Creational Design Patterns Used in Data Structures Implementation
Niculescu Virginia
Department of Computer Science
Babeș-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 creational 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

In order to build a data structure that is extensible and reusable, it is necessary to decouple the intrinsic and primitive behavior of the structure from the application specific behavior that manipulates it. In order to achieve this, design patterns can be used.

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

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 [3].

2 Abstract Factory and Singleton

Abstract Factory design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes [3]. 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 interfaces, not their implementations.
Figure 1: Abstract Factory design pattern.

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 the article “Structural Design Patterns Used in Data Structures Implementation” 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 kinds of list: 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 stacks: it is enough if we define factories that create different kinds of lists as base for stacks which are being instantiated (see Figure 2).

In order to make these things clear, we present a Java implementation of the class **StackL** in the Figure 3. 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**.
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 2: Using Abstract Factory and Adapter design patterns for implementing stacks.

Figure 3: Java implementation of the class StackL.
A stack could be created either using SLinkedListFactory or DLinkedListFactory (Figure 4). 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 [3]. The Figure 5 specify for the class that implements this pattern, a static member instance, and a static accessing method instance() that returns the unique instance. An important thing is the fact that the constructor is not public.

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, we may use Abstract Factory design pattern, where the factory will be a factory of priorities. It can be argued that a comparator, as for sorted structures, may be enough but in this case 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 [3] 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.
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;
- **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 **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
class BinaryTreeBuilder extends TreeBuilder {
    public BinaryTreeBuilder() { _currentBTree = null; }
    public void AddNode(TreeNode theNode){
        ...
    }
    public TreeNode GetTree(){
        return _currentBTree;
    }
    private TreeNode _currentBTree;
}

Figure 9: Builder for making binary trees.

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

Figure 10: Builder for making height-balanced binary trees.

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.

4 Conclusion

As we emphasized in the previous two articles, design patterns allow data structures to be implemented
in a very general and flexible way. Here, we have presented some examples based on creational design
patterns: Abstract Factory, Singleton and Builder.

Creational patterns abstract the instantiation process. 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.

It is important to teach students computer science concepts using solid software engineering practices,
and these have to start from the basic fields as data structures.
References


