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

No. 50/2017
pp. 11-18
doi: 10.17114/j.aua.2017.50.02

FEATURES AND DESIGN PATTERNS FOR A FLEXIBLE COLLECTION FRAMEWORK

D. Lupsa, R. Lupsa

Abstract. A good design of a collections framework allows the programmers to concentrate on the important parts of their program. In this study, the focus is on designing an easy to use and to extend collection framework. In order to do this we use the properties of a container (features) combined with appropriate design patterns.

2010 *Mathematics Subject Classification*: 68P05.

Keywords: data structures, collection frameworks, representation.

1. Introduction

Modern programming languages provide collection frameworks. They usually offer many different data structure implementations, each being suitable for specific situations. Different design choices are made, some of them stem from the language features and philosophy.

By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program [11]. In order to provide extensibility and adaptability for the future the introduction of many incrementally different types is needed, but huge hierarchies are hard to understand and to use [5] and they have little to do with clients' use of the classes. A possible solution is seen [5] by placing more emphasis on the user's (i.e. the client programmer's) point of view. As stated in [11], much of the engineering effort should go into the design of the class and the type system.

Feature-oriented programming is a vision of programming in which individual features can be defined separately and then composed to build a wide variety of particular products [3]. In computer science, it was shown that objects can be composed from individual features in a flexible way [9]. Our work is under the idea that objects can be created just by selecting the desired features.

This paper is organized as follows: section 2 presents the use of features in building collection framework. Section 3 presents the features we use in this work. Section 4 investigates some implementation aspects related to features and an efficient implementation. In Section 5 we present some implementation choices as well as design patterns we use and how they integrate features. Last section present some conclusions and ideas for future work.

2. Collection Design and the Use of Features

Features can be used to build collection in feature-based generative approaches [2], [1]. In some papers [9] , feature model is reported to be translated into programming languages by using inheritance and aggregation with delegation.

Features make the distinctions between the container types. In [6] the implementation is based on decorator pattern. In order to get a specific collection by decorating it with feature, an object is built on the top of another object. For example, a sorted set can be obtained by decorating base container with *Unique* and *Sorted* features. Containers are built over a basic container.

The work in [5] identifies a small number of software engineering concepts relevant to the design of libraries of collections. They distinguish three basic orthogonal semantic properties of collections: order (ordered, sorted, userOrdered), duplicates (duplFree, duplIgnore, duplError) and search (searchable). Particular collection types should be built by specifying their properties in terms of these basic types.

Yet Another Collections Library (YACL) [10] is a collection framework based on set theory. The project YACL considers a model in which Function extends Relation extends Set. Bags and Sequences extend Function. Hence a Function (equivalent to Java Map class/interface) is a type of Set. [10] is a collections library built on the top of Set in Java collections framework.

3. Features in our work

The starting idea of this work is that the user has to specify properties he needs in terms of features. Internally the framework will decide on the data structure to be used. The design is made such that the specifications of the properties should be easily extended and in a flexible way.

A modern approach in software design is user centered design that requires designers to analyze how users would like to use a product. On one hand, developers are encouraged to use abstractions, that permits them to quickly use library functions. In this way, one can easily write code without deeply understanding implementation details. These are users that are centered on the abstract properties of containers.

There are also programmers with a strong background on data structures. Knowing what's behind an abstraction gives a lot of information that most of the time is not captured entirely on that abstraction description. For example, some users would prefer to use directly data structures that exists behind this interface.

We consider a feature as being a distinctive property that characterizes the behavior of a collection [8]. Features considered for this study are *sorted*, *ordered* and *unique*. That implies the existence of: *no order* which is the implicit choice when no *sorted* or *ordered* are chosen, an *multiple* which is the implicit choice when feature *unique* is not chosen. An enumeration class named *Feature* will collect all these features and offer names for them.

On the other hand, we can add to the list of possible features to be chosen properties referring to the data structure implementation. Linked list, array block data structure, hash, linked hash, red-black tree are considered.

Feature enumeration can look as follows:

```
enum Feature
     SORTED, ORDERED, UNIQUE
     fLinkedList, fBlock,
     fHash, fLinkedHash, fRBTree
```

Behind the enumeration, we use a mechanism to decouple interface from implementations. It decides which concrete class based on which data structure is instantiated.

If user choices are specified by using a list of features, adding new perspective (set of features) is easy to be done when coupled with an appropriate design. This way of specifying user choice can be done such that do not add much computation for any of the features the user wants to use. For example, our intention is that (and our approach provides it), when user wants to choose himself the data structure to work with, the complexity is closed to that given by working directly with classes that implements data structures.

4. Features, Design and Efficiency

Sets of features define specific collection. Each of them should be studied in order to have an efficient implementation. An implementation should consider selecting a good, appropriate storage support for a desired collection as an important issue.

After deciding over an interface and the available feature to be chosen, a mechanism to choose the appropriate support container should be decided.

We choose design patterns that can be easily extended to integrate different types of basic containers. We propose a factory based design for a data structure

framework. It integrates two ways of choosing the needed collection and can easily extended in this direction.

A **Unique** feature (no duplicate element values are allowed) influence the add operation. We will not be allowed to add a new element if an equal element already exists. That can be treated similarly for all collection and independently of their implementation.

In this approach we will use a decorator to add this functionality to collections without having to add more code to them. The decorator will be hidden in collection factory, so that the user won't have to be aware of it and not use the specific syntax needed to get the decorated class (`new Unique<...>(new ...)`).

Features **Sorted**, user **Ordered** or no order (considered by default) refers to the order in which elements in the container are iterated. These features are strongly related to the choice of data structure in order to achieve time performance. We have to carefully select the support container type in order to optimize operations.

For a non-ordered collection, a hash table can be used. In a hash, search, insert and delete is performed in O(1) on average, but (usually) in O(n) in worst case. If we are concerned about improving worst-case, we have the choice of using a sorted collection.

For ordered collection, the most used are array representations and linked representation. Array representation assumes that for each element the user know its index, and if so, it is an extremely powerful method to access elements: both easy to use and quick to access (performs in O(1)). But removal of an element is performed in O(n), and also an insertion of an element on any other position except the last. In case of our chosen operation this refers to adding an element through the iterator (next to an existing element). On the contrary, in a doubly linked list adding or removing an element next to an existing one, as well as on the first and last position, is performed in O(1). But searching an element is in O(n).

If a *Ordered* collection is requested, the factory will choose a data structure that benefit from the advantages of both. A doubled linked list with a hash defined over it would meet this goal.

An efficient implementation for a *Sorted* collection is a balanced search tree. A self-balancing binary search tree structure containing n items allows the lookup, insertion, and removal of an item in O(log n) worst-case time.

5. Implementation issues

5.1. Operations

Operations that should exists in collection (when specificity of the collection does not restrain them) are adding and removing elements and iterating over them. Variations exist for any operation named above. For example, in existing collection, we can find an operation to remove an element by given its value. And starting from here, there are also multiple choices: remove all the elements with given value (if *multiple* values are allowed) or only the first of them (if there is some kind of order existing for the container: *sorted* or *ordered*).

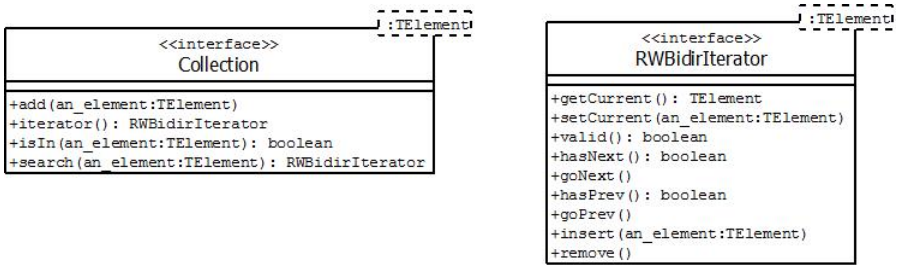

Figure 1: Collection and iterator: operations

In this implementation, we choose to consider the collection designed in conjunction with an iterator and the operations are split among them. Collections are designed to store elements (not pairs). Considered operations: names, parameters and their distribution over collection and iterator are illustrated in the figure 1. As we can see, the operation remove belongs to the iterator, and that implies that the element to be removed is identified by the current position of the iterator.

5.2. Design patterns

Our choices are conducted by the idea that a collection framework should be easy to extend by adding new data structure for data storage or by adding new view types over collection properties to be presented to the user. The importance of decoupling interface from implementations and ways to do it is presented in numerous papers ([4], [7]).

Factory pattern is used to create objects without exposing the creation logic to the client and let factory to decide which class to instantiate. It is responsible for creating a factory of related objects, that shares common interface, without specifying their classes externally. This mechanism allows us to implement a "hidden" mechanism to choose the appropriate storage support for a collection, that satisfies

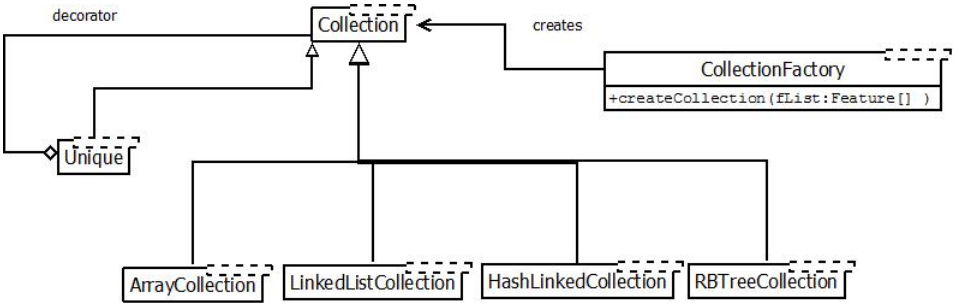

Figure 2: Class diagram

the user needs (according to the specified features).

Our approach uses a factory to build the concrete collection to be given to the user based on feature enumeration. Factory can also make decisions on the data structure to be used.

In our implementation, *CollectionFactory* has operation *createCollection* that creates a *Collection* object. It uses the information stored in feature list (an array of *Feature*s) to determine the type of object to be returned. In fact, it internally decides which data structure fits the specification given by the user. For example, for a container characterized by feature $ORDERED$, the operation *createCollection* internally decides to choose a linked hash data structure in order to optimize all the operations exposed by the interface. For a container with feature $UNIQUE$ and $ORDERED$, the *createCollection* operation will decorate the linked hash data structure with *Unique* and return the new obtained structure.

As described, another design pattern that is useful is decorator. When the uniqueness of the elements in the collections is needed, the collection is decorated with *Unique* that ensures the uniqueness of the elements in the container while the whole data structure implementation remains unchanged.

6. Conclusions and future directions

Collections can be defined in terms of features. Our approach uses features as a way to provide easy to use interface for the user.

This work presents an approach to collection framework design that uses features and factory design pattern and that can be easily extended. Factory design pattern allows us to create families of objects somehow independent of their concrete classes. Factory pattern not only that allows us to easily choose between different storage classes but it also can be used in order to easily extend this framework to integrate

other storage classes.

Starting from here, future work will include extensions that can be done by considering other data structures that can be added without modifying the way the user has to work with them, doubled by an intelligent algorithm that choses among them.

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] C. Bruno, S. Oliveira, T. Storm, A. Loh, W. Cook, *Feature-Oriented programming with object algebras*, In Proceedings of the 27th European conference on Object-Oriented Programming (ECOOP'13), 27-51, 2013.

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

[5] 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.

[6] D. Lupsa, V. Niculescu, R.Lupsa *Collections as Combinations of Features*, Acta Universitatis Apulensis, No. 42(2015), 67-78.

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

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

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

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

[11] ORACLE Java Documentation `https://docs.oracle.com/javase/8/`.

Dana 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*

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