Acta Universitatis Apulensis
ISSN: 1582-5329
http://www.uab.ro/auajournal/

No. 42/2015
pp. 67-78
doi: 10.17114/j.aua.2015.42.05

COLLECTIONS AS COMBINATIONS OF FEATURES

D. Lupsa, V. Niculescu, R.Lupsa

Abstract. We investigate in this paper an approach to collection design in which a set of features makes the distinctions between collections. Collections are defined based on combination of these features. We show how we can build new collections by decorating a storage support (a basic collection) with features. The implementation is based on decorator pattern.

2010 *Mathematics Subject Classification*: 68P05.

Keywords: data structures, collection frameworks, representation.

1. Introduction

Data structures are fundamental building blocks of algorithms and programs. They benefit by the concept of abstract data type, being defined in an accurate and formal way, while hiding implementation details. By using object oriented programming, we may define not only generic data structures by using polymorphism or templates, but also to separate definitions from implementations of data structures by using interfaces. Design patterns may move the things forward, and introduce more flexibility and reusability for data structures.

Instead of having rigid class structure, feature oriented programming allows to compose objects from individual features in a flexible way [10]. Following this new direction, we investigate the way a set features can be composed when creating objects. Its main advantage is that objects can be created just by selecting the desired features.

This paper is structured as follows: Section *Collection* provides an overview of theoretical concepts related to collection framework design. Section 3 characterizes collections by their features. Based on a decorator pattern, collection framework can be implemented on combination of these features (Section 4). This paper ends with some conclusions and suggestions for further work.

2. Collections

Collections are used to store, retrieve, manipulate, and communicate aggregate data.

There are different classifications and definitions for types corresponding to different collections. The existing implemented solutions - frameworks - are also very different. We are going to consider the next definition:

Definition 1. *A* ***collection*** *" sometimes called a container" is an object that groups multiple elements into a single unit.*

Implemented frameworks usually provide basic containers: list (vector, linked lists), bags and sets, maps, and, for some of them, the corresponding sorted collection is also available. From the design point of view, there are a number of differences, some of them stem from the language features and philosophy, some are simply design choices.

Java Collection Framework (JCF) defines a clean separation between interfaces and implementations. Interfaces are the heart and soul of the Java Collections Framework; every class implement at least one of them. Java Collections Framework major interfaces and relations between them are depicted in fig.1.

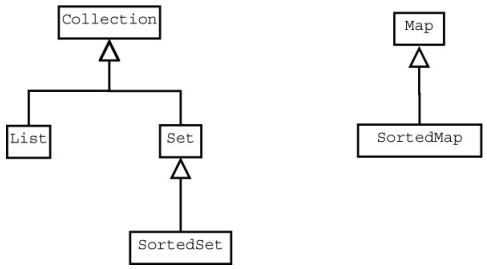

Figure 1: Java Collections Framework major interfaces

Implementations are grouped into some categories designed for different purposes. For example, general-purpose implementations are the most commonly used implementations, designed for everyday use. Special-purpose implementations are designed for use in special situations and display nonstandard performance characteristics, usage restrictions, or behavior. Abstract implementations are skeletal implementations that facilitate the construction of custom implementations. If we want to write our own implementation, reusability that comes along with abstract implementations provided by the Java platform makes it fairly easy to do this.

The Standard Template Library, or STL, is a set of C++ template classes that provides the basic algorithms and data structures of computer sciences. The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template.

In C++ STL, basic containers are grouped in:

1. Sequences: vector, deque, list
2. Associative Containers: set , multiset, map, multimap
3. Container adapters: stack, queue, priority_queue
4. and some others: string, rope, bitset

STL uses the notion of concept. Container classes are organized into a hierarchy of concepts. All containers are models of the concept Container; more refined concepts, such as Sequence and Associative Container, describe specific types of containers.

According to STL, a type conforms to a concept, or it is a model of a concept, if it satisfies all of those requirements

Types either meet a set of requirements conform to a concept, or are models of a concept. Concepts are not a part of the C++ language; there is no way to declare a concept in a program, or to declare that a particular type is a model of a concept. Nevertheless, concepts are an extremely important part of the STL. Using concepts makes it possible to write programs that cleanly separate interface from implementation. Programming in terms of concepts, rather than in terms of specific types, makes it possible to reuse software components and to combine components together.

Another way to model existing collections, is to reconsider the way they are defined. A collections framework based on set theory is Yet Another Collections Library (YACL) [11]. The project YACL consider a model in which Function extends Relation extends Set. Bags and Sequences extend Function. Hence a Function (equivalent to Sun's Map class/interface) is a type of Set. They build a theoretically sound collections library on the top of JCF Set: Set implements java.util.Set. All classes can be constructed from java.util collections and maps (where applicable).

In [6], the aim is to define a collection library for Java which uses interface multiple inheritance to offer a flexible framework for defining collection types rather than providing a complex exhaustive set of particular collection classes. They identify a small number of software engineering concepts relevant to the design of libraries of collections. They distinguish three basic orthogonal semantic properties of collections: ordering of elements, definition and handling of duplicate elements, definition of keys for efficient search. They use the next properties : order (ordered, sorted, userOrdered), duplicates (duplFree, duplIgnore, duplError) and search (searchable) that are intended to extend JCF. Particular collection types should be built by using derivation and by specifying their properties in terms of these basic types. For

example, the interface type 'Bag' can be defined as:

```
interface Bag[ELEMENT] extends Collection[ELEMENT] {}
```

and the type 'List' as:

```
interface List[ELEMENT]
extends UserOrdered[ELEMENT], Bag[ELEMENT] {}
```

2.1. Features

In software development, a feature is an increment in program development or functionality.

Feature oriented programming [10] allows you to compose objects from individual features in a flexible way. Its main advantage is that objects with individual services can be created just by selecting the desired features.

In some approaches, features are modeled as program transformations [1]. Base programs are 0-ary functions or transformations called values. Features are unary functions/transformations that elaborate (modify, extend, refine) a program by function composition. The design of a program is a named expression, for example:

```
p = f ∘ h -- program p has features f and h
```

In practice, this kind of approach needs an architecture for composing features with the required interaction handling, yielding a full object. Feature oriented programming merges the studies of feature modularity, generative programming, and compositional programming.

2.2. Our approach

Instead of a rigid class structure, we propose writing features which are composed appropriately when creating objects. Following the work in [8], [9] we define collections themselves in terms of their properties. In this work, we investigate the way we can model decorated collections over a base container that keeps pairs of elements.

A similar approach that investigates container definition and propose a new approach can be found in [11]. The difference is that we try to valorify concepts used in existing implementations. Building containers on the top of their properties is also approached in [6]. They identify some orthogonal semantic properties of collections: ordering, duplicate and search. Some of them are used as feature also in our approach (ordering and duplicate). We designed search operation corelated with iterator; a search returns an iterator to point on the searched element (following the definition of operation `find` in C++ STL). All our collections provide sequential access to any of their elements through iterators. Since all our collections are designed to be iterable (like JCF), search is available for all collections and it is not designed as a feature.

3. Collections by their features

There are two general and important aspects related to collections:

1. storage capability – the elements that are grouped together have to be stored into the memory in an accessible way;

2. specific behavior – the operations that are allowed for a specific type of container have different specifications.

The first aspect is directly connected to the data structure used for storing the elements. For example, for storage, we may use a continuous block of memory or a set of discontinuous blocks of memory (nodes) connected one to another using links (references).

To refer to specific behavior, we use *features*. Our features fundamentally characterize the collection behavior. For example, a set is characterized only by the fact there are no duplicate elements in the container. How these elements are stored, is not a fact that characterizes the set.

With features, a feature repository replaces the rigid structure of conventional class hierarchies. Following the general container concepts which are the base of C++ STL, features can define a container type (fig. 2). In this way, collections are defined by considering small number of software engineering concepts relevant to the design of libraries of collections.

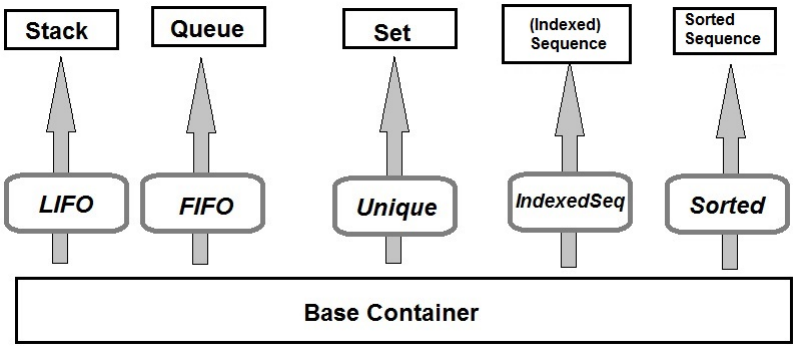

Figure 2: Features creating containers

Features can be combined in different ways, but not all of them are mutually compatible. For example, it is expected that LIFO is not compatible with FIFO, or with a sequence where items are inserted based on its index position, or with a sorted sequence. But unique could be combined with any feature listed here, as we can see in figure 3.

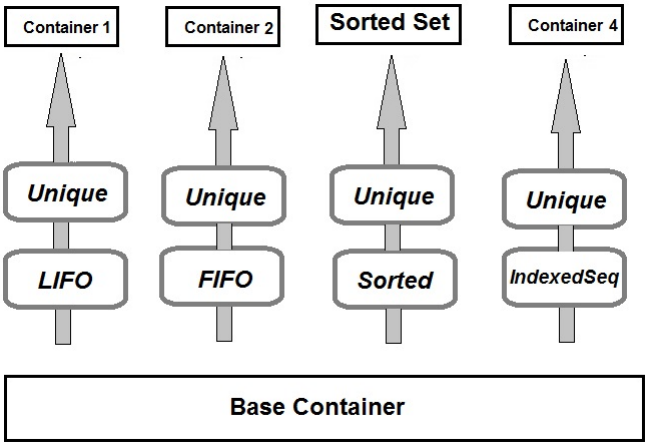

Figure 3: Features and some combinations

Features are symmetric, they can be composed in arbitrary orders. For example, the result of combining unique with sorted is the same with sorted combined with unique.

4. Features and decorations

In this approach, we view a container as being decorated with its features. The implementation is based on decorator pattern.

4.1. Features and decorator pattern

All containers have to be stored into the memory in an accessible way. A basic container should provide these capabilities. In our concrete implementation, we named it StorageSupport.

For each kind of a container, we define the creation of concrete objects via decorations. Defined decorator items are FIFO, LIFO, Sorted, IndexedSeq, Unique.

Classes that are part of this framework are depicted in figure 4.

4.1.1. Examples

Many combinations are possible. We are going to refer to some of them.

A set is a container in which a value can be there only once. It is obtained by decorating base container with Unique:

```
new Unique<...>(new StorageSupport<...>())
```

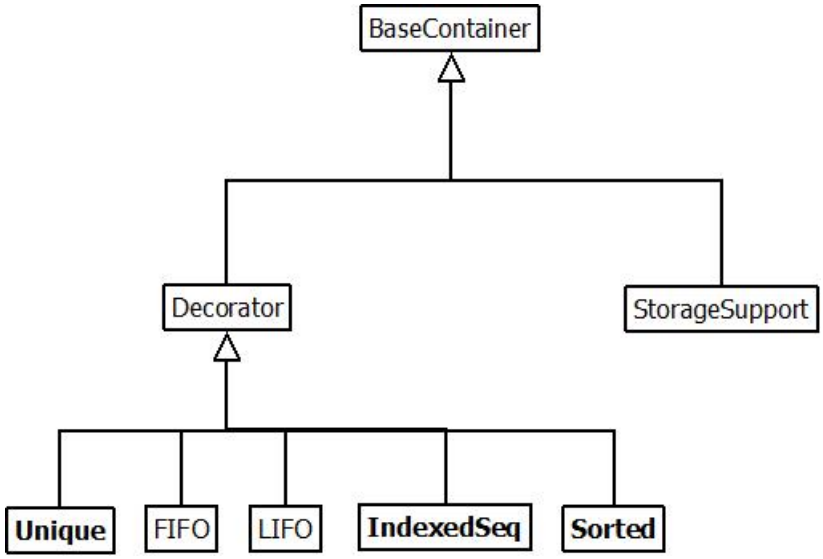

Figure 4: Classes in decorator pattern

Stacks and queues have a well defined internal stategy for adding (and removing) elements. These are called LIFO, respectively FIFO. They are obtained by decorating base container with LIFO :

```
new LIFO<...>(new StorageSupport<...>())
```

respectively FIFO:

```
new FIFO<...>(new StorageSupport<...>())
```

A sequence is an ordered collection of elements, where position of elementes are stated in the moment of inserting the elements. Indexed acces sequences use indexes to specify position of elements.

```
new IndexedSeq<...>( new StorageSupport<...>()
```

A sorted container is obtained by decorating base container with Sorted and by proving comparator function:

```
new Sorted<...>( new StorageSupport<...>(), new Comparator())
```

Combing decorations, new containers can be obtained. For example, a sorted set can be obtained by decorating base container with Unique and Sorted, in any order:

```
new Sorted<...>(
     new Unique<...>(new StorageSupport<...>()), new Comparator()
 )
```

or

```
 new Unique<...>(
      new Sorted<...>(
```

```
                new StorageSupport<...>(), new Comparator()
            )
        )
```

4.2. Implementation choices

In this section, we discuss how we implemented fetures with decorations in Java. We use the schema from fig. 4 to translate features as decorations. For each feature there is a concrete classes. Container objects are created by using feature decorations (as described in previous subsection).

An important choice for a concrete implementation is the type of the base container and all its operations. Base container is reused for any container that is generated. In our implementation, the base container, which is depicted shortly as StorageSupport in fig. 4, is a multimap. It is designed to be used in conjunction with a bidirectional, read-write iterator.

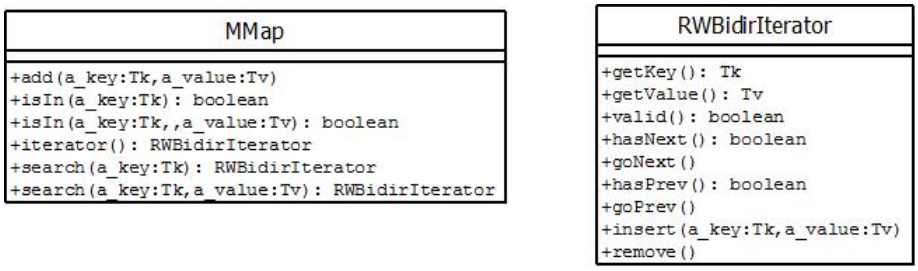

Figure 5: Base container and iterator. Operations

Base container is designed in conjunction with a bidirectional, read-write iterator. Our collections are all designed to be iterable. In our approach, iterable is a fixed property and is not added as a feature. There is always an iterator order of a container: all iterations over an unmodified container will provide elements in the same order; and modifier operations do not affect the order of the untouched items. For all the collections there is an iterator avalaible with insert and remove operations.

Operations are split between the two classes: container and iterator (fig. 5). Add is an operation that add an element on a default position, specific to container type (where appropriate), being defined by container. Operations remove and insert, that are defined as position based operations, are put in the iterator. Operation isIn (which corresponds to contains in JCF) returns a boolean value. The operation is part of the container itself. Search is also part of the container, but it returns an iterator pointing to the searched element. Our choice follows the idea from C++ STL , although there are reports that view search as a property[6] .

The disadvantage of using a multimap as a base container is that inside every container are stored pairs, not simply elements. In order to build a collection with (single) elements, we use an empty, dummy element. In order to do this, we provide an EmptyElement class, to be used as follows:

```
new StorageSupport<..., EmptyElement>()
```

That means that only the first element in the pair would be the element stored in the container. But this also mean that we carry all along two parameters (due to the restrictions of the same operation interface).

4.2.1. Implementation issues

Many decorations need to modify the basic operations of the collections. For example, operation *add* is affected by many decorations, and we will emphasize some of these situations in what it follows.

For a *Unique* decorated collection, if an element with the same value is already in the collection, the new element won't be added. The decoration verifies if the element is in the collection, then uses the back storage operation, as we can see in the following excerpt from program code:

```
public void add( Tk a_key, Tv a_value){
   RWBidirIterator<Tk, Tv> it =
        (RWBidirIterator<Tk, Tv>) backStorage().iterator();
   if (backStorage().isIn(a_key) == false) {
        backStorage().add( a_key, a_value);
   }
}
```

In the presented (java code) example, object backStorage is an instance of StorageSupport.

Within a *Sorted* decorator, first we find the (iterator based) position to be inserted, then we use iterator based insertion of the storage support to really add the element.

In order to have a first and last element, *FIFO / LIFO* decorations make use of iterator based order. For example, LIFO add the element after the (iterator based) last element. FIFO add the element before the first (iterator based) element, by using iterator based insert.

In an indexed collection, indexes use consecutive integers to specify positions of elements, and the index of an element changes over time. The *SeqIndexed* approach uses keys for indexes and values for elements. Keys are required to be Integer, and they can change with the insertion of new elements. An example is illustrated in table 1.

For user-ordered collections, sorted feature can be also used: keys and compare

Input (code sample)	Collection (iterator order)
`seq.add(1, "a");`	1 a
`seq.add(2, "b");`	2 e
`seq.add(3, "c");`	3 b
`seq.add(4, "d");`	4 c
`seq.add(2, "e");`	5 d

Table 1: Example of using IndexSeq feature

function can be used to set the user-wanted order. But this will operate differently from index feature. With sorted, no key value is modified, and key values does not have to be consecutives. In table 2 is presented an example, where a collection is built in a similar manner with indexing, by using Integer keys.

Input (code sample)	Collection (iterator order)
`seq_sort.add(3, "a");`	1 e
`seq_sort.add(4, "b");`	2 c
`seq_sort.add(2, "c");`	3 a
`seq_sort.add(25, "d");`	4 b
`seq_sort.add(1, "e");`	25 d

Table 2: Example of using Sorted feature

5. Conclusions and future directions

In the work reported here, features make the distinctions between the container types. A container is built on the top of a basic container, being decorated with features. The implementation is based on decorator pattern.

Our approach is connected with feature-based programming. Collections are defined in terms of features. Feature selection is based on the theoretical concepts used to data structure design and existing collections frameworks. C++ STL and JCF are considered. Following the work in [8], we select a small number of software engineering concepts relevant to the collections design.

Our approach benefits of the advantages of features, but without making use of feature-based generative approaches [2, 1, 3]. In some papers [10] , feature model is reported to be translated into programming languages by using inheritance and aggregation with delegation.

In [9] we considered an approach that still uses decorator pattern, but with some differences. The used storage support was in that case based on linear sequences, and this led to difficulties in implementing maps. Also, the iterators were based on the *Iterator* Java interface, and so, more specialized iterators were available only for some decorated collections.

In the future, we plan to extend our study, by considering other design patterns in this feature based approach of collection implementation. Possible choices [4], [7], would be to use adapter, bridge or abstract factory design pattern. Adapter design pattern [5] allows the conversion of the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces. Bridge design pattern [5] decouples an abstraction from its implementation so that the two can vary independently. Generally, if we have different ways of representation or storage, for a data structure, we may separate the storage from the data structure using Bridge design pattern. Abstract Factory design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

References

[1] D. Batory, B.J. Geraci, *Composition Validation and Subjectivity in GenVoca Generators*, IEEE Transactions on Software Engineering, 1997.

[2] D. Batory, S. O'Malley, *The Design and Implementation of Hierarchical Software Systems with Reusable Components*, ACM Transactions on Software Engineering and Methodology, 1992.

[3] E. Czarneck, *Generative Programming*, Addison Wesley, 2000.

[4] G. Czibula, V. Niculescu, *Fundamental Data Structures and Algorithms. An Object-Oriented Perspective*, Casa Cartii de Stiinta, 2011 (in Romanian).

[5] R. Johnson, J. Vlissides, E. Gamma, R. Helm, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995.

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

[7] V. Niculescu, *Storage Independence in Data Structures Implementation*, Studia Universitatis "Babes-Bolyai", Informatica, Special Issue, LVI(3), 2011, pp.21-26,.

[8] V. Niculescu, D. Lupsa, R. Lupsa, *Issues in Collections Framework Design*, Studia Universitatis "Babes-Bolyai", Informatica,Vol. LVII(4), 2012, pp. 30-38.

[9] V. Niculescu, D. Lupsa, *A Decorator Based Design for Collections*, Studia Universitatis "Babes-Bolyai", Informatica, Special Issue, LVIII(3), 2013, pp. 54-64.

[10] C. Prehofer, *Feature-Oriented Programming: A Fresh Look at Objects*, Springer, 1997, pp. 419-443.

[11] YACL, `http://sourceforge.net/projects/zedlib`.

Dana Lupsa, Virginia Niculescu, Radu-Lucian Lupsa
Department of Computer Science, Faculty of Mathematics and Computer Science,
Babeş-Bolyai University,
Address: 1, M. Kogalniceanu, Cluj-Napoca, Romania
email: *dana@cs.ubbcluj.ro, niculescu@cs.ubbcluj.ro, rlupsa@cs.ubbcluj.ro*