STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVIII**, Number 3, 2013

A DECORATOR BASED DESIGN FOR COLLECTIONS

V. NICULESCU$^{(1)}$ AND D. LUPSA$^{(1)}$

Abstract. We propose in this paper a design for a framework dedicated to collections data structures, based on which we are able very easily to use, to adapt, to transform, and to extend the collections. A certain collection type is seen as a set of features which are added to a storage support. The design is based on Decorator together with Proxy and Template Method design patterns. This design choice allows features to be dynamically added or removed and from this, a high degree of flexibility in creating and managing the collections is achieved. The framework could be easily extended, but in an organized and reliable manner.

1. Introduction

In a previous paper [10] we have analysed the general requirements for a framework dedicated to collections data structures, based on an analysis of the related work. We propose in this paper a framework for collections data structures, in which we are able, very easily to use, to adapt, to transform, and to extend the collections. The design of the framework relies on defining collections using features, and on a design infrastructure based on *Decorator* design pattern together with others such as *Proxy* and *Template Method* [5]. A certain collection type is seen as a set of features that are added to a storage support. These features could be dynamically added or removed and this leads to a high degree of flexibility in creating and managing the collections.

Received by the editors: May 9, 2013.

2010 *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 framework, design patterns, genericity, representation.

This paper has been presented at the International Conference KEPT2013: Knowledge Engineering Principles and Techniques, organized by Babeş-Bolyai University, Cluj-Napoca, July 5-7 2013.

2. Theoretical approach

The problems that could be emphasized related to the formal and accurate defined types that characterize the data structures are related to the fact that universal and overall accepted definitions could not be found. In the literature there are different classification and definitions for the types corresponding to different containers. And because of this, the existing implemented solutions - frameworks - are also very different [10, 8].

We may start from a general definition:

Definition 1 (Collection)**.** *A* ***collection*** *– sometimes called a container –is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data.*

There are two general and important aspects related to collections [9]:

(1) *storage capability* – the elements that are grouped together have to be stored into the memory in an accessible way; usually the term *container* reflects more this aspect;
(2) *specific behavior* – the operations that are allowed for a specific type of container have different specifications; usually the term *collection* is chosen to emphasize this aspect.

The first aspect is directly connected to the data structure used for storing the elements. For storage, we may use a continuous block of memory or a set of discontinues blocks of memory (nodes) connected one to another using links (references). A linked representation may have different structures: linear, tree-like, or others.

The set of operations that could be applied to a container may be different, but also their specification may be different from one collection type to another. In order to emphasize these differences from behavior point of view, we may identify a set of features that could be applied to a container. So, our approach is not based on abstract data types, but on specific behaviors defined with features.

Definition 2 (Feature)**.** *We consider 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.*

2.1. Storage capability.

Each collection has to be stored in the memory, in a way that allows elements to be added, removed, and retrieved. The storage capability of a collection could be considered as a basic, compulsory, implicit feature. It is an implicit feature that characterizes any collection; the set of operations of this basic feature could be seen in Figure 1.

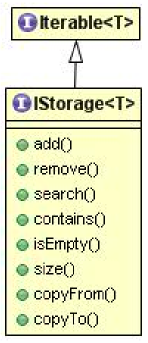

FIGURE 1. `IStorage` interface.

The formal specification that characterizes the storage capability could be given using Hoare style specifications.

-: The postcondition of the method `add(e:Element)` assures just the fact that the element e is in the storage;
-: The postcondition of the method `remove(e:Element)` assures that one instance of the element e has been removed from the storage if such an instance exists;
-: The number of the elements in the storage is returned by the method `size()`, and we may obtain an reading iterator over the elements of the storage using the method `iterator()`.
-: The methods `copyFrom()` and `copyTo()` are import/ export operations; they allow the elements of an entire other container to be added into, and also to insert all the elements into another container.

Instead of considering *Iterable* as an independent feature we have considered the existence of an iterator on the storage as being implicit. The reason of this decision is based on the fact that many features could be easily added and implemented based on iterators. If the iterability is implicitly considered then the correctness of other features definitions is assured easier. Also, there are implicit implementations of certain operations based on iterators.

In conclusion, we have considered that:

- memory representation,
- iterability, and
- searchability

are implicit properties of each collection type of our framework.

Our decision to consider all of them as basic properties is based on the fact that in the proposed framework the first level data structures (which are used for storage) are not created as a combination of their basic properties.

Two important memory representation categories have to be considered:

-: block representation,

-: linked representation.

The block representation means that a single continuous block of memory cells is used for storing the elements. There are no many options for achieving this: simple array implementation is the main choice.

A linked representation means that we may use memory locations at different addresses – nodes, and the elements could be retrieved based on using link information between these nodes. Examples of this category are the linked list and the linked trees.

2.2. **Specialized behavior – specialized containers.** Starting from a concrete storage structure we may create different collection types, by adding different behaviors.

Definition 3 (Behavior). *A* behavior *is defined as being formed of a combination of a set of basic features.*

In Table 1 the considered features are presented.

Generally, a set is characterized only by the fact there are no duplicate elements in the container. The feature `unique` defines the operation `add` with the same argument list as in the basic storage type, but changes the postcondition of the operation, by assuring the fact that the argument is added only if its value is not yet present in the container. How these elements are stored, is not a fact that characterizes the set.

`Searchable` feature certifies the fact that there is an operation for searching an element in the container with a time-complexity less than $O(n)$, which is the time-complexity of the implicit searching operation. Sorted arrays or the binary searching trees are example of searchable containers.

`Ranked` is a feature that specifies an added behavior that allows the access to the elements based on their rank. A rank of an element in a collection is equal to the rank of it in the traversal executed by the implicit iterator. And so, this could be added not only to sequences.

`FlagDeletion` allows logical deletion of the elements. This means that an element is not really removed from such a collection, but it is just logically marked for deletion. A `purge` operation will do the real deletion of all marked elements.

A container could have the `DeepOwnership` over the elements that it collects. From the implementing point of view this means that when an element is added into, a copy of it is created and this copy is stored, and when an element is deleted, it is destroyed, too.

`Sequence` assures the fact that the elements are in a particular linear order; so there are first and last elements, and each elements in between have a previous and a succesive element.

Stacks and queues specify specific behaviors, and because of that, they could be seen as features.

If the elements that are stored form pairs $(key, value)$ this means that we have an *associative* container. There are several variants to achieve this.

`Synchronized` feature assures the fact that the container could be used in multiprogramming, by several threads of execution.

Many other features could be defined, and this represents the main modality of extending the framework.

2.3. **Features classification.** The features could be classified depending on how they change the behavior of the container:

(1) features that preserve the default container operations, but changes their specifications; ex. `Unique`;
(2) features that add new operations; ex. `Ranked, Sequence`;
(3) features that restrain the set of operations; ex. `UnmodifiableStorage`;
(4) features that restrain the implicit set of operations, but add some other new operations; ex. `Stack, Queue` – they eliminate `remove`, and introduce `extract()`;

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. So, they are specializations of a more general feature `RuleBasedExtraction`.

As we have mentioned before one goal of the framework is to allow features to be added and removed dynamically. Still we may identify some restrictions; for example there are features, which could not be added after we have already added some elements into the support container. All these features are specializations of `EmptyStorage` feature.

Generally, between features we may establish specialization/generalization relationships.

In order to create new kind of containers a linear combination of features can be used. We consider that 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`.

The features that add new operations are not symmetric with the rest of the features. `Stack`, for example, add the operation `extract()` that allows the elements to be extracted based on the LIFO rule; it should be the last added feature in order to allow this operation to be accessible.

Because of these, we introduce levels for all features, based on which we will impose an order to combine the features. Table 1 presents the levels, the features of each level, and the existence or not of the symmetry property of each level.

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.

A level is *symmetric* iff all features defined inside it, could be added in a symmetric way. This means that we can add as many features we want of that level, in any order. From a *non-symmetric* level we may add only one feature; there is mutual exclusion between the features of such a level.

The feature `Unmodifiable` is applied when we want to use an existing collection only for storing and searching. We add this feature to assure that the collection state will not be changed; these kinds of collections do not need synchronization. So, the features `Unmodifiable` and `Synchronized` could belong to the same unsymmetric level.
In order to assure a proper synchronization of all the collection operations, we have considered the `Synchronized` feature in a level as high as possible. For example, `Synchronized` feature should be added after `Unique` feature, since if the verification of the existence of a value in the container would not have done in a synchronized manner then the result will probably not be correct. Still, the features that change the basic `IStorage` interface should be on the highest level.

Since we have introduced levels for each feature, we may formally define the notion of *a well defined collection*:

Definition 4 (Well-defined collection)**.** *If the collection C is defined as*

$$C = F_1 \circ F_2 \circ \cdots \circ F_n \circ S$$

where F_i are features, and S is a storage instance, then the collection C is **well-defined** *if:*

- level condition*:*

$$level(F_1) \geq level(F_2) \geq \cdots \geq level(F_n)$$

and
- mutual exclusion condition*:*

$$\forall \; level \; l \neq 2; \quad \exists! \; F_i (0 < i \leq n) \; such \; that \; level(F_i) = l$$

We may consider few concrete examples:

- `Searchable` $\circ$ `Unique` $\circ$ `SortedSequence` $\circ$ `Array` represents a searchable sorted set stored into a storage of block memory representation type – `Array`; the collection is a well-defined collection since $level$(`Searchable`) = $level$(`Unique`) = 2; $level$(`SortedSequence`) = 1; and the level 2 is symmetric.
- `Ranked` $\circ$ `Unique` $\circ$ `DeepOwnership` $\circ$ `LinkedList` represents a ranked set that is the owner of its elements, and a `LinkedList` is used for storage; the collection is a well-defined collection since $level$(`Ranked`) = 4; $level$(`Unique`) = $level$(`DeepOwnership`) = 2; and the level 2 is symmetric.

3. Framework Design

The main idea and advantage of the framework is the following:

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

Decorator design pattern [5] fits very well to this way of creating new types of containers, and this is why we have chosen to used it in framework implementation. The *Decorator* pattern is combined with *Proxy* pattern [5], since the decorations could be easily implemented by adding prefix and suffix operations that precede and succeed the initial operations. Also, *Template Method* pattern [5] is useful for implementing and using these operations.

Generally, Decorator pattern allows responsibilities to be added to an object by modifying the existing methods, not by adding methods to the object's interface. This means that in a classical usage of the pattern, the interface presented to the client must remain constant as successive layers are specified. This corresponds to symmetric features. For the highest level features the same infrastructure is used, but the new interfaces of these features are accessible if they are added as the last decoration, but also they take the benefits of the lower levels decorations.

The symmetric features modify the specifications of the basic feature operations. Prefix proxy operations could modify the preconditions, and suffix proxy operations could modify the postconditions.

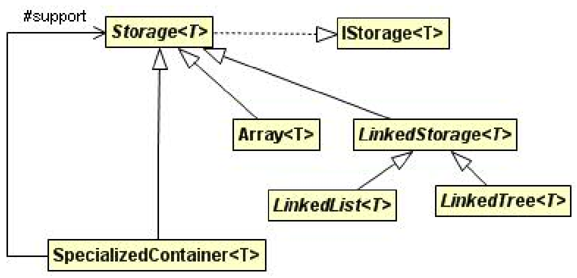

FIGURE 2. Decorator design for defining specialized collections.

A decoration that correspond to a feature that restrains the basic features implements prefix proxy operations that block the execution of the operations that have to be excluded.

3.1. **Specialized Containers.** 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 (Figure 2).

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. We have:

- `prev_add` and `post_add`,
- `prev_remove` and `post_remove`,
- `prev_search` and `post_search`,
- `prev_it` and `post_it`.

The postcondition of the method `add` assures the fact 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.

The proxy methods for adding are used also for other similar operations such as `insert` and `set` of iterators.

When we define a container with several decorations, the proxy methods of each decoration is called in a chain. The following example illustrates these calls.

Example[Proxy methods]

```
Storage<Integer> s =
          new Deco1<Integer>(new Deco2<Integer>(new Array<Integer>()));
s.add(4);
```

First, the method `s.add(4)` calls the method `prev_add` defined in `Deco1`, then the method `prev_add` from `Deco2` is called, and then (if it is the case) the element is included using the method `add` from `Array` class. After this, the methods `post_add` from `Deco2` and `Deco1` are called in this order.

The proxy methods are very important for operations specializations. The following examples emphasizes their use.

: 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` in such a way that if the element is already into the container, the add operation of the storage support is not longer called.

: Another example is for `DeepOwnership` feature that creates a copy of the element to be added, inside the own method `prev_add`. In the method `post_remove` the reference to the copy of the element could be set to null. If the framework would be ported into a language as C++, here the destructor could be explicitly called.

: In order to define synchronous access to a container we use (`Synchronized` feature). The same proxy methods are used: a `prev` method locks the storage container, and a `post` method releases it.

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.

This design of the framework allows a very flexible and dynamic adaptation: one decoration could be added and used for a while, and then could be dynamically removed.

4. Related Work

There are many others collections frameworks as well. We have analyzed in [10] some of them [4, 6, 12, 13, 14, 15, 3], and emphasies the different approaches and some general requirements.

Another related approach is connected with feature-based programming and generative programming [1, 2]. Generative programming has important advantages such as: static composition and adaptation, which lead to efficiency, and external and internal adaptations, which leads to flexibility. Still, there are also important drawbacks; such as using as composition operator only the parameterization of specific language constructs (types, classes, functions) , but the most important is the lack of dynamic composition.

Our approach embraced the same idea of using features, but we have considered only behavior features. We have imposed a delimitation of the storage aspects of a container from the behavioral aspects. Another important difference is at the design level, and it is given by the *Decorator* pattern (and the other connected design patterns). Based on this, the features could be added and removed dynamically.

5. Conclusions and Further Work

Decorator pattern has been used in order to allow the creation of a new collection based on dynamic composition of the features that characterize the corresponding data structure. In this way we may add or remove features dynamically. Also, *Proxy* design pattern could help us for defining specialized operation based on their basic variants.

The fact that only linear combinations of features are allowed could be seen as a disadvantage since the features with changed interface are visible only if they are finally added. Still, when working with a collection at one moment only one such feature is used. Because we allow features to be added and remove dynamically, linear combination is not longer a disadvantage.

Scalability is an important issue related to collections libraries, or framework. There are so many categories of data structures with so many variants, such that a classical approach would lead to an enormous number of classes to cover all. Our approach masters the incrementation of the necessary number of classes.

This approach has been directed by the reason of creating a framework easy to use and extend. The design of the framework induces an order in using the collections but also in developing new extensions. This is achieved while the scalability is preserved.

As a further work we intend to develop a concrete implementation of the proposed design and analyse the achieved usability, flexibility and performance of the resulted framework.

References

[1] D. Batory,B.J. Geraci:*Composition Validation and Subjectivity in GenVoca Generators.* IEEE Trans. Software Engineering'97.

[2] K. Czarnecki, U. Eisenecker: *Generative Programming.* Addison Wesley, 2000.

[3] J. Bloch: *The Java Tutorial. Trail: Collections* `http://docs.oracle.com/javase/tutorial/collections/`.

[4] M. Evered, G. Menger, J. L. Keedy, A. Schmolitzky: *A Useable Collection Framework for Java*, 16th IASTED Intl. Conf. on Applied Informatics, Garmisch Partenkirchen, 1998.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1994.

[6] D.R. Musser, A. Scine:*STL Tutorial and Reference Guide: C++ Programming with Standard Template Library*, Addison-Wesley, 1995.

[7] V. Niculescu: *A Uniform Analysis of Lists Based on a General Non-recursive Definition.* Studia Universitatis "Babeş-Bolyai", Informatica, Vol. LI, No. 1 pp. 91-98 (2006).

[8] V. Niculescu, G. Czibula: *Fundamental Data Structures and Algorithms. An Object-Oriented Perspective*, Casa Cărţii de Ştiinţă, 2011 (in Romanian).

[9] V. Niculescu,: *Storage Independence in Data Structures Implementation*, Studia Universitatis "Babeş-Bolyai", Informatica, Special Issue, LVI(3), pp. 21-26, 2011.
[10] V. Niculescu, D. Lupsa, R. Lupsa: *Issues in Collections Framework Design.* Studia Universitatis "Babeş-Bolyai", Informatica,Vol. LVII, No. 4 (Dec. 2012), pp. 30-38.
[11] C. Szypersky, S. Omohundro, S. Murer: *Engineering a Programming Language: The Type and Class System of Sather*, in Programming Languages and System Architectures, ed. J. Gutknecht, Springer-Verlag, pp. 208-227, 1993.
[12] P. Sestoft , N. Kokholm: *The C5 Generic Collection Library for C# and CLI* `http://www.itu.dk/research/c5/`
[13] *Fastutil: Fast & compact type-specific collections for Java*, `http://fastutil.dsi.unimi.it/`
[14] *Guava project*,`https://code.google.com/p/guava-libraries/`
[15] *YACL - Yet Another Collections Library*,`http://sourceforge.net/projects/zedlib`

(1) Babeş-Bolyai University, Department of Computer Science, Cluj-Napoca, Romania

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

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