraries. Direct efficient acces could be simulated using the *reference* Java feature as it were a location but this usage may cause some inconsistencies such as out-of-date access if the object bound to the reference is not in the collection anymore.

The purpose of this paper is to apply a reengineering process on the JCF mainly to provide it with direct efficient access and a more powerful version of its iterators, whilst also improving some other quality aspects such as usability, changeability and accuracy. The proposal is based on the use of a framework [5, 11], called *Shortcut-Based Framework* (SBF). The SBF is based on the concept of *shortcut* that we have defined to drive the design and implementation of collection libraries with both sequential and direct efficient access. Shortcuts can be defined as the encapsulation of the location of objects in collections. As items and locations do in other libraries, shortcuts provide an abstract, reliable and efficient alternative direct access to the elements stored in the collection. The definition of the SBF has been driven by the goal of exploiting OO facilities both at the design and implementation levels. As a result, we provide an implementation-independent approach based on the use of shortcuts to implement a generic collection which acts as a base class of the rest of collections of the library. Shortcuts allow to implement only once, in the base class, the most common capabilities (e.g., iterators) in a highly efficient and reliable way. The implementation of both the shortcuts and the common capabilities is decoupled from the details of the implementation of its inheritors.

The addition of shortcuts to an existing library such as JCF requires only a few implementation changes; the core data structures and algorithms can be reused. Since the addition of shortcuts does not affect the former observable behavior of the departing library, the running software applications that use the previous version of the library will not require to be modified. In other words, the new version is fully compatible with the standard one.

2. THE JAVA COLLECTION FRAMEWORK

The Java Collection Framework (JCF) [1], introduced in 1998 in the Java JDK 1.2, has become the standard collection library for Java. The design goals of JCF focus on simplicity (i.e., easy manipulation) and extensibility (i.e., easy addition of new collections) by employing the Java language features.

2.1 JCF Hierarchy

The hierarchy of JCF (see Fig. 1) is based on the definition of interfaces for collection classes. It includes:

- Two independent base interfaces, *Collection* and *Map*. In maps, objects are identified by keys.
- The interfaces of two concrete collections, *Set* and *List* which inherit the *Collection*'s interface.
- For each of these interfaces, an abstract implementation that defines a few abstract methods that are used in default implementations of other methods of the collection.
- The interfaces *SortedSet* and *SortedMap*, that specify sorted versions of *Set* and *Map* respectively.
- Some implementations appearing as leaves of the hierarchy, such as *LinkedList* and *HashMap*.

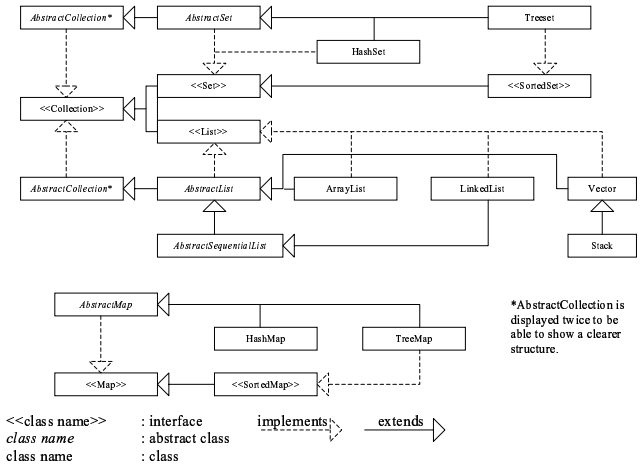


Figure 1: The JCF hierarchy of collections.

2.2 JCF Drawbacks

The intensive use of JCF by the Java community has demonstrated its usefulness but, at the same time, has shown some drawbacks that hamper some quality aspects that could be therefore improved. We present the ones have motivated our proposal using some quality factors defined on a quality model for collection libraries [7] based on the ISO/IEC 9126-1 quality standard [10].

- General Suitability: Iterators. Common iterators are just forward and cannot be combined arbitrarily with updates.
- General Suitability: Direct access. The concept of location for direct access to elements in the collection does not exist.
- Usability: Generality. JCF offers a few different collections, which may not be appropriated for a concrete scenario. This is due to the fact that JCF is a framework, not a library, therefore it only provides the core set of classes for collections. The extension of this set

which other collections more adequate to a specific scenario is left to the JCF user, but this is not always easy (see the next item).

- Changeability: Extendibility. Although JCF abstract implementations are designed to be easily extended, they have many abstract methods that must be implemented either in new collections or new implementations of existing abstract collections. Moreover, some of the non-abstract methods of existing abstract collections must be overridden in order to increase their efficiency (e.g., *get* method of the *Map* class).

- Changeability: Modularity and Internal reusability. 1) There are a few implementation steps, because intermediate nodes of the library are abstract implementations with not much code in them. Only leafs of the hierarchy are full implementations. 2) Some common capabilities (remarkably iterators) are implemented from the scratch on each abstract implementation of the collections' hierarchy and in some cases leaf classes override the methods implemented in abstract classes due to efficiency requirements (e.g., *HashMap* override *get* and other key-based methods).

- Accuracy: Accurate access by iterator. Iterators may become out-of-date in some collection classes when a new object is inserted to, or an existing one is removed from, the collection. JCF collections do not offer operations to know if an iterator is still valid or not (it is monitored via exceptions).

- Accuracy: Accurate direct access. Althought JCF does not provide this kind of access, the usage of the *reference* Java feature as location provokes out-of-date access if the object is not in the collection.

3. THE SHORTCUT-BASED FRAMEWORK

In this section we outline the main features of the *Shortcut-Based Framework* (SBF) that we have designed with the aim of solving the JCF drawbacks mentioned in the last section. This is achieved by means of the *Shortcut* concept that we propose at the core of our framework. We present the SBF in a language- and even library-independent manner, and we will show in the next section how the framework has been tailored to the specific case of the JCF.

3.1 The Shortcut Concept

Shortcuts encapsulate the feature of location or position of an object in a collection. Shortcuts provide abstract, reliable and efficient access to the elements stored in the collection. As mentioned in the introduction, most of the existing collection libraries recognize the need for such a kind of alternative access method; however, the mechanisms that they offer are *ad-hoc*, implementation-dependent and not totally reliable. In our proposal, we provide an implementation-independent approach based on the use of shortcuts to implement a generic collection which acts as a base class of the rest of concrete collections. Shortcuts allow implementing only once, in the base class, the most common capabilities (e.g., iterators) in a highly efficient and reliable way. The implementation of both the shortcuts and the common capabilities are decoupled from the details of the implementation of its inheritors. We have designed the framework with adaptability in mind, to be able to apply it to different collection libraries. Therefore, we have not fixed the details of the hierarchy (i.e., which concrete collections, and which concrete operations in them, do exist); the framework includes just its general layout.

3.2 The SBF Hierarchy

Figure 2 shows the hierarchy we have chosen for the SBF.

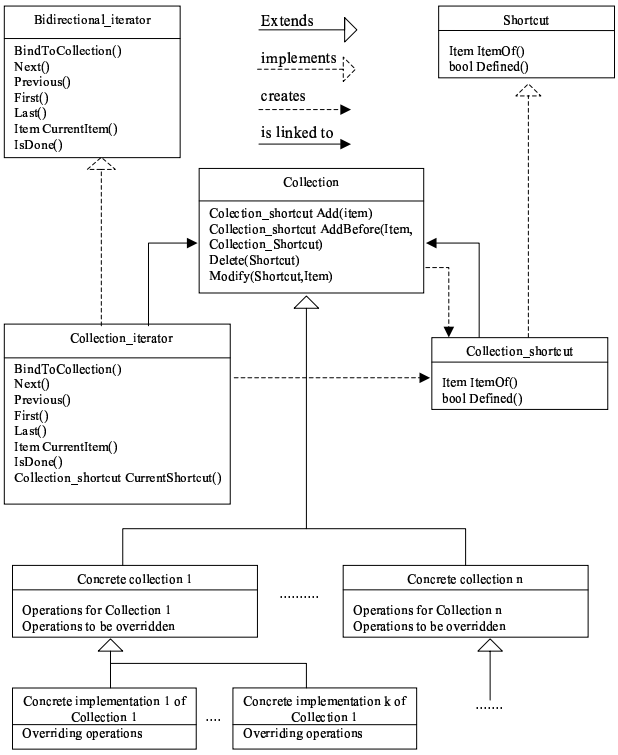


Figure 2: The Shortcut-Based Framework

- *Bidirectional_iterator*. An abstract class that provides the interface of iterators that support forward and backward traversal of a collection.

- Collection_iterator. An efficient implementation of Bidirectional_iterator enlarged with a new method that returns the shortcut bound to the current item of the iterator. This class is implemented over the base class Collection; as a consequence, it is fully independent of the specific kind of collection. All operations of this class shall have $O(1)$ as order of magnitude to guarantee highly-efficient access by iterator.

- *Shortcut*. Defines the basic interface of the concept of shortcut which encapsulates the feature of location or position of objects.

- Collection_shortcut. An efficient (i.e., $O(1)$ time for each operation) and secure implementation of the Shortcut interface. Collection_shortcut is implemented over the base class Collection; as a consequence, it is fully independent of the specific kind of collection but can be used for access to items them store.

- Collection. This base class acts as a common parent class for all kind of collections. It provides the interface and implementation of the most common capabilities of collection libraries.
- Concrete collections. Children classes of Collection that are not leaves, which represent different types of collections (list, map, etc.). Each of them adds the interface and implementation of its specific functionalities to the ones inherited from the Collection class.

  The strategy chosen to implement these classes consists on storing the items in the base class Collection and the shortcuts bound to them in a concrete implementation (an array, dynamic storage, ...; see below). The order of magnitude of the implementation is optimal: algorithms of concrete implementations do not suffer increased complexity, neither in the worst case nor in the average case, because access via shortcuts requires just constant time. In order to do this, specific operations of concrete collections are implemented using (if it is necessary) an operation implemented by their subclasses (i.e., using the Template Method design pattern [9]). Concrete collections also define as protected the interface of the deferred operations that appear as a result of applying the Template Method design pattern (that we call *concrete interface*) and implement a (in some cases non-efficient) version of them using the Collection interface and shortcuts. We want to remark that this implementation strategy uses the base class Collection as a black box and, at the same time, makes the concrete collection a black box for its children classes. Moreover, all the operations of the collection class are $O(1)$. Last but not least, each concrete collection is an implementation (non-abstract) class.
- Concrete implementations. Children classes of concrete collections that are leaves. These classes implement the concrete interface by means of data structures. They inherit all the functionalities of the concrete collection and as a consequence their implementation can be made avoiding iterators and locations. On the other hand, inherited implementations may remain if they already fulfill efficiency requirements.

3.3 Some Implementation Details

The essential point consists in maintaining an efficient mapping from shortcuts to items in the *Collection* base class. There are three possibilities to implement this mapping depending on the underlying memory management scheme: 1) dynamic storage without garbage collection; 2) dynamic storage with garbage collection; 3) static memory. In this work we use the second option (see [15] for the rest of the schemes).

On the one hand, the *Shortcut* class is implemented with a *pointer* (a reference in Java) that points to a tuple containing the object and a *deleted* flag, i.e., an attribute to record if the object is deleted or not. On the other hand, the *Collection* base class is implemented with a double linked list of these tuples in order to have efficient bidirectional iterators. Figure 3 contains a graphical representation of this scheme with a concrete implementation that stores the shortcuts in tree-form.

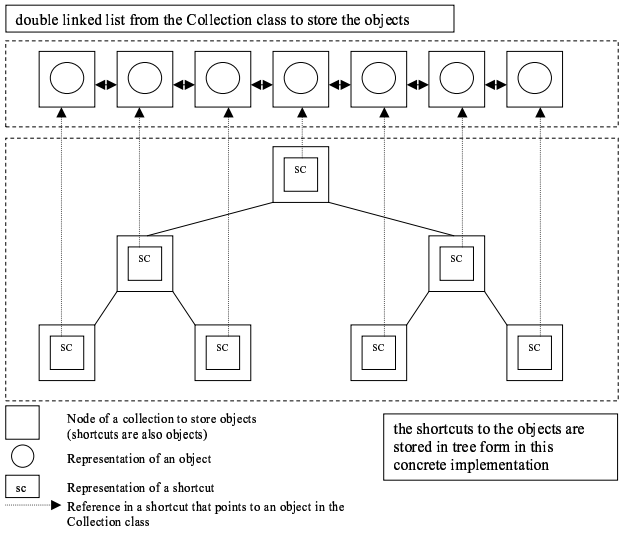


Figure 3: Implementation of the SBF

4. APPLYING THE SBF TO JCF

In this section we study the application of the library-independent SBF to the JCF. This application shall overcome some difficulties, among which we remark a key constraint in order to make our approach feasible: the enlarged JCF with shortcuts shall assure that existing software systems that use (i.e., create instances of classes of) the original JCF do not need to be modified to work with the shortcut version of the JCF, but just recompiled. This property allows to substitute the original JCF by the JCF with shortcuts without any cost. This constraint has mainly three consequences:

- The names of the classes and interfaces have to be the same as in the original JCF even in the case of abstract classes.
- The same methods (same names and functionality) have to be offered as in the original JCF.
- The hierarchy of the original JCF must be preserved in the design because of casting. For example, an existing class C that uses the ArrayList from the JCF can cast it to AbstractList, AbstractCollection and Collection. Therefore, when C is recompiled with the new JCF with shortcuts, the cast shall remain valid.

We present next the design coming from the application of the SBF, taking into account the previous points and the capabilities of Java. A hierarchy with the most representative classes of the new JCF's version is displayed in Fig. 4, together with the relationships with the SBF represented by rectangles in the figure.

Iterator(bi-directional_iterator)**:** The Iterator interface from the JCF extended with methods to make bi-directional traversal possible and some extra methods for additional functionalities. As a result, this interface contains methods to traverse a collection forward and backwards; the method currentShortcut() from the SBF; isOutOfDate() to check if

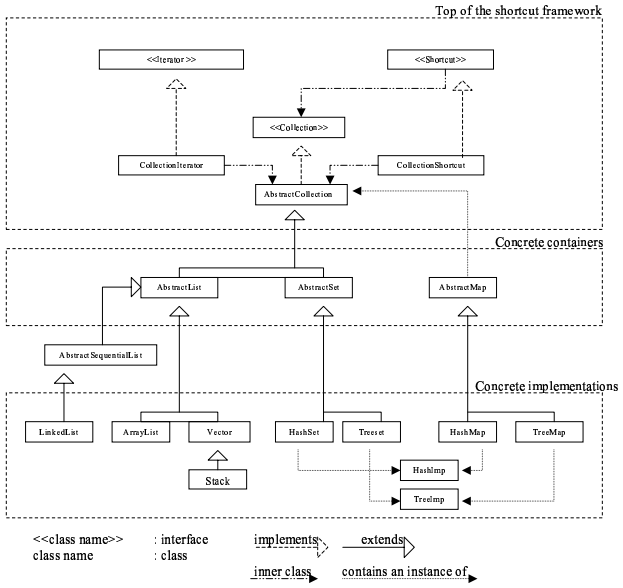


Figure 4: JCF Hierarcy after applying the SBF.

the iterator is still in valid condition or not; and two methods to reset the iterator to the first or the last item in the iteration order. Since in the JCF all the iterators appearing in the hierarchy implement the Iterator interface, they all at least offer the methods mentioned here.

Shortcut: The class as defined in the SBF. The only difference is that in this design, it becomes an inner class of *Collection* and it contains an extra method that returns the *Collection* it is linked to. The reason for making *Shortcut* an inner class is the extra method itself, because it returns a Collection and in turn there are *Collection* methods that return a Shortcut. In other words, *Collection* and *Shortcut* use each other in their method declarations.

Collection: This interface is not defined by the SBF, but it must be included to keep the JCF with shortcuts compatible with existing classes as explained above. *Collection* is based on the Collection interface from the original JCF. It contains all the method definitions from the JCF Collection and it declares the shortcut methods from the Collection class of the SBF. Besides these method declarations it contains the *Shortcut* interface. It must be noted that the JCF *Collection* interface already declares a method that adds an object, which does not allow to declare the corresponding method that adds an object and returns the shortcut, defined in the SBF's Collection class, in this interface too. For this reason, we introduce an extra method (Shortcut scToLastItemAdded()) to return a shortcut to the last object added to the collection; in other words, we have split the SBF's add method into two methods.

CollectionIterator and *CollectionShortcut*: The implementation of the *Iterator* resp. *Shortcut* interface as defined in the SBF.

AbstractCollection: This is really the SBF's Collection class. Like in the original JCF, it implements the methods declared in the *Collection* interface. Furthermore, it contains *CollectionShortcut* and *CollectionIterator* as inner classes. The name of this class is a bit confusing, since it is not really an abstract class because all its methods are implemented and it can be instantiated. The reason why the class is not renamed is compatibility, as mentioned before.

The remaining interfaces: These interfaces (Map, Set, SortedSet, etc.), not displayed in Fig. 4 for brevity, are the same interfaces as in the original JCF (Fig. 1) without changes. Furthermore, the interfaces are implemented by the same classes as in the original JCF.

AbstractSet, *AbstractList* and *AbstractMap*: These are the concrete collections from the SBF. These concrete collections are derived by extending the SBF's Collection class (*AbstractCollection* in this case) as prescribed by the SBF. An exception is *AbstractMap* because it cannot be derived directly from the base class by inheritance due to specifications in the JCF Map interface. Among others these specifications imply that both the AbstractCollection and AbstractMap (in the original JCF) contain a remove method that only differ in their return type. In Java these methods cannot override each other or be contained next to each other. To make it possible for the AbstractMap to extend the AbstractCollection, the remove method has to be changed or removed from one of these classes. But due to the compatibility constraint (all classes from the original JCF have to offer the same methods in this design), this is not possible. Therefore we have decided to contain an instance of *AbstractCollection* in *AbstractMap*. In this way, *AbstractMap* has access to the methods and classes of *AbstractCollection* and it can override these methods when necessary. These three classes are all full implementations classes like *AbstractCollection* and can all be instantiated.

AbstractSequentialList: It is an empty class in this design and it is only included for compatibility. In the JCF this abstract class mostly implements the methods that throw an UnsupportedOperationException and implements the abstract declared methods in AbstractList for their use with sequential lists. With the application of the SBF this is not necessary anymore, because AbstractList can be made a full implementation class, with implementations that are suitable for use with sequential lists too.

LinkedList, ArrayList, Vector, Stack, HashSet, TreeSet, HashMap and TreeMap: These JCF classes fall into the SBF concrete implementations category. Besides the implementation of the methods used in the concrete classes (AbstractSet, etc.), they only add the specific methods for that concrete implementation.

HashImp and *TreeImp*: In the original JCF, HashSet is implemented by using the implementation of HashMap. On the one hand, the HashSet contains an instance of a HashMap. On the other hand, for the implementation of the HashSet methods, the methods from the HashMap are used (e.g., the add method is implemented by calling the put method of the HashMap, the remove method of HashSet calls the remove method from the HashMap, etc.). With the application of the SBF, this implementation is not possible. HashSet should only add specific methods to the methods defined in

its parent classes (AbstractCollection and AbstractSet) or override the methods used for the implementation of AbstractSet; HashSet should not override the general methods defined in its parent classes (like add, remove, etc.). Therefore we have decided to add an extra class to the design that defines a hash implementation (HashImp) that is used in both HashMap and HashSet to implement their methods (specific methods or methods used to implement their parent classes). The treatment of classes for trees is the same: we have added a tree implementation (TreeImp) to the design that is used for implementing TreeSet and TreeMap. Both HashImp and TreeImp have been implemented using implementations from the HashMap and TreeMap of the original JCF respectively.

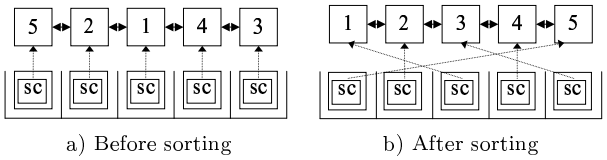
The implementation of the JCF with shortcuts[2] has been made reusing as much as possible code from the original JCF. The concrete implementations (e.g, HashMap) remain almost the same, there are, however, some slight changes to manage shortcuts when necessary. More precisely, a single operation to access to the object bound to the shortcut has been added, except for the *add* operation for which we need two operations, for compatibility as mentioned above. Regarding the additional methods and structures of the SBF, they have been implemented from the scratch.

5. GENERIC METHODS

In addition to the class hierarchy presented in Sect. 2, JCF offers also some generic methods defined in a class called *Collections*. Most of the generic methods defined by this class, like sort, search, rotate and reverse, can only be applied to lists. Therefore, to view the objects in, e.g., a HashSet in sorted order, the sort method cannot be used directly; instead, the objects of the HashSet have to be stored previously in a list (i.e., a LinkedList) or else a specific method for HashSet must be built, damaging reusability. Besides these generic methods, the Collections class also offers "wrapper" methods that convert a given collection into another type of collection. For example, it offers a method *synchronizedCollection(Collection c)* that wraps the Collection c in a wrapper class to provide synchronized access to the methods that are declared in the Collection interface.

During the reengineering process presented in this paper, the methods defined by the Collections class have been adjusted when necessary to assure their validity with the JCF with shortcuts (basically, adding the access to the object corresponding to the shortcut). But more interesting, some extra generic methods like the sort, reverse and rotate methods for lists haven been implemented to work on the AbstractCollection class. This can be considered an improvement of the original JCF: not only lists but all types of collections can be sorted, reversed, etc. (with respect to the iteration order), without previous conversion to a list.

For example, Fig. 5 a) contains a representation of a HashSet in which some Integer objects have been stored. The upper part of the figure shows the AbstractCollection in which the objects of the HashSet are stored. The bottom part shows the buckets from the HashSet in which the shortcuts to the Integer objects have been stored. Figure 5 b) displays the HashSet after the sort method of AbstractCollection has been applied on it. Notice that the sort method only sorts the linked list and does not change the order in which the shortcuts are stored in the HashSet. Since the shortcuts remain stored in the same order, nothing changes in the behaviour of the HashSet except for the iteration order. The iterator of the HashSet, as all iterators in the JCF with shortcuts, iterates using somehow the CollectionIterator, that returns the objects in the (sorted) order they are stored in AbstractCollection.



a) Before sorting b) After sorting

Figure 5: Applying the sort method on *HashSet*

6. COMPATIBILITY TESTING

For testing compatibility of the two versions, we have used a hierarchy of testing classes. The hierarchy is similar to the hierarchy of the JCF itself for making testing easier and more comprehensive. The reasons for using a hierarchy in the implementation of the tests are:

- When a test is written for a class in the JCF hierarchy, the test can and should be run on its subclasses too, because they all contain the same functionality as the parent class the test is written for.
- When a test has to be adjusted, it only has to be adjusted in the class where it is implemented; its subclasses would automatically be updated, avoiding inconsistency between the tests for the different classes.
- The test classes can use the testing utilities implemented in their parent classes.
- Copies of testing code can be avoided. In general two kinds of tests have been performed on the implementations of the design to look for bugs.

A test program was written for each concrete implementation, to perform a series of tests on its (public) methods that were defined in the original JCF for this implementation. Every test program was first compiled and run using the classes from the original JCF (importing the package that contains the original JCF). Next the program was compiled and run using the classes from the JCF with shortcuts (importing a different package). Debugging was carried out as the comparison of the results from both test, because both the new and the original version of the JCF should generate the same results (compatibility). Once we completed our tests, we found two bugs that can be considered bugs of the original JCF.

The first bug which exists in the TreeMap can be demonstrated with the following code:

[2]The implementation code is available at [21].

```
TreeMap tm = new TreeMap();
String fillArray[] ={"Barcelona","Amsterdam","Paris"};

//put the keys with a dummy value in the TreeMap
for(int i=0; i< fillArray.length; i++)
    tm.put(fillArray[i],"Dummy");

//get a view of all the key-value pairs in the TreeMap
Set s = tm.entrySet();

//get a reference to a key-value pair
Iterator it = s.iterator();
it.next();
Object aux = it.next();

//this will print Barcelona=Dummy
System.out.println(aux);

// remove the object (key-value pair)
it.remove(aux);

// check if the key-value pair is still in the TreeMap
//It should print false because we removed it before
//but it prints true
System.out.println("contains aux? " + s.contains(aux));

//The key-value pair aux references is now Paris=Dummy
//instead of Barcelona=Dummy
System.out.println(aux);
```

This bug appears because in some cases the TreeMap copies the value and key of another key-value pair to the key-value pair that is removed. In the example above, the key-value pair referenced by aux is really removed from the TreeMap but, when it was removed, the TreeMap changed this key-value pair to another key-value pair. In the JCF with shortcuts this bug does not appear because the TreeMap works with shortcuts and not directly with the key-value pairs.

The second bug appears with the use of Maps. Maps offer a possibility to retrieve the values stored in them by returning a Collection that contains the value component from the key-value pairs stored. Both HashMap and HashTree return an AbstractCollection for this purpose. However the equals method of AbstractCollection is not really implemented; it returns true if the AbstractCollections that are compared are the same instance, in other case it will return false. This means that two value collections obtained in this way can not be compared. The following code demonstrates the bug:

```
HashMap hm2,hm = new HashMap();
Collection a,b;
String fillArray[] ={"Amsterdam","Barcelona","Paris"};

//put the keys with a dummy value in the map
for(int i=0; i< fillArray.length; i++)
    hm.put(fillArray[i],new Integer(i));

//make a duplicate of the HashMap we filled
hm2 = (HashMap)hm.clone();

//get the value views from both HashMaps
a = hm.values();
b = hm2.values();

//this will return true (both HashMaps are equal
//because they are duplicates)
System.out.println("hm.equals(hm2)" + hm.equals(hm2));

//as should be with the two views, but this will
//return false (both value views are not equal)
System.out.println("a.equals(b)" + a.equals(b));
```

This bug is also solved in the JCF with shortcuts because it implements the equals method of the AbstractCollection, using the linked list in which the objects of all concrete implementations are stored. In the original JCF the bug cannot be solved in this way because it is unknown in AbstractCollection how the objects of the concrete implementations are stored.

7. CONCLUSIONS

In this paper, we have applied a reengineering process to the Java Collections Framework (JCF). This process has been carried out by tailoring a framework called Shortcut-Based Framework (SBF). The SBF defines a class hierarchy centered on the concept of shortcut, a mechanism that encapsulates the concept of position of objects. The application of the SBF on the JCF has been completed successfully and can be downloaded from [21].

We think that the main benefits of our proposal are the following:

- We have enlarged the functionality of the JCF by providing a key feature that does not appear in the original version. Although "the main design goal [of the JCF] was to produce an API that was reasonably small, both in size, and, more importantly, in conceptual weight" [20], the absence of access by position has a negative impact in so many applications that its addition becomes crucial for making JCF more usable.
- The resulting library satisfies some important properties. Firstly, it is fully compatible with the original version, which means that the programs using JCF need not to be modified when replacing JCF by JCF with shortcuts. Secondly, the internal quality of the library has been improved in a way such that extensibility becomes easier. Thirdly, shortcuts provide a secure and reliable access by position; in fact, shortcuts avoid some typical problems of access by position presents in other libraries (e.g., out-of-date shortcuts are avoided, even when objects in the collection are deleted; iterations and updates may be combined without any restriction). Last, time efficiency of the existing operations do not increase except for a small factor (shortcut management only requires one or two $O(1)$ extra assignments to be executed in the methods); our benchmarks do not show any significant lose of efficiency.
- Reengineering of the JCF has been performed by applying a well-defined, library-independent (even language-independent) framework. Due to this fact, we may apply the same process to other libraries with similar problems as we have done with the Ada95 Booch Components library [14]. In particular, we have mentioned in Sect. 4 a pair of obstacles we have overcome successfully showing the feasibility of the framework implementation.
- During the process, we have discovered two bugs on the JCF that have been fixed in the shortcuts version we are proposing. Also, we have improved the applicability of JCF generic methods.

The price to pay for the adoption of JCF with shortcuts is related to efficiency. In addition to the extra assignments for shortcut management, that we argued that is not really important, extra space for shortcut storage is required in the collections. We have computed elsewhere [15] the total extra amount needed. Trade-offs shall be analyzed to conclude whether this increment is compensated by the benefits of our proposal.

8. REFERENCES

[1] K. Arnold, J. Gosling and D. Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2000.

[2] F. Bachman et al. *Technical Concepts of Component-Based Software Engineering*. Informe del Software Engineering Institute de la Carnegie Mellon University, CMU/SEI-2000-TR-008, 2000.

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

[4] 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.

[5] L.P. Deutsch. Design reuse and frameworks in the smalltalk-80 system. *Software Reusability, Volume II. Applications and Experience*, ACM, 1989.

[6] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification, 1.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[7] Authors. A Quality Model for the Ada Standard Container Library. In *Reliable Software Technologies Ada-Europe 2003*, volume 2655 of *Lecture Notes in Computer Science*, pag. 283-296. Springer-Verlag, 2003.

[8] M.T. Goodrich, M. Handy, B. Hudson and R. Tamassia. Accessing the internal organization of data structures in the JDSL library In *Workshop on Algorithm Engineering and Experimentation (ALENEX '99)*, volume 1619 of *Lecture Notes in Computer Science*, pages 96-111. Springer-Verlag, 1999.

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

[10] ISO/IEC Standards 9126-1 Software Engineering – Product Quality – Part 1: Quality Model, June 2001.

[11] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22-35, June/July 1988.

[12] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

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

[14] Authors. 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.

[15] Authors. A Framework for Designing and Implementing the Ada Standard Container Library. To appear in *SIGAda 2003*. San Diego, California, 7-11 December 2003.

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

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

[18] O. Nierstraz and D. Tsichritzis. *Object-Oriented Software Composition*. Prentice Hall, 1996.

[19] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[20] http://java.sun.com/products/jdk/1.2/docs/guide/collections/overview.html

[21] http://www.lsi.upc.es/∼jmarco/JCFwithSBF.zip

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

Research Reports - 2004

- LSI-04-1-R : *Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations*, Rodríguez, E. and Kapur, D.
- LSI-04-2-R : *Comparison of Methods to Predict Ozone Concentration* , Orozco, J.
- LSI-04-3-R : *Towards the definition of a taxonomy for the cots product´s market* , Ayala, Claudia P.
- LSI-04-4-R : *Modelling Coalition Formation over Time for Iterative Coalition Games*, Mérida-Campos, C. and Willmott, S.
- LSI-04-5-R : *Illegal Agents? Creating Wholly Independent Autonomous Entities in Online Worlds*, Willmott, S.
- LSI-04-6-R : *An Analysis Pattern for Electronic Marketplaces*, Queralt, A. and Teniente, E.
- LSI-04-7-R : *Exploring Dopamine-Mediated Reward Processing through the Analysis of EEG-Measured Gamma-Band Brain Oscillations*, Vellido, A. and El-Deredy, W.
- LSI-04-8-R : *Studying Embedded Human EEG Dynamics Using Generative Topographic Mapping*, Vellido, A. and El-Deredy, W. and Lisboa, P.J.G.
- LSI-04-9-R : *Similarity and Dissimilarity Concepts in Machine Learning*, Orozco, J.
- LSI-04-10-R : *A Framework for the Definition of Metrics for Actor-Dependency Models*, Quer, C. and Grau, G. and Franch, X.
- LSI-04-11-R : *QM: A Tool for Building Software Quality Models*, Carvallo, J.P. and Franch, X. and Grau, G. and Quer, C.
- LSI-04-12-R : *COSTUME: A Method for Building Quality Models for Composite COTS-based Software Systems*, Carvallo, J.P. and Franch, X. and Grau, G. and Quer, C.
- LSI-04-13-R : *Enabling Collaboration in Virtual Reality Navigators*, Theoktisto, V. and Fairén, M. and Navazo, I.
- LSI-04-14-R : *DesCOTS: A Software System for Selecting COTS Components*, Carvallo, J.P. and Franch, X. and Grau, G. and Quer, C.
- LSI-04-15-R : *Evaluation and symmetrisation of alignments obtained with the Giza++ software*, Lambert, P. and Castell, N.
- LSI-04-16-R : *A note on the use of topology extensions for provoking instability in communication networks*, Blesa, M.J.
- LSI-04-17-R : *An ISO/IEC-compliant Quality Model for ER Diagrams*, Costal, D. and Franch, X.
- LSI-04-18-R : *A Case Study on Pruning General Ontologies for the Development of Conceptual Schemas* , Conesa, J.
- LSI-04-19-R : *Adding Efficient and Reliable Access Paths to the JCF*,

- LSI-04-20-R : *Exploiting Simple Corporate Memory in Iterative Coalition Games*,
- LSI-04-21-R : *On the Semantics of Operation Contracts in Conceptual Modeling* , Queralt, A. and Teniente, E.
- LSI-04-22-R : *Complexity issues on bounded restrictive H-coloring*, Díaz, J. and Serna, M. and Thilikos, D.M.
- LSI-04-23-R : *Chromatic number in random scaled sector graphs*, Díaz, J. and Sanwalani, V. and Serna, M. and Spirakis, P.
- LSI-04-24-R : *Bounds on the bisection width for random d-regular graphs*, Díaz, J. and Serna, M. and Wormald, N.C.

Hardcopies of reports can be ordered from:

Núria Sanchez
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Campus Nord, Mòdul C6
Jordi Girona Salgado, 1-3
03034 Barcelona,Spain
nurias@lsi.upc.es

See also the Departament WWW pages, http://www.lsi.upc.es/