STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVI**, Number 3, 2011

STORAGE INDEPENDENCE IN DATA STRUCTURES IMPLEMENTATION

VIRGINIA NICULESCU

Abstract. Design patterns may introduce new perspectives on the traditional subject of data structures. They introduce more flexibility and reusability in data structures implementation and use. We analyze in this paper some design patterns that can be used for data structures implementation, and their advantages. This analysis emphasizes how design patterns could be used in order to obtain implementation of the data structures based on storage independence.

1. Introduction

Data structures [4, 8] represent an old issue in the Computer Science field. By introducing the concept of *abstract data type*, data structures could be defined in a more accurate and formal way. A step forward has been done on this subject with object oriented programming [1]. Object oriented programming allows us to think in a more abstract way about data structures. Based on OOP 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.

2. First Level and Second Level Data Structures

The study of the different data structures emphasizes the fact that we can make the following classification:

- *first level* or fundamental data structures;
- *second level* data structures which are characterized by the fact that their implementations use first level data structures.

Received by the editors: March 30, 2011.

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, design patterns, genericity, representation.

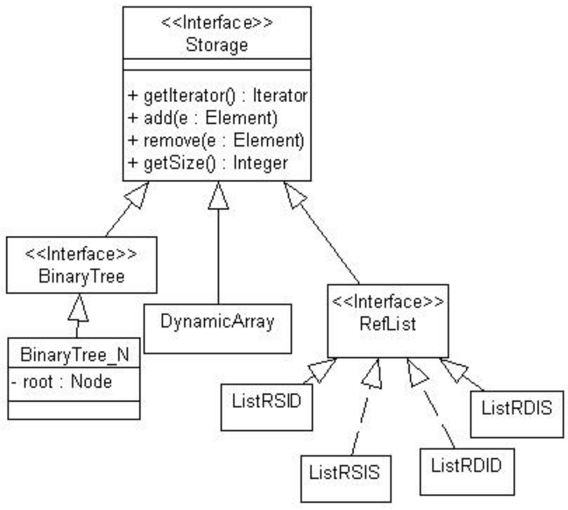

FIGURE 1. Storage interface, and some fundamental data structures - arrays, lists and trees.

The arrays, and linked representations for lists and trees are considered *first level* data structures. In order to implement a set or a map we can use an array, or a linked list or a tree; so sets and maps are examples of second level data structures. Figure 1 presents the UML class diagram for some first level data structures. We have considered dynamic arrays, lists that are implementation of an interface RefList for which the positions of elements in the list are of a reference type (singly or doubly linked, with dynamic or static allocation), and binary trees with a linked representation using nodes. (A reference is considered to be any value that is used in order to obtain another value; examples of references are: memory addresses (pointers), indices in a table, etc.)

2.1. ***Adapter.***

The problem that could arise is when we have an already developed library for fundamental data structures that is not based on this framework. In this case the *Adapter* [3] design pattern can be used.

Adapter design pattern 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.

The adaptation has to be done in a way that minimize the time-complexities of the implementations of adapted methods. For example, in order to adapt a linked list to be used as a simple storage we may define the method add from

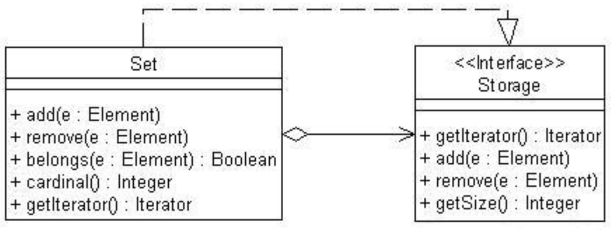

FIGURE 2. Building sets using generic storage for representation.

the Storage interface by using the method addFirst from the list implementation which has a time complexity of $\Theta(1)$.

2.2. ***Bridge.*** In order to implement a second level data structure we have to start from the corresponding abstract data type corresponding to which we may define an interface, and then based on the possible representations to implement concrete classes. The process could be simplified by using the *Bridge* design pattern.

Bridge design pattern 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.

We may consider the case of *Set* data structure. The advantages of this separation is that we will have only one class *Set*, and we may specify when we instantiate this class what kind of storage we want, for a particular situation.

This solution for implementing sets uses a reference to a general storage. The diagram of the Figure 2 presents the details of this solution. Set – the new created data structure – could also be seen as a storage that can be used in other contexts, and because of this the class Set implements the interface Storage too.

Since the class *Set* uses a storage in order to store its elements, the constructor of the class Set have to be able to initialize this storage. A direct and simple solution –but not the best – would be to give to this constructor a parameter (of type storage) that could initialize the storage.

The specific operations of the Set data structure are implemented based on the operations of the storage.

Other examples may be considered for multi-sets, maps, dictionaries, a.s.o.

Important restrictions related to the storage are that initially it has to be empty, and also it has to be an unshared storage.

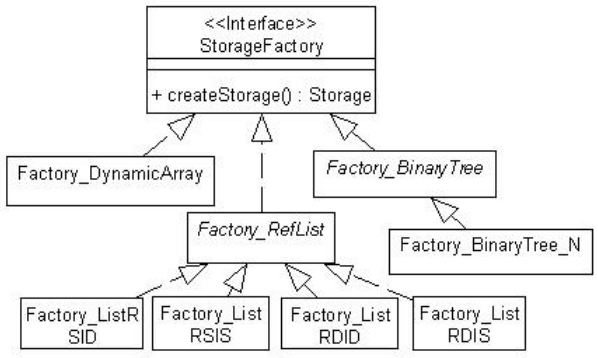

Figure 3. Factories for creating arrays, lists, and binary trees.

Generally, the data structures are used in very many different contexts and because of this it becomes important to allow their creation in a flexible and dynamical way.

These requirements could be achieved by using creational design patterns.

More precisely, we will use *Abstract Factory* to create each special storage dynamically.

Singleton design pattern assures the fact we have only one instance created for a certain type, and we have a global access point to it. Since we don't need more than one instance of a specific factory class, *Singleton* design pattern will be used for each.

Abstract Factory design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes [3].

The concrete products, in which we are interested in, are the fundamental data structures. This means that we define factories for each type of these products; the method createStorage() returns an empty storage of a specialized type. The Figure 3 emphasizes this.

The solution for implementing sets using *Bridge* but also *Abstract Factory* and *Singleton* is presented in the Figure 4.

3. Specialized Storages

An evaluation has to be done relative to the efficiency of the implementation of the operations add, and remove. For a general storage, the post-condition of the operation add specified just the fact that the parameter (of type Element) exists in the storage. In a similar way the remove operation assure the fact that one instance equal to the parameter has been removed from the storage. The implementation of the operation belongs of the class

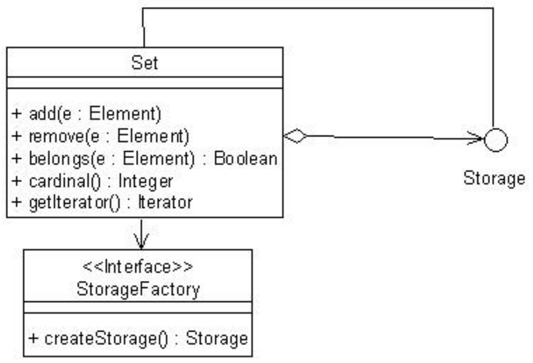

FIGURE 4. Building sets using factories for creating different storage for representation.

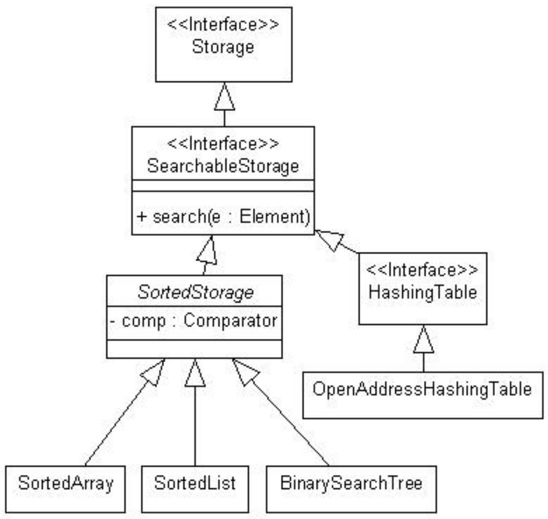

FIGURE 5. Different types of *Storages* and the relations between them.

Set is based on using the iterator – that implies a time-complexity linear in the size of the storage. In order to improve this, a specialized storage has to be defined – SearchableStorage, which add a new method search. A searchable storage is a storage that is able to implement a searching operation with a time-complexity better than for a sequential search. Examples of searchable storages are a hash table, and different types of sorted data structures.

Another useful specialization of a storage is SortedStorage. This kind of storage is useful for the implementation of sorted data structures where elements are compared using an instance of type Comparator. *Comparator* is another design pattern which is very much used in relation with data structure implementation. A SortedStorage is a specialization of a SearchableStorage since for sorted data structures we can define efficient searching operations. The Figure 5 presents the relation between these types of storages. The specification of the interface SortedStorage enforces the postcondition of the method getIterator by imposing the condition that the order in which the elements are iterated is based on comparison criteria specified by the comparator.

4. Conclusions

By separating the concrete representation of a data structure by the behaviour of its type we introduce a new level of indirection and so a new level of abstraction. Using this, we are able to implement data structures based on different fundamental data structures without creating more than one class. So, a new level of genericity is introduced, too.

Storage interfaces also introduce a classification between data structures. We emphasize the fact that each data structure could be used as storage for another data structure or as a generic storage in a generic program.

As it is already known, design patterns are very important mechanism for increasing the level of abstraction in programming. In order to achieve storage independence we have used design patterns as *Abstract Factory, Singleton, Bridge, Comparator* and *Adapter*.

Acknoledgement: This work was supported by CNCSIS - UEFISCDI, project number PNII - IDEI 2286/2008.

References

[1] Bruno R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*, Wiley Computer Publishing, 1999.
[2] H.E. Eriksson, M. Penker, *UML Toolkit.* Wiley Computer Publishing, 1997.
[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995.
[4] E. Horowitz. *Fundamentals of Data Structures in C++.* Computer Science Press, 1995.
[5] D. Nguyen. *Design Patterns for Data Structures.* SIGCSE Bulletin, 30, 1, 1998, 336-340.
[6] V. Niculescu, *Teaching about Creational Design Patterns*, Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts, ECOOP'2003, Germany, July 21-25, 2003.
[7] V. Niculescu. *On Choosing Between Templates and Polymorphic Types. Case-study.*, Proceedings of "Zilele Academice Clujene", Cluj-Napoca, June 2003, pp.71-78.
[8] D.M. Mount. *Data Structures*, University of Maryland, 1993.

Department of Computer Science, Babeş-Bolyai University, Cluj-Napoca
E-mail address: vniculescu@cs.ubbcluj.ro