Acta Universitatis Apulensis No. 34/2013
ISSN: 1582-5329 pp. 335-354

A DESIGN PATTERNS PERSPECTIVE ON DATA STRUCTURES

Virginia Niculescu

Abstract. Design patterns may introduce a new perspective on the traditional subject of data structures. We analyze in this paper how design patterns can be used for data structures implementation, and their advantages. The presentation is leaded by the design patterns classification: behavioral, structural and creational, and we prove here that all classes of patterns could be successfully used for data structures implementation. By using design patterns for data structures construction we emphasize new perspectives on data structures: separation between concepts and their representations, storage independence, reusability, and others. These could bring important changes in the design of the new collections libraries or frameworks. Examples that illustrate the impact of the design patterns in this context are presented, too. This analysis shows that design patterns could bring important advantages in basic fields of computer science, too.

2000 *Mathematics Subject Classification*: 68P05.

1 Introduction

Data structures (DS) represent an old issue in the Computer Science field [2, 4]. 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 [2, 11]. 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 [10, 11].

Design patterns may move the things forward, and introduce more flexibility and reusability for data structures. In this paper we intend to present the how behavioral, structural and creational design patterns could be used for implementing data structures. The presentation is leaded by the design patterns and their classification given in [1]. We prove here that all classes of design patterns could be successfully used for data structures implementation.

By using design patterns in data structures construction we emphazise new perspectives on data structures: separation between concepts and their representations, storage independence, reusability, and scalability. These could bring important changes in the design of the new collections libraries or frameworks.

We also present some examples that illustrate the impact of design patterns on the implementation of data structures, how they could increase the genericity and flexibility.

2 Behavioral Design Patterns Used for DS Implementation

Behavioral patterns deal with encapsulating algorithms, and managing or delegating responsibility among objects [1].

2.1 Iterator

Iterator design pattern [1] provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

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.

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).

The classical responsibility of an iterator is specified by the following operations: retrieving the current element, moving to the next element, and verifying the existence of un-iterated elements; this corresponds to a read-only, one-directional iterator. If also the previous element (relatively to the current) could be retrieved, then the iterator is a bidirectional one, and if the elements could be retrieved randomly then the iterator is a random-type iterator (such an iterator is usual for array based representation of the container). Beside this kind of operations an iterator

could define removal and insertion operations; in this case we have read-and-write iterators. Such an iterator is considered to be robust if ensures that insertions and removals do not interfere with traversal, and it does it without copying the container.

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

2.2 Template Method

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

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 to 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 1. 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 a non-homogeneous 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 S. Merritt taxonomy [3]. 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. This approach 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

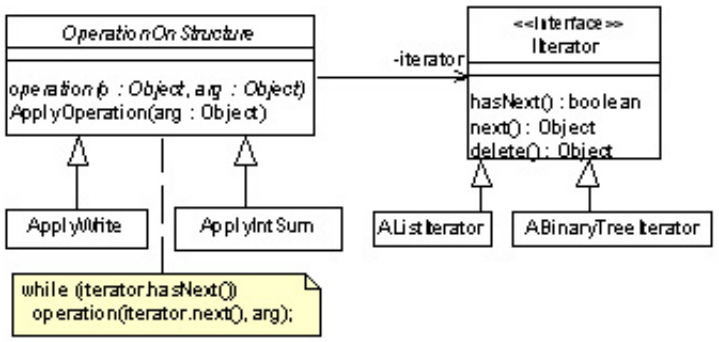

Figure 1: *`OperationOnStructure` Template Method* Class.

will fit all other well-known, more "low-level" algorithms. For example, splitting off 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 [6].

2.3 Visitor

`Visitor` design pattern is used in order to represent an operation to be performed on the elements of an object structure. Usually, it is used in order to aplly a certain operation to all elements of a container. Visitor lets you define a new operation without changing the classes of the elements on which it operates [1].

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 type 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* defines a visitable container and it can enumerate its elements; it may provide a high-level interface to allow the visitor to visit its elements, and it may

```
interface Element {
   public void accept (Visitor v) ;
}
abstract class ObjectStructure 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 2: Java interface - `Element`, and an abstract visitable agregate class.

either be a *Composite* (pattern) or a collection such as a list or a set.

Visitor design pattern could be successfully applied when we need to introduce different operations on the non-homogeneous elements of a data structure.

In the Figure 2 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 *Element*). In a visitable element class, the operation *accept* calls the corresponding method of *Visitor*, depending on the concrete type of that element.

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 [1].

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 3). Encapsulating

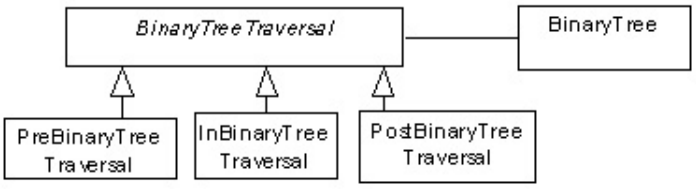

Figure 3: A *Strategy* class for tree traversal.

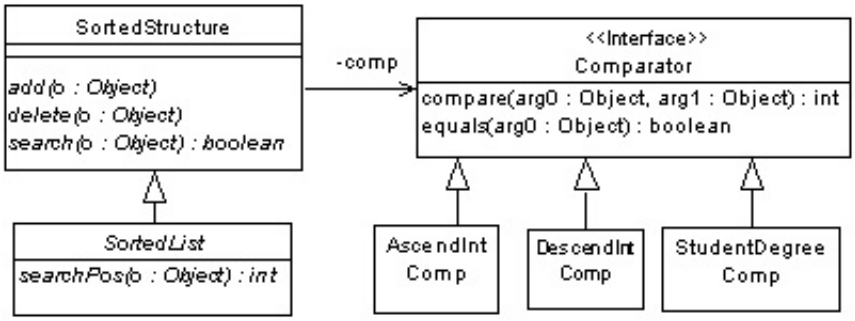

Figure 4: Using *Comparator* for implementing sorted lists with different comparison criteria.

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 4). *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 implementation. 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, and this implements the priority queue interface, 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.

2.6 State

An interesting implementation of containers using *State* design pattern is presented in [5]. 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. Using this convention, the null pointer is used to represent the non-existence of an object only.

The gap between the conceptual view of a container structure and its implementation could be narrowed this way. 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 that 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 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 simple. 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

```
public class List implements IContainer{
   private AListNode _link; //state
   AListNode link () {
      return _link;
   }
   void changeLink (AListNode n) {
   //Change state;
      _link = n;
   }
     //Post:  this List exists and is empty.
   public List (){
   }
     //Pre :  key and v are not null.
   public void insert (Object key, Object v) {
      _link.insert (this, key, v);
   }

//***Other constructors and methods...  }
abstract class AListNode {
     //Pre :  l, k and v are not null.
   abstract void insert (List l, Object k, Object v);
   //***Other abstract methods...
}
class NonEmptyListNode extends AListNode {
   private Object _key;
   private Object _val;
   private List _tail;
      //Pre :  k and v are not null.
      //Post:  this node exists and contains k, v, and an
empty tail.
   NonEmptyListNode (Object k, Object v) {
      _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 5: Java implementation of linked lists using *State* pattern.

`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, the class `List` adds to its behavior the three container methods: `insert, remove`, and `retrieve`. A sketch of the Java implementation of such a list is given in Figure 5.

3 Structural Design Patterns Used for DS 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. 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 [1].

3.1 Flyweight design pattern

A *flyweight* is an object that minimizes memory usage by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory. Often some parts of the object state can be shared, and it is common practice to hold them in external data structures and pass them to the flyweight objects temporarily when they are used.

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.

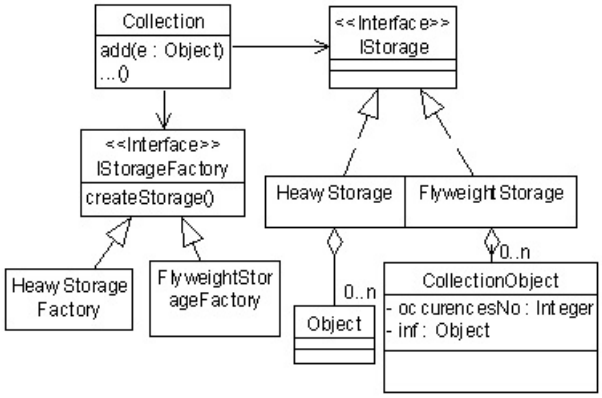

Figure 6: An implementation for collections based on *FlyWeight*, *Bridge*, and *Abstract Factory*.

- *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 a simplification of this design pattern when we implement a collection (bag) data structure. 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 6). 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.

3.2 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) [9]. For 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 section).

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 considered 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. (*Singleton* is used since we don't need more than one instance of a specific factory class.)

3.3 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 may 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.

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 state 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.

3.4 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 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.

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.

3.5 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 [1].

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 define 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 the 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;
- 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.

4 Creational Patterns used for DS Implementation

Creational design patterns abstract the instantiation process. A class creational pattern uses inheritance to vary the class that is instantiated, whereas an object creational pattern will delegate instantiation to another object [1].

4.1 Abstract Factory and Singleton

Abstract Factory design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes [1]. We can use it when:

- a system should be independent of how its products are created, composed, and represented.
- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and we need to enforce this constraint.
- we want to provide a class library of products, and we want to reveal just their interaces, not their implementations.

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

- *AbstractFactory* declares an interface for operations that create abstract products;
- *ConcreteFactory* implements the operations to create concrete product objects;
- *AbstractProduct* declares an interface for a type of product object;
- *Product* defines a product object to be created by the corresponding concrete factory, and implements the *AbstractProduct* interface;
- *Client* uses interfaces declared by *AbstractFactory* and *AbstractProduct* classes.

An example of using *AbstractFactory* for data structures has been presented in Section 3 when we have emphasized the possibility to separate the storage of a container from its interface. AbstractFactory classes are defined in order to allow choosing the type of storage when a container is instantiated.

If we consider the case of the stacks, we have an interface `Stack` that defines the specific operations based on ADT Stack; a stack is a specialized list, and so it could be constructed based on a list. But we may use different kind of lists: ArrayList, DynamicLinkedList (with dynamically allocated nodes), or StaticLinkedList (with nodes which are allocated into a table). We don't have to define many classes for

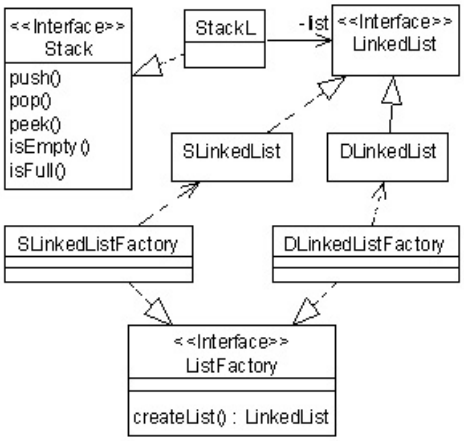

Figure 7: Using *Abstract Factory* and *Adapter* design patterns for implementing stacks.

```
class StackL implements Stack {
   public StackL (ListFactory factory) {
      list = factory.createList(); // an empty list is created
   }
   public void push(Object o) {
      list.addFirst(o);
   }
   public Object pop() {
      o = list.removeFirst();
      return o;                                                    }
   }
   public Object peek() {
      o = list.getFirst();
      return o;
   }
   public boolean isEmpty() {
      return list.isEmpty();
   }
   private LinkedList list;
};
```

Figure 8: Java implementation of the class `StackL`.

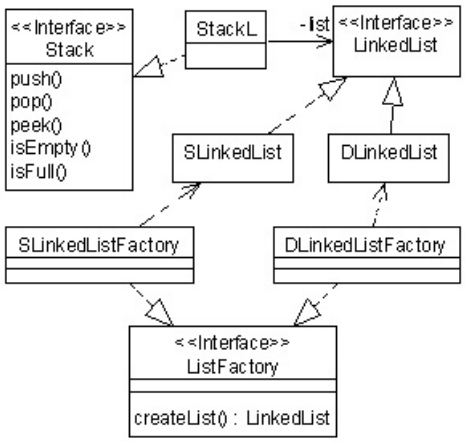

Figure 9: Using *Abstract Factory* and *Adapter* design patterns for implementing stacks.

stacks: it is enough if we define factories that create different kinds of list as a base for stacks which are being instantiated (see Figure 9).

In order to make these things clear, we present a Java implementation of the class `StackL` in the Figure 8. The stack has a member `list` of type `LinkedList`, which is created when the stack is instantiated; the creation is the responsibility of the factory which is given as a parameter to the constructor. If the parameter has the type `SLinkedListFactory` it will create a list of type `SLinkedList`, and if it has the type `DLinkedListFactory` it will create a list of type `DLinkedList`.

A stack could be created either using `SLinkedListFactory` or `DLinkedList-Factory` (Figure 10). Only one instance is necessary for any kind of factory, so *Singleton* design pattern is used, too. Usually, we don't need more than one instance of a factory class.

Singleton design pattern ensures a class has only one instance and provides a global point of access to it [1].

Another example could be given for priority queues. A *Priority Queue* is a queue in which elements are added based on their priorities. These priorities could represent different things for different objects and they may have different types. Also, for objects of a certain type we may define different priorities. For example, for students we may define a priority depending on their average degree when we intend to use a priority queue for scholarships, but if we intend to give social scholarships we need a priority depending on their social situation (which may be computing based on several attributes of the `Student` class). In order to achieve this flexibility,

```
// a stack based on a static linked list
   Stack s1 = new StackL(SLinkedListFactory.instance());
// a stack based on a dynamic linked list
   Stack s2 = new StackL(DLinkedListFactory.instance());
```
}

Figure 10: Examples of `StackL` instantiation.

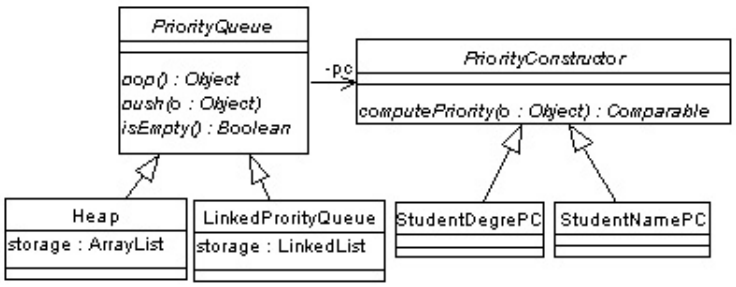

Figure 11: Using *Abstract Factory* for implementing a priority queue.

we may use *Abstract Factory* design pattern, where the factory will be a factory of priorities [8]. It can be argued that a comparator, as for sorted structures, may be enough but there are cases when we need to know also the values for the chosen priority.

`PriorityConstructor` is the abstract factory class, and its method `computePriority` calculates the priority for the object argument; the result should be an object which is `Comparable`. Subclasses of this abstract factory are defined for special priorities for special types of objects. A single instance of each of these subclasses is enough, so *Singleton* [1] design pattern can be used, too.

In addition, one advantage of this approach of implementing priority queues is that we do not have to store the priorities for the constitutive elements. They may be computed at every moment by the `PriorityConstructor`.

4.2 Builder

Builder design pattern separates the construction of a complex object from its representation, so that the same construction process can create different representations.

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

- *Builder* specifies an abstract interface for creating parts of a *Product* object;
- *ConcreteBuilder* constructs and assembles parts of the product by implementing the *Builder* interface, defines and keeps track of the representation it creates, and provides an interface for retrieving the product;

```
// Builder for making trees - base class
class TreeBuilder {
   public void AddNode(TreeNode theNode) {}
   public TreeNode GetTree() { return null; }
   protected TreeBuilder() {};
}
```

Figure 12: Builder for making trees - the base class.

```
class BinaryTreeBuilder extends TreeBuilder {
   public BinaryTreeBuilder(){ _currentBTree = null;}
   public void AddNode(TreeNode theNode){
      ...
      }
   public TreeNode GetTree(){
      return _currentBTree;
   }
   private TreeNode _currentBTree;
}
```

Figure 13: Builder for making binary trees.

- *Director* constructs an object using the *Builder* interface;
- *Product* represents the complex object under construction. *ConcreteBuilder* builds the product's internal representation and defines the process by which it's assembled, and includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

The following example uses *Builder* design pattern in order to create binary search trees. Two different builders are defined: one that create simple binary search trees (unbalanced), and another that create balanced binary search trees.

The `TreeBuilder` class hierarchy implements the *Builder* design pattern. It encapsulates the building of a composite data structure and hides the details of how that structure is composed by defining an `AddNode` member function to add nodes to the tree. The base class defines this interface.

New nodes are added to the current tree being built by allocating them and passing them to the TreeBuilder `AddNode` member function. The completed tree is returned by the `GetTree` member function. The `BinaryTreeBuilder` subclass implements a simple binary tree. No effort is made to balance the tree.

The `AddNode` member function traverses the current tree, comparing the new node to the current node. Once the correct location is located (i.e., a leaf node is reached) the new node is added as a left or right child of the leaf node.

```
class HBTreeBuilder extends TreeBuilder {
   public HBTreeBuilder(){
         _currentBTree = null;
       }
   public TreeNode GetTree(){
         return _currentBTree;
   }
   public void AddNode(TreeNode theNode){
         ...
      }
   private TreeNode _currentBTree;
}
```

Figure 14: Builder for making height-balanced binary trees.

The `HBTreeBuilder` subclass builds height-balanced binary trees. To implement a height-balanced binary tree we restructure the tree as we add new nodes. This class has the same structure as `BinaryTreeBuilder`; the difference is in the `AddNode` member function, which traverses the current tree, comparing the new node to the current node. This version keeps track of the previous two nodes visited as the tree is traversed. Once the correct location is found (that is, a leaf node is reached) the current and previous nodes are checked to see if both lack a child on the opposite side. If they do, then the subtree from the grandparent node (that is, two nodes above the leaf) is re-arranged such that it forms a balanced tree when a new node is added.

5 Conclusion

Design patterns allow data structures to be implemented in a very general and flexible way. Since we have so many types of data structures the possibility of creating some of these based on some fundamental data structures and add new properties easily could influence the scalability of a collections library bringing important advantages.

Behavioral patterns focus more on communication and interaction, dynamic interfaces, object composition, and object dependency. We have presented some examples of data structures that use the advantages brought by behavioral design patterns: *Iterator, Template Method, Visitor, Stategy, Comparator, State*. Structural patterns focus on the composition of classes and objects into larger structures. The given examples illustrates the advantages of the following design patterns: *Flyweight, Bridge, Adapter, Decorator, Composite*. When you are designing complex

object oriented systems we often rely on composite objects along with class-based inheritance. Creational patterns are used to delegate instantiation, abstract behavior, and hide instantiation and composition details. By using some examples we have presented advantages of the patterns: *Abstract Factory*, *Singleton* and *Builder*, when are used for data structures implementation.

Understanding design patterns is an important aspect of modern software development. Usually design patterns are introduced late into an undergraduate curriculum. 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 [7]. We believe that design patterns deserve a more ubiquitous role in the undergraduate curriculum. Some modifications and simplifications of the design patterns could be necessary when they are used for data structures construction, but these do not interfere with the primary message of each design pattern but rather highlight it.

References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995.
[2] E. Horowitz. *Fundamentals of Data Structures in C++*. Computer Science Press, 1995.
[3] S. Merritt, *An Inverted Taxonomy of Sorting Algorithms*, Comm. ACM, 28, 1 (Jan. 1985), 96-99.
[4] D.M. Mount. *Data Structures*, University of Maryland, 1993.
[5] D. Nguyen. *Design Patterns for Data Structures*. SIGCSE Bulletin, 30, 1, March 1998, pp. 336-340.
[6] D. Nguyen. *Design Patterns for Sorting*. SIGCSE Bulletin 2001 2/01 pp. 263-267.
[7] V. Niculescu. *Teaching about Creational Design Patterns*, Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts, ECOOP'2003, Germany, July 21-25, 2003.
[8] V. Niculescu. *Priority Queues Implementation Based on Design Patterns*, Proceedings of the Symposium "Zilele Academice Clujene", 2006, pp. 27-32.
[9] V. Niculescu. *Storage Independence in Data Structures Implementation*, Studia Universitatis "Babes-Bolyai", Informatica, Special Issue, LVI(3), pp. 21-26, 2011.
[10] D.R. Musser, A. Scine, *STL Tutorial and Reference Guide: C++ Programming with Standard Template Library*, Addison-Wesley, 1995.
[11] B.R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*, Wiley Computer Publishing, 1999.

Virginia Niculescu
Department of Computer Science
Babes-Bolyai University of Cluj-Napoca
Address: 1, M. Kogalniceanu, Cluj-Napoca, Romania
email:*vniculescu@cs.ubbcluj.ro*