Behavioral 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 behavioral 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

Data structures [4, 10] 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 allow 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.

We may start from the definition of a data structure: A data structure is a group of data/elements, which has an *organization* defined by a *structure* and by a specific *set of operations*. For each data structure an Abstract Data Type may be defined: the domain describes the structure and the elements' shape, and the operations define the behavior. Interfaces that describe the types could be defined for each data structure; they describe only the operations, but the structures are implied by them.

Design patterns may move the things forward, and introduce more flexibility and reusability for data structures. We intend to present the advantages of using design pattern in data structures implementation into a serie of three articles. The decision to split the presentation has been made in order to be able to give examples and explanations. In the first article we will discuss about behavioural design patters, in the second one the analysis will be done on structural design patterns, and the last will treat creational design patterns.

It is not our intention to present an exhaustive study. We present some examples that illustrate the impact of design patterns on the implementation of data structures, how they could increase the genericity and flexibility.

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

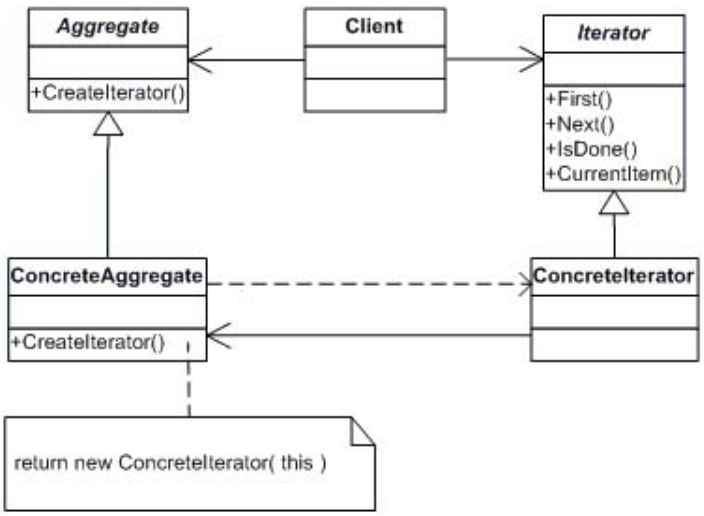

Figure 1: `Iterator` design pattern.

2 Behavioral Design Patterns Used for Data Structures Implementation

2.1 *Iterator*

Iterator design pattern [3] provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

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

- *Iterator* defines an interface for accessing and traversing elements.
- *ConcreteIterator* implements the Iterator interface, and keeps track of the current position in the traversal of the aggregate.
- *Aggregate* (AbstractContainer) defines an interface for creating an *Iterator* object
- *ConcreteAggregate* (Container) implements the *Iterator* creation interface to return an instance of the proper *ConcreteIterator*.

The *Iterator* design pattern is maybe the first design pattern that has been used for data structures. Its advantages are so important such that probably now there is no data structures library that does not use it. For each container (aggregate object) we may define an iterator class which implements the general interface *Iterator*.

The key idea in this pattern is to take the responsibility for access and traversal out of the container object and put it into an iterator object. An iterator object is responsible for keeping track of the current element; it knows elements have been traversed already. An iterator allows us to access an aggregate object's content without exposing its internal representation, but also support multiple traversals, and provide uniform interface for traversing different containers (polymorphic iteration).

Iterator has many implementation variants and alternatives. Based on who control the iteration we may classify iterators as external iterator, when the client controls the iteration, and internal iterators when the iterator controls it. If we consider who defines the traversal algorithm, when the container defines it and use the iterator to store just the state of the iteration, we have a cursor, since it merely points to the current position in the container (a well known example is a list with cursor).

An iterator is consider to be robust if ensures that insertions and removals do not interfere with traversal, and it does it without copying the container.

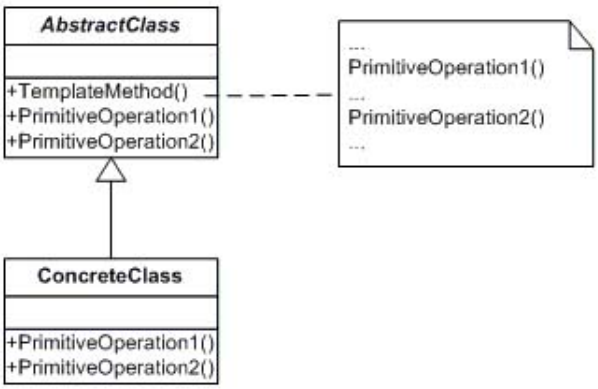

Figure 2: *Template Method* design pattern.

An iterator can be viewed as an extension of the container that created it. The iterator and the container are tightly coupled.

2.2 *Template Method*

Template Method design pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure [3] .

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

- *AbstractClass* defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in *AbstractClass* or those of other objects.
- *ConcreteClass* implements the primitive operations ot carry out subclass-specific steps of the algorithm

The following example is closely related to iterators. Based on iterators, many generic operations for containers can be defined. The example considers the application of a specific operation to the elements of a container. For example, we need to print the elements of a container, or we need to transform all the elements of a container based on the same rule, etc. For implementing this, an abstract class `OperationOnStructure` could be defined, as it is showed in the Figure 3. The method `applyOperation` is the template method, and `operation` is the abstract method that is defined in the subclasses. Using these classes it is possible to print a list, or a binary tree, or to square the elements of an list of integers, etc.

This approach is good if all the objects have the same type since a cast operation is done inside the concrete operations. When we have objects of different types stored in a container, and we want to apply different operations (depending on the concrete type of the element) then *Visitor* design pattern is appropriate to be used.

Another interesting example of using *Template Method* pattern is related to the implementation of different sorting methods, based on Merritt taxonomy [5]. At the top of her sorting taxonomy is an abstract divide-and-conquer algorithm: split the array to be sorted into two subarrays, (recursively) sort the subarrays, and join the sorted subarrays to form a sorted array. Merritt considers all comparison-based algorithms as simply specializations of this abstraction and partitions them into two groups based on the complexity of the split and join procedures: easy split/hard join and hard split/easy join. At the top of the groups easy split/hard join and hard split/easy join are merge sort and quick sort, respectively, and below them will fit all other well-known, more "low-level" algorithms. For example, splitting off

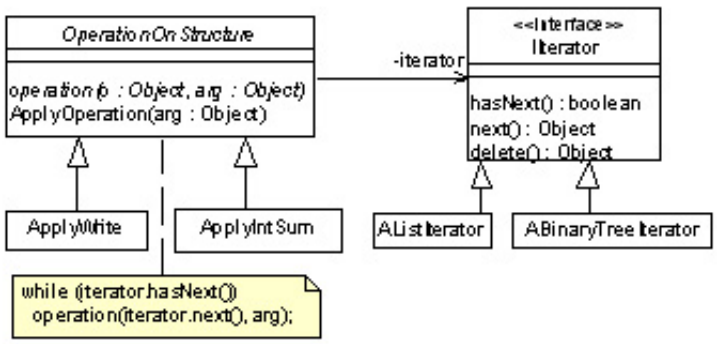

Figure 3: *OperationOnStructure Template Method* Class.

only one element at each pass in merge sort results in insertion sort. Thus insertion sort can be viewed as a special case of merge sort.

Sorting could be modeled as an abstract class with a template method to perform the sorting. This method delegates the splitting and joining of arrays to the concrete subclasses, which use an abstract ordering strategy to perform comparisons on objects [7].

2.3 *Visitor*

Visitor design pattern is used in order to represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

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

- *Visitor* declares a *Visit* operation for each class of *ConcreteElement* in the object structure. The operation's name and signature identifies the class that sends the *Visit* request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the elements directly through its particular interface
- *ConcreteVisitor* implements each operation declared by *Visitor*. Each operation implements a fragment of the algorithm defined for the corresponding class or object in the structure. *ConcreteVisitor* provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
- *Element* defines an *Accept* operation that takes a visitor as an argument.
- *ConcreteElement* implements an *Accept* operation that takes a visitor as an argument
- *ObjectStructure* can enumerate its elements may provide a high-level interface to allow the visitor to visit its elements may either be a *Composite* (pattern) or a collection such as a list or a set

When we need to introduce different operations on the nonhomogeneous elements of a data structure, *Visitor* design pattern could be applied.

In the Figure 5 we present the interfaces and an abstract class that could be used in order to apply *Visitor* pattern for containers. A *Visitor* has to be defined as an interface that contains different methods that correspond to different types (types of the elements that will be stored in the container). Elements of a visitable container should be also visitable (they have to implement the interface *Visitable*). In a visitable element class, the operation *accept* calls the corresponding method of *Visitor*, depending on the concrete type of that element.

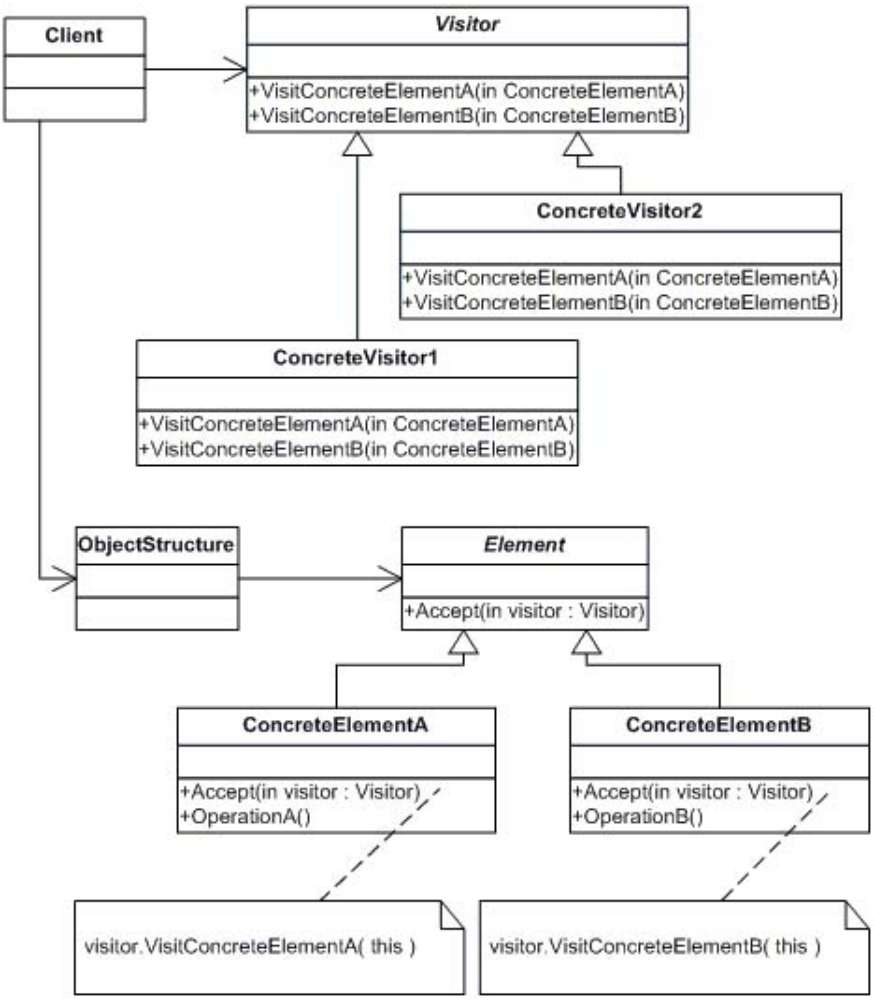

Figure 4: *Visitor* design pattern.

```
interface Element {
   public void Accept (Visitor v) ;
}
abstract class VisitableContainer implements Container, Element{
   public abstract void Accept (Visitor v) {
      Iterator it = getIterator(); //getIterator is abstract here
      while (it.hasNext()){
         Element vo = (Element)it.getElement();
         vo.Accept(v);
         it.next();
      }
   }
}
```

Figure 5: Java interfaces and abstract classes used in order to apply *Visitor* pattern for containers.

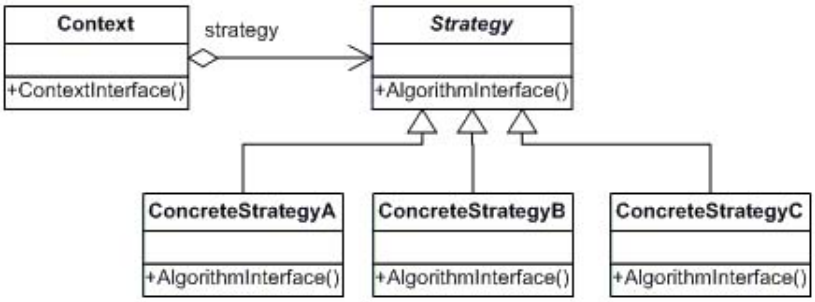

Figure 6: *Strategy* design pattern.

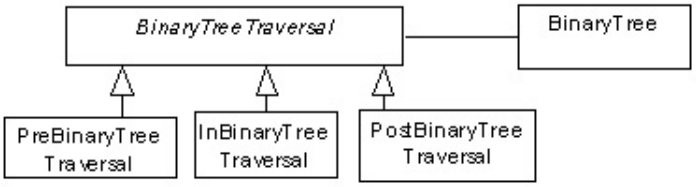

Figure 7: A *Strategy* class for tree traversal.

2.4 *Strategy*

Strategy design pattern defines a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it [3].

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

- *Strategy* declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a *ConcreteStrategy*;
- *ConcreteStrategy* implements the algorithm using the *Strategy* interface;
- *Context* is configured with a *ConcreteStrategy* object, maintains a reference to a *Strategy* object and may define an interface that lets *Strategy* access its data.

A good example is represented by the binary tree traversals. We may have preorder, inorder and postorder traversal for binary trees. In this way we may dynamically choose the traversal order for a binary tree (Figure 7). Encapsulating the algorithm in separate *Strategy* classes lets us vary the algorithm independently of its context, making it easier to switch and understand.

2.5 *Comparator*

For sorted structures it is very important to allow different comparison criteria. Also, defining a generic sorted structure means that we may construct sorted structures on different types of objects with different comparison operations (Figure 8). *Comparator* design pattern could be seen as a special kind of *Strategy* design pattern, since it specifies how two objects are compared.

Comparators are very common and they were introduced before using object oriented programming for data structures. Function parameters are used in imperative programming in order to implement comparators. They are used especially for sorted structures, and sorting algorithms.

This design pattern can be successfully used for priority queues, too. An abstract class `PriorityQueueWithComparator` is defined, which implements the interface `PriorityQueue`, and aggragates the `Comparator`. In this way, the priorities of the elements are not stored into the queue; we establish only an order between elements, using a comparator.

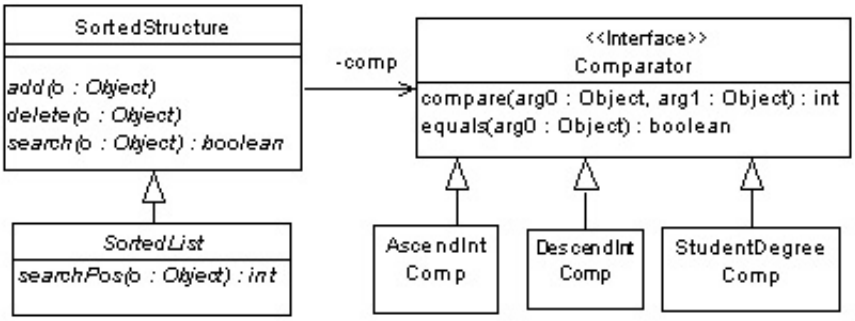

Figure 8: Using *Comparator* for implementing sorted list with different comparison criteria.

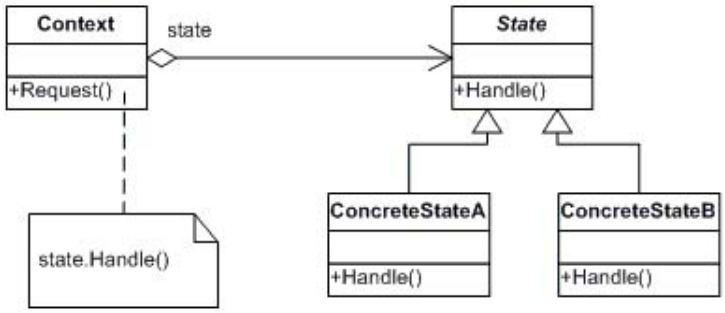

Figure 9: `State` design pattern.

2.6 *State*

An interesting implementation of containers using *State* design pattern is presented in [6]. The key here is to encapsulate states as classes. A container class could be considered as an abstraction defined by three methods: `insert, remove`, and `retrieve`. Most implementations based on dynamic memory allocation use the null pointer to represent the empty structure. Because the semantics of a null pointer is too low-level to adequately encapsulate the behavior of an object, such implementations have a high degree of code complexity, and are cumbersome to use. A null pointer (or a null reference in Java) has no behavior and thus cannot map well to the mathematical concept of the empty set, which is an object that exists and that has behavior. The goal is to narrow the gap between the conceptual view of a container structure and its implementation. In this convention, the null pointer is to represent the non-existence of an object only. Emptiness and non-emptiness are simply states of a container

A container structure is a system that may change its state from empty to non-empty, and vice-versa. For example, an empty container changes its state to non-empty after insertion of an object; and when the last element of a container is removed, its changes its state to empty.

For each distinct state, the algorithms to implement the methods differ. For example, the algorithm for the retrieve method is trivial in the empty state -it simply returns null- while it is more complicated in the non-empty state. The system thus behaves as if it changes classes dynamically. This phenomenon is called "*dynamic reclassification*".

The *State* pattern is a design solution for languages that do not support dynamic reclassification directly. This pattern can be summarized as follow:

- Define an abstract class for the states of the system. This abstract state class should provide all the abstract methods for all the concrete subclasses.
- Define a concrete subclass of the above abstract class for each state of the system. Each concrete state must implement its own concrete methods.

- Represent the system by a class containing an instance of a concrete state. This instance represents the current state of the system.
- Define methods for the system to return the current state and to change state.
- Delegate all requests made to the system to the current state instance. Since this instance can change dynamically, the system will behave as if it can change its class dynamically.

Application of the *State* pattern for designing a linked list class is straightforward. We name this class, `List`, and the abstract class for list states, `AListNode` (as in abstract list node). `AListNode` has two concrete subclasses: `EmptyListNode`, and `NonEmptyListNode`. The `EmptyListNode` has no data while the `NonEmptyListNode` contains a data object, and a tail, which is a `List`. One can see how closely this implementation maps to the following portion of the abstract definition of a list: If a list is empty, it contains no data object. If it is not empty, it contains a data object called the head, and a list object called the tail. The class `List` contains an instance of a concrete subclass of `AListNode`. Via polymorphism, it can be an `EmptyListNode` or a `NonEmptyListNode` at run time. In order to qualify it as a container class, we add to the behavior of class `List` the three container methods: `insert, remove`, and `retrieve`. A sketch of the Java implementation of such a list is given in Figure 9.

3 Conclusion

Using design patterns in data structures implementation represents a step forward in this field. Design patterns allow data structures to be implemented in a very general and flexible way. We have presented some examples of data structures that use the advantages brought by structural design patterns: *Iterator, Template Method, Visitor, Stategy, Comparator, State.*

Behavioral patterns deal with encapsulating algorithms and managing or delegating responsibility among objects. They focus more on communication and interaction, dynamic interfaces, object composition, and object dependency.

For teaching, introducing design patterns in presentation of the data structures could represent also an important advantage. Students may beneficiate of an easy way of understanding design patterns in the early stages of their preparation [8]. It is not need for complex applications in order to illustrate some design patterns, and their advantages. In this way, students realize that what important is to use design patterns at any level of the design.

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] S. Merritt, *An Inverted Taxonomy of Sorting Algorithms*, Comm. ACM, 28, 1 (Jan. 1985), 96-99.

[6] D. Nguyen. *Design Patterns for Data Structures*. SIGCSE Bulletin, 30, 1, March 1998, pp. 336-340.

[7] D. Nguyen. *Design Patterns for Sorting*. SIGCSE Bulletin 2001 2/01 pp. 263-267.

[8] V. Niculescu, *Teaching about Creational Design Patterns*, Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts, ECOOP'2003, Germany, July 21-25, 2003.

```
public class List implements IContainer
{
private AListNode _link; //state.
AListNode link () {
return _link;
}
void changeLink (AListNode n) {
//Change state;
_link = n;
}
public List (){
//Post:  this List exists and is empty.
//constructor code goes here
}
public void insert (Object key, Object v) {
//Pre :  key and v are not null.
_link.insert (this, key, v);
}  //***Other constructors and methods...
}
abstract class AListNode {
abstract void insert (List l, Object k, Object v);
//Pre :  l, k and v are not null.
//***Other abstract methods...
}
class NonEmptyListNode extends AListNode {
private Object _key;
private Object _val;
private List _tail;
NonEmptyListNode (Object k, Object v) {
//Pre :  k and v are not null.
//Post:  this node exists and contains k, v, and an empty
tail.
_key = k;
_val = v;
_tail = new List ();
}
void insert (List l, Object k, Object v) {
if (k.equals (_key)) {
_val = v;
}
else {
_tail.insert (k, v);
}
}
//***Other methods
}
class EmptyListNode extends AListNode {
void insert (List l, Object k, Object v) {
changeLink (new NonEmptyListNode (k, v);
} //***Other methods
}
```

Figure 10: Java implementation of linked lists using *State* pattern.

[9] V. Niculescu. *On Choosing Between Templates and Polymorphic Types. Case-study.*, Proceedings of "Zilele Academice Clujene", Cluj-Napoca, June 2003, pp.71-78.

[10] D.M. Mount. *Data Structures*, University of Maryland, 1993.