Structural Design Patterns Used in Data Structures Implementation

Niculescu Virginia
Department of Computer Science
Babeş-Bolyai University, Cluj-Napoca
email address: vniculescu@cs.ubbcluj.ro

November, 2005

Abstract

Design patterns may introduce a new perspective on the traditional subject of data structures. They introduce more flexibility and reusability in data structures implementation and use. This analysis shows that design patterns could bring important advantages in basic fields of Computer Science, too. Beside the advantages brought for data structures, presenting them in this perspective represents also a simple and efficient modality to introduce design patterns in early courses of computer science. We analyze in this paper structural design patterns that can be used for data structures implementation, and their advantages. Examples that illustrate the impact of the design patterns in this context are presented.

Keywords: data structures, design patterns, genericity.

1 Introduction

Understanding design patterns is an important aspect of modern software development. Frequently, design patterns are introduced late into an undergraduate curriculum. We believe that design patterns deserve a more ubiquitous role in the undergraduate curriculum. In particular, we have found that with slight modifications and simplifications we are able to effectively introduce specific design patterns in our first programming course. Further, the necessary modifications and/or simplifications do not interfere with the primary message of each design pattern but rather highlight it.

This is the second article from a series of three that analyze the advantages of using design patterns in data structures implementation. Structural design patterns are treated in this article.

For generic data structures, templates have been used intensively, STL [9] library being the most known and used one. There are several design patterns which are used in this library, such as *Iterator* pattern which was presented in the previous article, but also *Adapter* pattern which is going to be explained here, too. Modern libraries of data structures are based on design patterns, so their understanding is now important also at the lower level classes.

Any kind of container is formed by a number of elements which are usually of the same type. A specific container has properties and behavior which are not dependent on the type of its constitutive elements. A container has a capacity, it can be full or empty, and objects can be inserted and withdrawn. A searchable container is a container that supports efficiently search operations. For each type of container – list, vector, stack, queue, etc. – we may define a corresponding abstract data type, and so, at the implementation a class (only one for each type of container) should be delivered. A very common solution is based on templates or parametric data types: the parameter is the type of the elements stored in the container. The problem is that we impose by this to use only languages where the mechanism of creating templates is included. In C++ we have such a mechanism which allows us not to create a single class, but to specify only once the pattern for creation of some classes that are different only by the type of

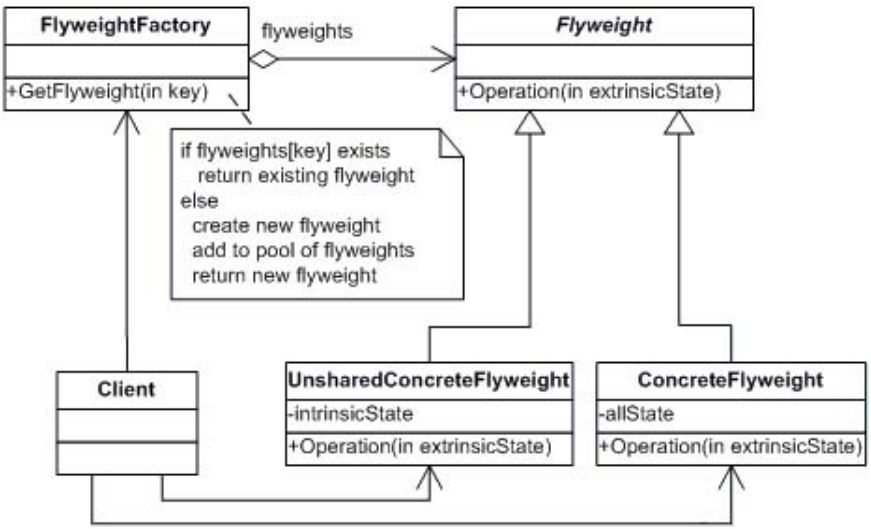

Figure 1: *FlyWeight* design pattern.

some parameters. The mechanism which was included in Java since JDK 1.5 is better since just one class is created for each container. Still, we may use a more general solution which is based on polymorphism. We may consider the type of the elements as being a superclass `ElementType` and all the specific types that we want to use to be subclasses. The solution is convenient in object-oriented languages based on a single class hierarchy, in which case the root class is considered for tt ElementType.

2 Structural Design Patterns Used for Data Structures Implementation

Structural patterns are concerned with how classes and objects are composed to form larger structures; the class form of the *Adapter* design pattern is an example. Structural *class* patterns use inheritance to compose interface or implementations. Structural object patterns describe ways to compose objects to realize new functionality; an example is *Composite* design pattern.

UML diagrams [2] are used in order to facilitate the examples understanding. Other examples could be also analyzed.

2.1 *FlyWeight*

FlyWeight design pattern uses sharing to support large numbers of fine-grained objects efficiently.

The classes and/or objects participating in this pattern are:

- *Flyweight* declares an interface through which flyweights can receive and act on extrinsic state.
- *ConcreteFlyweight* implements the *Flyweight* interface and adds storage for intrinsic state, if any. A *ConcreteFlyweight* object must be sharable. Any state it stores must be intrinsic, that is, it must be independent of the *ConcreteFlyweight* object's context.
- *UnsharedConcreteFlyweight.* Not all *Flyweight* subclasses need to be shared. The *Flyweight* interface enables sharing, but it doesn't enforce it. It is common for *UnsharedConcreteFlyweight* objects to have *ConcreteFlyweight* objects as children at some level in the flyweight object structure.

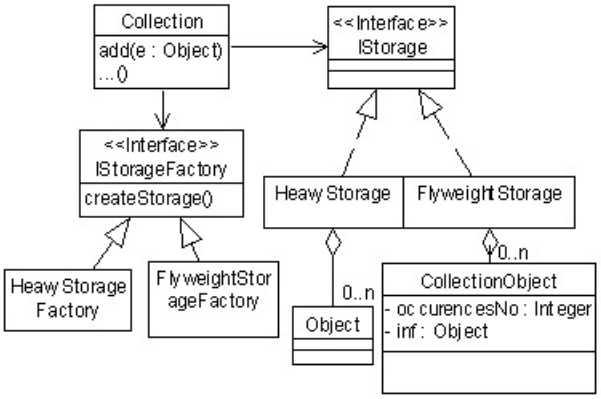

Figure 2: An implementation for collections based on *FlyWeight*, *Bridge*, and *Abstract Factory*.

- *FlyweightFactory* creates and manages flyweight objects and ensures that flyweight are shared properly. When a client requests a flyweight, the *FlyweightFactory* objects supplies an existing instance or creates one, if none exists.
- *Client* maintains a reference to flyweight(s), and computes or stores the extrinsic state of flyweight(s).

We may apply this idea when we implement a collection. If the objects which are stored are "heavy" – need a lot of memory to be stored – and also there are many replication of the same object, we may use a flyweight representation for the collection. This means that if there are many occurrences of the same element, it is stored only once and the number of occurrences is also stored (Figure 2). This representation of collection is a common representation, but it also represents a good and very simple example of applying *FlyWeight* design pattern.

When we extract an object from a flyweight collection, we have two possibilities: either in the collection there is only one such element in which case we return that object, or we have the case when we have many instances of that object, in which case we return a copy of the stored object and decrease the corresponding number of occurences.

2.2 *Bridge*

Bridge design pattern decouples an abstraction from its implementation so that the two can vary independently

The classes and/or objects participating in this pattern are:

- *Abstraction* defines the abstraction's interface, and maintains a reference to an object of type *Implementor*.
- *RefinedAbstraction* extends the interface defined by *Abstraction*.
- *Implementor* defines the interface for implementation classes. This interface doesn't have to correspond exactly to *Abstraction*'s interface; in fact the two interfaces can be quite different. Typically the *Implementator* interface provides only primitive operations, and *Abstraction* defines higher-level operations based on these primitives.
- *ConcreteImplementor* implements the *Implementor* interface and defines its concrete implementation.

Based on this design pattern we may achieve the independence of representation for a collection – so, to separate the elements storage from the abstraction (the interface of a particular collection). For

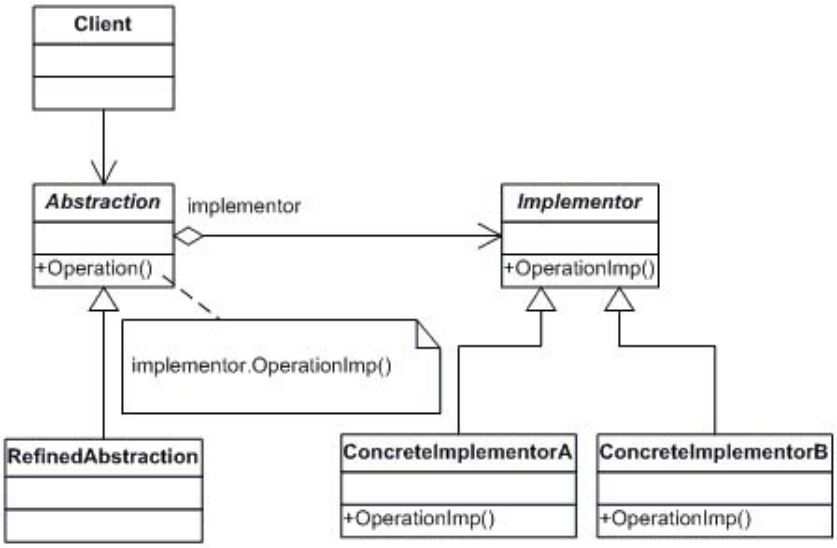

Figure 3: *Bridge* design pattern.

example we may allow two representations (storages) for the collection: a "heavy" one which stores all the objects sequentially, and a "flyweight" one which stores an object only once even if there are many occurrences of that object in the collection. For the last variant, each object is stored together with the number of its occurrences. Also, we want to choose the representation dynamically, based on which type of objects we intend to store; so *Abstract Factory* design pattern will be used for creating the storage of the collection (we will discuss in more detail about this pattern in the next article).

Generally, if we have different ways of representation, or storage, for a data structure, we may separate the storage from the data structure (*Bridge* design pattern) and use *Abstract Factory* to create a special storage dynamically. Another example could be consider for implementing *sets*: their storage could be based on linked lists, vectors, trees, etc. 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.

2.3 *Adapter*

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 classes and/or objects participating in this pattern are:

- *Target* defines the domain-specific interface that *Client* uses.
- *Adapter* adapts the interface *Adaptee* to the *Target* interface.
- *Adaptee* defines an existing interface that needs adapting.
- *Client* collaborates with objects conforming to the *Target* interface.

One of the disadvantages of using polymorphic collection classes (not parameterized collection) is that the collections store arbitrary objects, which means, in particular, that compile-time type checking cannot be performed on the collections to ensure that the objects they contain belong only to a desired subclass. For example, a `Vector` that is intended to be used only to contain `String`s could accidentally have an Integer added to it. Furthermore, the code for extracting the `String`s from such a `Vector` involves typecasting, which can make the code quite inelegant.

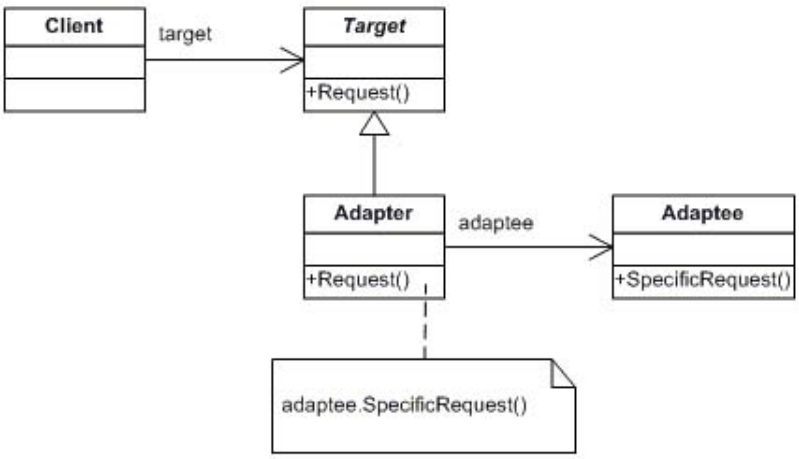

Figure 4: *Adapter* design pattern.

A solution to this problem is to implement `StringVector` class that is similar to a `Vector` but contains only `String`s. In this new class, the methods' signatures and return values would refer to `String`s instead of `Object`s. For example, the "add" method will take a String as its parameter and the "get" method will have String as the return type.

The natural way to implement the `StringVector` class is by using the *Adapter* pattern. That is, a StringVector object contains a reference to a Vector in which the Strings are actually stored. In this way, the interface is adapting the Vector class to the desired `StringVector`'s interface.

Adapter design pattern could be also used in order to adapt a general list to be a stack (or a queue). A stack follows the principle: "First-In First-Out", and the opertions with stacks are: push, pop, and verification operations (isEmpty, isFull). The operation of the stack will be implemented based on the list operations. So, a list is adapted to be a stack.

2.4 *Decorator*

Decorator pattern is used to attach additional responsibilities to an object dynamically. We can add extra attributes or "decoration" to objects with a certain interface. The use of decorator is motivated by the need of some algorithms and data structures to add extra variables or temporary scratch data to the objects that will not normally need to have such variables. Decorators provide a flexible alternative to subclassing for extending functionality.

The classes and/or objects participating in this pattern are:

- *Component* defines the interface for objects that can have responsibilities added to them dynamically.
- *ConcreteComponent* defines an object to which additional responsibilities can be attached.
- *Decorator* maintains a reference to a *Component* object and defines an interface that conforms to *Component*'s interface.
- *ConcreteDecorator* adds responsibilities to the component.

For example, in implementing balanced binary search trees we can use a binary search tree class to implement a balanced tree. However, the nodes of a binary search tree will have to store extra information such a a balance factor (for AVL trees) or a color bit (for red-black) trees). Since the nodes of a generic binary search tree do not have such variables, they can be provided in the form of decorations.

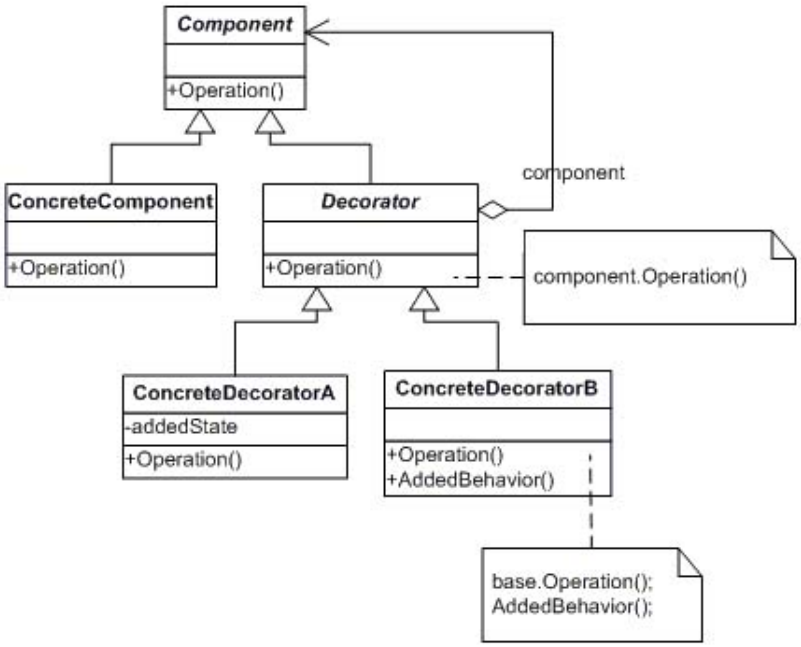

Figure 5: *Decorator* design pattern.

In the implementation of graph traversal algorithms, such as depth-first and breadth-first we can use the decorator design pattern to store temporarily information about whether a certain vertex of the graph has been visited.

2.5 *Composite*

Composite design pattern compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly [3].

The classes and/or objects participating in this pattern are:

- *Component* declares the interface for objects in the composition, implements default behavior for the interface common to all classes, as appropriate, and declares an interface for accessing and managing its child components. Optional it could defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- *Leaf* represents leaf objects in the composition. A leaf has no children, and defines behavior for primitive objects in the composition.
- *Composite* defines behavior for components having children, stores child components, and implements child-related operations in the *Component* interface.
- *Client* manipulates objects in the composition through the *Component* interface.

The implementation of tree structures are obvious examples of using *Composite* design pattern. A formal definition of a tree says that it is either empty (no nodes), or a root and zero or more subtrees. In the case of binary trees each composite component has two children: left and right subtrees.

Any operation on these binary trees could be implemented by the applying the following three steps (not necessarily in this particular order):

- apply the operation to the left subtree;

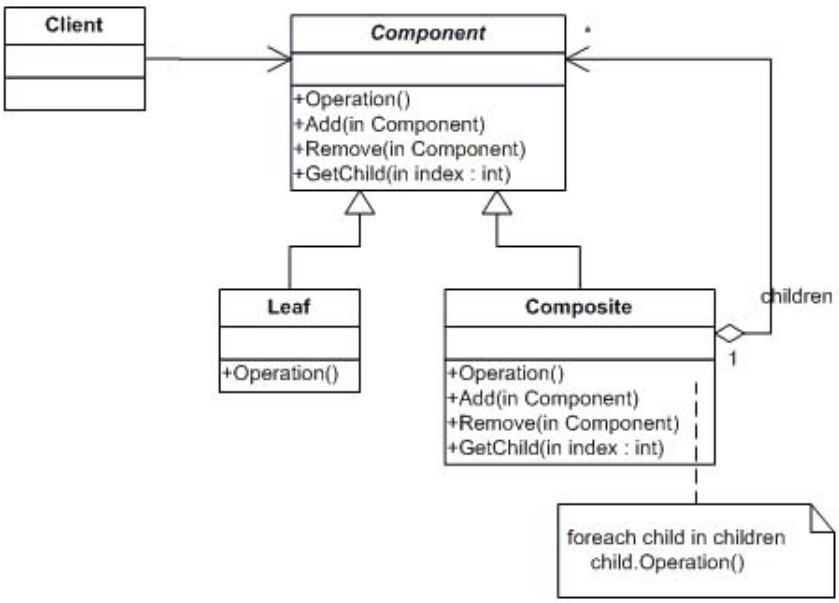

Figure 6: *Composite* design pattern.

- apply the operation to the right subtree;

- apply the operation to the root;

and then combine the results.

Examples of such operations are: determining the height, the number of nodes, etc. There is a strong relation between recursion and this way of representing data structures.

Multidimensional or heterogeneous linked lists may also be implemented based on this design pattern. A heterogeneous linked list is formed by elements which are not of the same type: they could be simple data or they could be also lists. So, a node of such a list is either an atomic node (contains a datum, and no link reference), or a node that refer to a sublist. As for trees the operations on heterogeneous list are easily implemented if the representation is based on *Composite* pattern.

3 Conclusion

We have presented advantages for data structures implementation of the following design patterns: *Flyweight, Bridge, Adapter, Decorator, Composite*.

Structural patterns focus on the composition of classes and objects into larger structures. They deal with run-time compositions that are more dynamic than traditional multiple inheritance, object sharing and interface adaptation, and dynamic addition of responsibilities to objects.

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, March 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.

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