STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVIII**, Number 4, 2013

PERFORMANCE AND RELIABILITY IN THE DEVELOPMENT OF A DECORATOR BASED COLLECTIONS FRAMEWORK

V. NICULESCU

Abstract. The paper presents some problems and their corresponding solutions encountered during the implementation process of a framework for collections defined based on features. The design is extensively based on the *Decorator* pattern in order to allow dynamic composition of the features that characterize a data structure. The *Decorator* pattern is combined with other patterns such as: *Proxy* and *Template Method*. The presentation is ruled by the different categories of features that could be added to a collection. Issues related to performance and efficiency are analyzed for each category of features.

1. Introduction

In the paper "A Decorator based Design for Collections" [5] a new design for a framework for collection data structures has been proposed.

The design is directed by the reason of creating a framework easy to use and extend. The main idea a of the framework is the following:

Anytime a feature could be added to a collection and then could be removed.

We considered a feature as being a distinctive property that characterizes the behavior of a collection – an operation or a set of operations with defined arguments, together with their semantic expressed by a clear specification. It is something that fundamentally characterizes the collection behavior [5, 4].

The design is based on the *Decorator* pattern [1], in order to allow dynamic composition of the features that characterize a data structure. The *Decorator* pattern is combined with *Proxy* pattern [1], since the features could be easily implemented by adding prefix and suffix operations that precede and succeed

Received by the editors: September 20, 2013.

2000 *Mathematics Subject Classification.* 68P05.

1998 *CR Categories and Descriptors.* E.1 [**Data**]: Data Structures; E.2 [**Data**]: Data Storage Representation .

Key words and phrases. data structures, collections frameworks, genericity, representation.

the initial operations. Also, *Template Method* pattern [1] is useful for implementing and using these operations. The design introduces an order in using the collections, but also in developing new extensions. This is achieved while the scalability is preserved.

In order to validate the emphasized advantages of this design approach, concrete implementation is needed. This paper presents the problems and their solutions, which occurred during the first iteration of the implementation process (the implementation is done in Java language). Section 2. reviews the proposed design, the following three sections present the problems encountered during the implementation process, and how their solving was directed by the performanace and reliability desiderates. The last section presents conclusions and further possible improvements that could be considered.

2. The design review

The design makes a clear distinction between *Storage Capability* and *Specialized Behavior* of each collection type [5, 3, 2].

For a collection, the storage is set by using a fundamental data structure, and the behavior is determined by the added features.

The implicit properties of each collection type are following:

- memory representation,
- iterability, and
- searchability

The interface `IStorage`, which extends `Iterable` interface, defines this contract.

In order to create a new kind of containers a linear combination of features can be used. Each feature is wrapped around the previous feature, or storage (storages could be seen as basic features).

Some of these features are *symmetric* – could be combined in any order without changing the result. Examples of this type are: `Unique` and `DeepOwnership`.

In Table 1 the considered features are presented.

Level	*Features*	*Symmetry*
4	`Ranked, Stack, Queue, PriorityOueue, Map, DMap, OMap`	no
3	`Synchronized,Unmodifiable`	no
2	`Unique, FlagDeletion, DeepOwnership, Searchable`	yes
1	`Sequence, SortedSequence, Heap, BSTree, Hashing`	no
0	all the storages types	no

Table 1. Features and their level based classification.

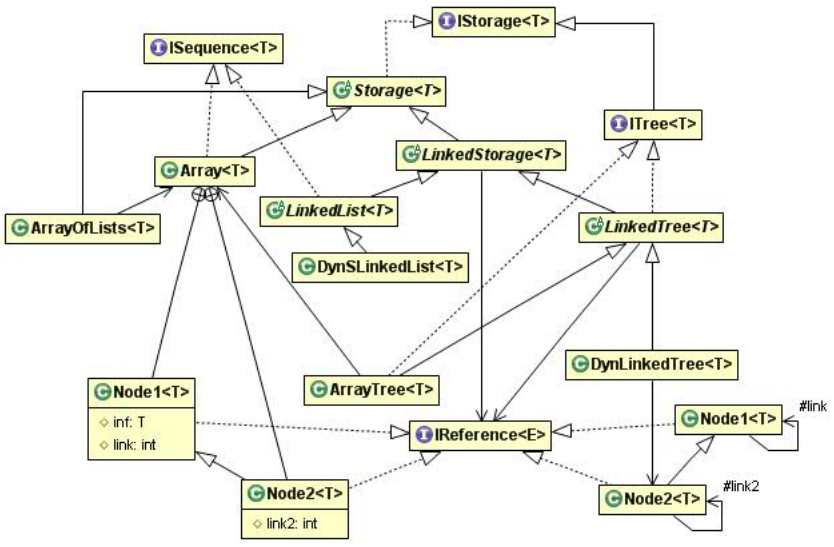

FIGURE 1. Storages classes: a subset.

Features like `Stack`, `Queues` or `PriorityQueue` have all in common the fact that they use a special rule (LIFO, FIFO, etc.) in order to extract the elements from their storage. They are specializations of a more general feature `RuleBasedExtraction`. Also, there are other similar features – we may call them *hidden* – which are not used directly by the user, but their role is to denote a common behavior of others; other examples: `Comparison`, `EmptyStorage`.

3. The Base of the Implementation

3.1. **References and Iterators.** Linked memory allocation implies the possibility to create different, sometimes complicated, structures, and so it is used by several categories of storages – linear lists, trees, etc. The linked memory representation is based on working with references, and in order to obtain a correct level of genericity (which has to be high), the implementation was based on an abstraction `IReference` that corresponds to the logical definition of a reference:

Definition 1 (Reference)**.** *A* **reference** *is considered to be any value that could be used in order to obtain another value. Examples of references are: memory addresses (pointers), indices in a table, linked nodes, etc.*

```
public interface IReference<T>{
        T getInf();
        void setInf(T e);
```

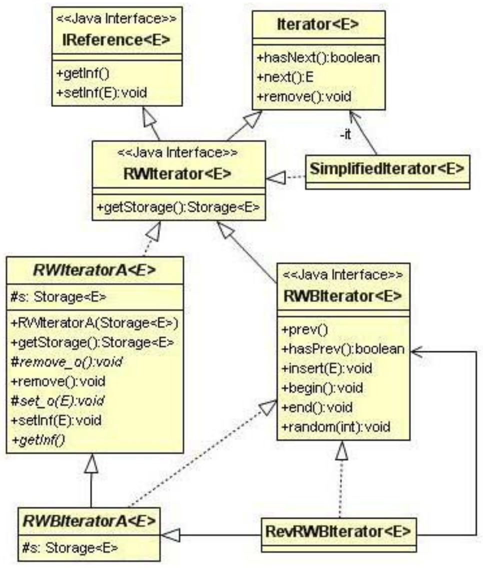

Figure 2. References and different iterator types.

}

This definition of *references* allows us to treat uniformly all the fundamental structures with linked representation: lists singly and doubly linked, trees, etc. The common behavior of linked structures is extracted into the class `LinkedStorage<T`.

In order to assure the compatibility of the framework with JFC collections, the basic Java interfaces from the package `java.util`: `Iterator` and `Iterable` are used. The class `Storage` implements `Iterable` interface, and all the iterators implements `Iterator` interface.

An important difference is given by the fact that, in the context of this framework, iterators are also references, since they correspond to the reference definition.The linked storages are based internally on `Nodes` classes which also implement `IReference`; but the linked storages interfaces use only `IReference` type. In this way, nodes and iterators could be both used instead of references – depending on the context.

Because of this, the interface `RWIterator` (which is implemented by all iterators in the framework) extends the both interfaces `Iterator` from `java.util` package, and `IReference`.

A bidirectional, reading and writing iterator is defined through the interface `RWBIterator`, and this defines operations such as `insert` and moving forward and backward.

In order to simplify the implementation of new concrete iterators, corresponding abstract classes `RWIteratorA`, `RWBIteratorA` offer partial implementations of the iterators contracts.

3.2. **Specialized Containers as Decorators.** The specializations of the containers are defined as Decorators, and the root decorator class is `SpecializedContainer<T>` that extends `Storage<T>`, but which also uses a storage of type `Storage<T>`. Each feature will be introduced as a decoration of the storage.

This class defines template methods for each independent methods of the class `Storage<T>`. These methods call the *proxy methods* that precede and succeed the calls of the `Storage<T>` methods.

For example the definition of the method `add` inside the class `SpecializedContainer<T>` is as follows:

```
public IReference<T> add(T e){
  // template method:
        e = prev_add(e); // previous action before support.add operation
        IReference<T> real_add = null;
        if (e != null)
                real_add = support.add(e); //the add operation on the support
        e = post_add(e); //// successive action after support.add operation
        return real_add;
        }
```

In this way it is assured that if the element has been added, then a not null reference on the added element is return; usually this reference is of iterator type.

When we define a container with several decorations, the proxy methods of each decoration is called in a chain. It is possible to define a container as follows:

```
Storage<Integer> s =
 new Deco1<Integer>(new Deco2<Integer>(new Array<Integer>()));
```

If we want to add an integer (for example the value 4) we have:

```
s.add(4);
```

In order to define a set we have to assure the fact that no duplicates are included into the container. `Unique` defines a decoration that assures this fact. This could be easily implemented using these proxy methods, more precisely by defining the method `prev_add` as follows:

```
public T prev_add(T e) {
                Iterator<T> it = search(e);
                if ( it==null) return e; //the element was not found so it could be added
```

```
            return null;
    }
```

A container could be modified not only directly by sending corresponding methods to it, but also through an iterator built over it. Because of this the proxy methods have to be used by the iterator operations, too.

An iterator of type `RWIterator` has the methods `remove`, and `setInf` that modify the iterated container. For changing the current element, the proxy methods used for adding could be used. In order to assure that the proxy methods are called, *TemplateMethod* design pattern is used. The abstract class `RWIteratorA` defines the template methods, and all concrete iterators of the concrete storages extend this class. For a bidirectional iterator, the method `insert` uses a similar solution based on the abstract class `RWBIteratorA`.

4. Performance and reliability when adding a new decoration

When we are implementing some features, it is possible to improve the efficiency by working directly on the base support storage.
For example `Ranked` extends the common container interface with the following methods:

- `int getRank(T e)`
- `T getElem(int index)`
- `void setElem(T e, int index)`
- `T remove(int index)`

The rank of an element is equal to its rank in the list that results by iterating the container. For a basic storage of type `Array` these operations are very fast (time-complexity equals to $(O(1))$), since we have direct access based on the index. If we base our implementation on iterators the resulted time-complexity is linear.

In order to obtain a good performance we have to use the specific methods from `Array` class if the base support is of this type. Using a method `getBaseSupport()` of the class `SpecializedContainer` we may obtain the basic support and verify its type. Also, using a similar approach we may obtain the list of all existing decorations.

The solution seems to be very simple, but it has an important problem. By extracting the base support we eliminate the features that decorates our container, and so the proxy methods `prev_add, post_add`, etc. of all the eliminated features will not be called.

We may solve this by storing the list of all features that were removed when the base support was extracted.

Based on this list we may directly call all the proxy operations.

In order to simplify and to collect this behavior, a new decoration class is defined `FeaturesStorage`. This memorizes a list of features, and redefines the proxy operations. For example, the method `prev_add` is overridden as following:

```
public T prev_add(T e){
        return direct_call_prev_add(e,features_list);
}
        protected T direct_call_prev_add(T e, Array<Storage<T>> features_list){
                        for(Storage<T> s: features_list)
                                e = s.prev_add(e);
                        return e;
                }
```

For successive proxy methods (`post`) the calls are done in reverse order.

Remarks

- The class `FeaturesStorage` could induce the impression that the infrastructure based on *Decorator* pattern is not necessary since we may just define a list of desired features and then create the corresponding kind of storage. The infrastructure based on *Decorator* pattern is important in order to achieve the proposed flexibility that allows features to be dynamically added or removed.
- "What if the `FeaturesStorage` is created twice for the same container?" Such a thing could happen when a certain feature is searched for (e.g. `Sequence`). In this case we have to take care if there is a decoration of type `FeatureStorage`, and to search inside this decoration, too. The problem could be solved by defining a union operation over the instances of the class `FeatureStorage`.
- If the order in which the decorations are added is appropriate, then in many cases `FeatureStorage` is not necessary.
- "What if the decoration that has produced a `FeatureStorage` instance is removed?" The method `removeDecoration()` that eliminates the top decoration is responsible to restore the collection state at the its state before the decoration has been added. This could be easily achieved by overriding the method correspondingly in each decoration class.

5. Sequences

The interface `ISequence` characterizes a sequence; it specifies that there is a first element, a last element, and also, that a reading, writing, bidirectional iterator could be created on a sequence; the bidirectional iterator respects the order of the elements of the sequence, and assures that, for each element, we may obtain the previous and successive elements if they exist.

If the base support is a storage of type `ISequence` then the `Sequence` feature could be added; otherwise an exception is thrown. The base sequence `seq` is extracted and all the methods that characterize a sequence are sent to this object `seq`. As for the `Ranked` feature case, the same problem related to the eliminated features appears. The solution is based again on direct calls of the proxy methods, by using an instance of type `FeaturesStorage`.

Still, there is another problem related to the bidirectional iterator. The returned iterator is the iterator on the base sequence. The iterator works directly on the base sequence, and so no proxy operations are called when `remove, insert` or `setInf` operations of the iterator are called. This could be solved by creating a proxy iterator that adds and uses a list of features when redefines the methods.

The proxy class is `FRWIteratorA<T>`, which extends the class `RWIteratorA<T>`; a simple iterator *it* of type `RWBIteratorA<T>` is wrapped inside and all the methods are redefined. For example, the implementation of the method `setInf` of the iterator is as follows:

```
public void setInf(T e){
            e = fs.prev_add(e);
            if (e != null)
                    it.set_o(e);
            fs.post_add(e);
            return ;
      }
```

(fs stores the `FeaturesStorage` object that stores all the extracted features.)

`Sequence` feature is important since other decorations as `Stack` and `Queue` are based on it. The stacks and queues are sequences with special input/output principles.

5.1. **Stacks and Queues.** These kinds of structures are sequences with special input/output principles, and so the derivation of their corresponding classes is done from `Sequence` class.

The interface `IRuleBasedExtraction` specifies two operations:

- `extract():T`
- `toBeExtracted():T`

There are several possible rules that could be used for defining the next element to be extracted:

- LIFO - Last In First Out – used by the stacks;
- FIFO -First In First Out – used by the queues;
- the element with greatest predefined priority – used by the priority queues.

Performance issue: For stacks, we may improve the performance by transforming the sequence into a reverse sequence, if necessary. A reversed sequence is obtained from a sequence by changing the order of the elements: the first becomes the last, the last becomes the first, and all the other elements are reversed. If the base storage is `Array` then it is more efficient to add and extract the elements to/from the end; if the base storage is a `LinkedList` then it is more efficient to add and extract elements in the front of the list.

The priority queue decoration could add an intermediate `Heap` decoration.

Reliability issue: Since for the stacks and queues we have special input/output principles it is necessary to modify their content only through `add` and `extract` operations. In order to assure this, the method `post_iter()` transform the iterator over this kind of structures into an iterator of the type `SimplifiedIterator` (which is only a reading iterator). Also, classical container operations such as `remove` and `search` are excluded by defining the appropriate prefix proxy operations (proxy operations for these always return `null`).

6. Collections with Comparable elements

The storages that stores comparable elements have to define a comparison method. This comparison method is either based on `Comparable` interface, or on a `Comparator` instance (from `java.util`). This behavior is defined as a decoration implemented in the class `Comparison`. This class stores a `Comparator` member – `comp` (implicitly initialized with null), and defines a protected method `compare` that uses the comparator - if such comparator is defined, or uses the natural order defined based on `Comparable` interface.

6.1. **Sorted Sequences.**

The sorted sequence decoration specializes `Comparison` decoration. First action that is induced by this decoration is to add implicitly a decoration of type `Sequence` over the support storage (if this is not already present).

For adding a new element in this kind of storage, the `insert` method of a bidirectional iterator is used. Using the iterator, we find the correct position of the new element, and then the element is inserted correspondingly.

In order to assure that the correct order of the elements, could not be affected, the iterator created on this kind of storage is a `SimplifiedIterator` that is only a reading iterator.

6.2. **Searchable Collections.**

As we have mentioned in the Section 2 we decorate a collection with `Searchable` decoration when we need a searching operation with a time-complexity better than the implicit linear one (obtained using iterators).

`Searchable` decoration class verifies and assures (by adaptation if it is the case) that the existing storage is an efficiently searchable one.

There are several possible cases:

- If there is one of the following decorations: `BSTree<T>` or `Hashing<T>`, then the storage is searchable, and nothing else should be done.
- If the base storage has the type `Array<T>` and if it has a decoration of type `SortedSequence<T>` then a binary search algorithm could be defined and used (this is a case for which the search method is overridden).
- If the base storage has the type `Array<T>`, and it doesn't have a `Comparison` decoration, then `Hashing` (based on an open addressing hash table implementation) decoration is added, as the innermost decoration.

In conclusion, `Searchable` decoration could transform, in some cases, the container, by adding additional decorations. This is an example of self-adaptation of a collection.

7. Synchronization

In order to solve the synchronized access to a data structures, we have considered in the definition of the `Storage` class, an integer field that store the number of threads that share this container as a common resource. This field is memorized for all the containers, but the space-complexity is insignificantly changed.

For a proper synchronized behavior, the decoration done with `Synchronized` class is necessary. The proxy methods are used, and they are redefined for each operation that modifies somehow the container. The `prev` operations *wait* for the monitor, and `post` operations release it.

A container could be modified also using iterators, but the methods from iterator such as `remove` or `insert` use the proxy methods `pre_remove, post_remove`, respectively `pre_add, post_add`. So, if the synchronization is done using these methods this means that the modifications through the iterators are also synchronized.

Of course, that if we work with different iterators from different threads and we modify the container, we could arrive to improper state situations. For example, one thread deletes an element which is currently pointed by an iterator from another thread; after the deletion the state of the iterator of the second thread will be in an improper state. This compromise is accepted by current implementations of the specialized synchronized variants of the collections from JCF. The user is responsible for using iterators only inside complete synchronized blocks - from the beginning of using them to the end.

We have the possibility to force this constraint in our framework by redefining the proxy methods for getting iterator: `prev_iter` and `post_iter`, in

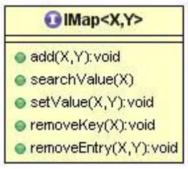

FIGURE 3. The IMap interface.

the `Synchronized` decoration class. The method `prev_iter` obtains the synchronized access, and the method `post_iter` transforms the iterator to be returned into a synchronized iterator. The class `SynchronizedIterator` is just a simple wrapper class for `RWIterator<T>`.

We have also to add a method `releaseContainer` that releases the container monitor, when the iterator is not used anymore.

Still, even in this case, it is the user responsibility to call this method, in order to let other threads to have access to the container.

A possibility to force this release would be to define for this synchronized iterator class a `finalize` method that release the container monitor. Still, the solution is not practical in Java, since the moment of finalization is not known for an object.

8. Associative collections

The main representative for associative collections are the *map* or *dictionary* collections. The interface `IMap<X, Y>`, where X represents the type of keys, and Y the type for the values (Figure 3) defines the corresponding contract.

We could create concrete map classes starting from the `SpecializedContainer<T>` in different ways (Figure 4):

(1) to replace the generic type `<T>` with a `MapEntry<X,Y>` type, and so creating containers with elements that represents pairs of keys and values; the resulted class is `Map<X,Y>` that extends the class `SpecializedContainer<MapEntry<X,Y>>` and implements `IMap<X,Y>`;
(2) to create a class that aggregates two storages: one for keys of type T (inherited from the super class), and another for the associated values of type V; the resulted class is `OMap<T,V>` that extends the class `SpecializedContainer<T>`, and implements `IMap<T, V>`;
(3) to create a map class with keys of type K and values of type T, which extends `SpecializedContainer<T>`, and implements `IMap<K,T>`. In this case a class `RefElem` which extends `MapEntry<K, IReference<T>>`

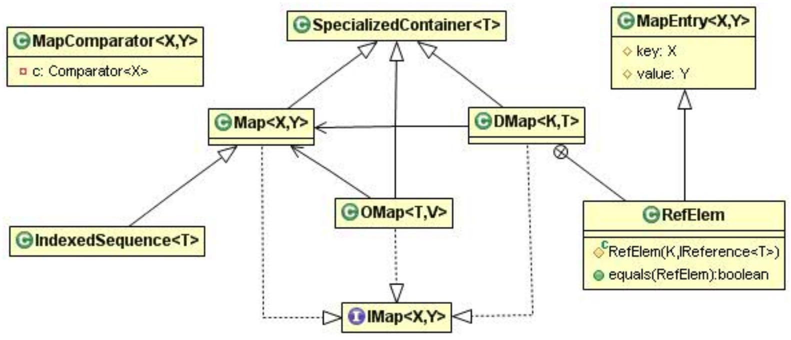

FIGURE 4. Classes that implement the `IMap` interface.

is used; this means that for each key we store a reference to the corresponding value.

```
class DMap<K, T> extends SpecializedContainer<T> implements IMap<K,T>
```

If we use the first variant for creating a map, it is required to explicitly use the type `MapEntry` (his could be considered as a disadvantage). For example:

```
Map<Integer, String> mp= new Map<Integer,String>(
 new SortedSequence<MapEntry<Integer,String>>(
     new Unique<MapEntry<Integer,String>>(
                  new DynSLinkedList<MapEntry<Integer,String>>()),
  new MapComparator<Integer,String>( new IntComparator())));
mp.add(10, "zece");
mp.add(8, "opt");
```

If we want to create a sorted map, we should provide a `MapComparator<X,Y>` which could be created based on a simple `Comparator<X>` on the keys type.

The last variant (from the enumerated possible variants) allows us to decorate an existing collection by adding to each element a key, which assures a fast access to the value. The class `DMap` aggregates an instance of type `Map<K, IReference<T>>` where the keys are stored. This instance could be created using different storages, but storages based on hash tables or binary search trees are very advantageous.

The time-complexity of adding operation is obtained as a sum of the time-complexity of the adding the value in the base support and the time-complexity of adding the key into the associated support. For the support of the keys it is important to assure a searchable decoration, and this could lead to a time-complexity for adding of $O(\log n)$. For values, any support is acceptable - so a time-complexity of $O(1)$ is easy achieved.

The difference between the last two approaches – one based on `OMap` and the other based on `DMap` – could be expressed by the following statement: in the first approach the keys are decorated with values, and in the other the values are decorated with keys.

FlagDeletion feature decorates the storage with OMap, which add boolean values to each element (which represent the keys). A false value represents the fact that the element was marked for deletion.

9. Framework usage

As we have specified before, the framework allows the creation of new collections by specifying in a proper order the characteristic features. We will give few examples.

Example 1. [Set+Tree]
If the user needs a sorted set collection, with a fast searching method, he/she may create it with the following statement:

```
IStorage<Integer> set =
        new Unique<Integer>(
                new BSTree<Integer>(
                        new DynLinkedTree<Integer>()));
```

If after a while a synchronized access it is necessary, the set could be wrapped as:

```
set = new Synchronized<Integer>(set);
```

The synchronized access could be removed by using:

```
set = ((SpecializedContainer<Integer>)set).removeDecoration();
```

Example 2. [Features of the highest level]
If the user needs a stack of integers, he/she may create it with the following statement:

```
Stack<Integer> stack =
        new Stack<Integer>(
                new Array<Integer>());
```

We may notice that the object stack should be declared as having the type Stack, not as a general IStorage.

If after using the stack for the initial purpose, based on the LIFO principle, the stack is not empty, the same object could be used as a simple collection:

```
IStorage<Integer> sup_stack= stack.removeDecoration();
```

If for elements which have been remained in the collection we would like to have a direct access based on the rank, we may add Ranked feature:

```
Ranked<Integer> rank_coll = new Ranked<Integer> (sup_stack);
```

```
for (int i =0 ; i<rank_coll.size(); i++){
        System.out.println("the next element in the stack"+ rank_coll.getElem(i));
}
```

10. Conclusions and Further Work

The presented framework is based extensively on the *Decorator* pattern in order to allow dynamic composition of the features that characterize a data structure. In this way we may add or remove a feature dynamically. Also, *Proxy* and *Template Method* design patterns ware used for defining specialized operation based on their basic variants.

Since is based on a constructive way for defining behaviors, the presented approach eliminates the need for “Fat Interfaces” that JFC builders used [7].

In order to be useful, a collection framework should not only have a nice design that could lead theoretically to important advantages, but should also be efficiently implemented, and issues as performance and reliability should be treated carefully. The implementation process of the proposed framework was leaded by these desiderates.

In order to increase the performance, the framework defines some adaptations to be done automatically in certain situations. In this way, it is not only the responsibility of the user to know, and to combine properly the possible features.

The implementation problems and their solutions, encountered in this first stage of the framework implementation, were presented here. Further improvements could be surely introduced, and it is possible to emphasize other possible automatic adjustments. Further work includes other similar improvements.

For a proper validation, testing activity is very important, too. So, the next step would treat the problem of testing, which should be based on very well analyzed and defined set of test-cases. Concrete performance comparisons with other existing similar frameworks (at least with JFC) are going to be conducted, in order to have a complete evaluation of the utility of the proposed framework.

References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1994.
[2] V. Niculescu, G. Czibula: *Fundamental Data Structures and Algorithms. An Object-Oriented Perspective*, Casa Cartii de Stiinta, 2011 (in Romanian).
[3] V. Niculescu,: *Storage Independence in Data Structures Implementation*, Studia Universitatis "Babes-Bolyai", Informatica, Special Issue, LVI(3), pp. 21-26, 2011.
[4] V. Niculescu, D. Lupsa, R. Lupsa: *Issues in Collections Framework Design.* Studia Universitatis "Babes-Bolyai", Informatica,Vol. LVII, No. 4 (Dec. 2012), pp. 30-38.

[5] V. Niculescu, D. Lupsa: *A Decorator based Design for Collections*. Studia Universitatis "Babes-Bolyai", Informatica, Special Issue (Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT2013, Cluj-Napoca (Romania), July 5-7, 2013), pp 54-64.

[6] *Generic Java*,`http://download.oracle.com/javase/1.5.0/docs/guide/language/generics.html`

[7] *Java. The Collections Framework*,`http://download.oracle.com/javase/1.5.0/docs/guide/collections/`

Department of Computer Science, Babeş-Bolyai University, Kogalniceanu 1, 400084, Cluj-Napoca, Romania

E-mail address: `vniculescu@cs.ubbcluj.ro`