Design Patterns for the Data Structures and Algorithms Course

Bruno R. Preiss
University of Waterloo
Bruno.Preiss@UWaterloo.Ca

Abstract

Design patterns have recently emerged as a vehicle for describing and documenting recurring object-oriented designs. More significantly, they offer up a long-awaited framework for teaching good software design. This paper espouses the use of object-oriented *design patterns* in the teaching of the *second course* in computer science, viz., the data structures and algorithms course.

To use design patterns effectively, it is necessary to present the various data structures and algorithms in a common programming framework. This paper also espouses the use of a single, unified class hierarchy and the commitment to a single design throughout the teaching of the second course.

1 Introduction

One cannot learn to design programs just by reading a book. It is a skill that must be developed by practice. I believe that, after learning the rudiments of program writing, students should be exposed immediately to examples of complex, yet well-designed program artifacts so that they may learn about designing good software.

A good professional practitioner studies the works of others. With experience, he discovers the recurring patterns in software and incorporates these patterns into his own practice. By learning to use these patterns, his object-oriented designs are more flexible and reusable.

The benefits of using object-oriented methodologies and, in particular, of design patterns are profound. It does not make sense to save them for advanced courses. At many institutions the second CS course is either *data structures* course or a combined *structures and algorithms* course[5]. I believe in the teaching the second course that one should preach design patterns from day one!

What are the pedagogic benefits? It is said that "computer science is [the] science of *abstraction*[1]." Design patterns are emerging as the abstractions that are appropriate for talking about designs. They provide a framework for thinking about and for comparing design decisions and trade-offs. More importantly, they provide a common *vocabulary* for describing software designs.

Nevertheless, adopting object-oriented methods and, in particular, the use of design patterns requires a *paradigm shift*. It has been my experience that this is often harder for the teacher than it is for the student. A significant amount of intellectual inertia must be overcome.

2 Object Hierarchies and Design Patterns

There is more to object-oriented programming than simply encapsulating in an object some data and the procedures for manipulating those data. Object-oriented methods deal also with the *classification* of objects and they address the *relationships* between different classes of objects.

The primary facility for expressing relationships between classes of objects is *derivation*—new classes can be derived from existing classes. What makes derivation so useful is the notion of *inheritance*. Derived classes *inherit* the characteristics of the classes from which they are derived. In addition, inherited functionality can be overridden and additional functionality can be defined in a derived class.

Virtually all the data structures typically covered in the second course can be presented in the context of a single class hierarchy. In effect, the class hierarchy is a taxonomy of data structures. Different implementations of a given abstract data structure are all derived from the same abstract base class. Related base classes are in turn derived from classes that abstract and encapsulate the common features of those classes.

In addition to dealing with hierarchically related classes, experienced object-oriented designers also consider very carefully the interactions between unrelated classes. With experience, a good designer discovers the recurring patterns of interactions between objects. Recently, programmers have to compile and publish catalogs of the common design patterns[2].

The following sections describe the design patterns that are easily incorporated into the the second course:

2.1 Containers

A container is an object that holds within it other objects. A container has a capacity, it can be full or empty, and objects can be inserted and withdrawn from a container. In addition, a *searchable container* is a container that supports efficient search operations.

Virtually all the data structures covered in the second course can be viewed as specialized containers. Clearly, data structures such as stacks, queues, deques, ordered lists, sorted lists, hash tables, and scatter tables are viewed as containers. Even trees and graphs can be considered as specialized containers.

2.2 Iterators

An *iterator* provides a means by which the objects within a container can be accessed one-at-a-time. All iterators share a common interface, and hide the underlying implementation of the container from the user of that container.

Iterators are crucial to the teaching of data structures: This is because they allow a complete separation between the *use* of a data structure and its *implementation*. E.g., virtually every data structure can be implemented using using an array as the underlying foundational data structure or using some sort of linked (pointer-based) data structure. Iterators provide a universal interface for the systematic enumeration of the elements of a data structure regardless of its underlying implementation.

2.3 Visitors

A visitor represents an operation to be performed on all the objects within a container. All visitors share a common interface, and thereby hide the operation to be performed from the container. At the same time, visitors are defined separately from containers. Thus, a particular visitor can be used with any container.

Visitors are particularly useful when used in conjunction with tree and graph traversal functions. Typical tree traversal algorithms intermix the code which does the traversal with the application specific code. This leads to pedagogic difficulties because the purpose of a particular operation (traversal vs. application) is not manifestly evident.

By using a visitor, we separate the application specific from the traversal code. For example, we can talk about depth-first traversal in the abstract, and then show, by the implementation of suitable visitors, how application specific operations are built on top of the underlying generic traversal.

2.4 Adapters

An *adapter* converts the interface of one class into the interface expected by the user of that class. This allows a given class with an incompatible interface to be used in a situation where a different interface is expected.

An interesting application of the idea of an adapter is to illustrate the relationship between the abstract notion of depth-first traversal and specialized traversals such as preorder, inorder, and postorder traversal. For example, suppose we devise a "printing visitor," the purpose of which is to print every object it visits. By *adapting* this "printing visitor" so that it implements the required interface, it is possible to show that preorder, inorder, and postorder traversals are all just specialized versions of the depth-first traversal.

2.5 Singletons

A singleton is a class of which there is only one instance. The class ensures that there only one instance is created and it provides a way to access that instance. For example, we may use a singleton to represent explicitly a special *null object instance*.

2.6 Data Structure Class Hierarchy

Figure 1 shows a class hierarchy that comprises the data structures typically covered in the second course. Two kinds of classes are shown in the figure: *abstract classes*, which look like this **Abstract Class**, and *concrete classes*, which look like this Concrete Class. Lines in the figure indicate derivation; base classes always appear to the left of derived classes.

This taxonomy of data structures has a number of interesting characteristics:

- It clearly illustrates that many of the data structures can be implemented using either an array as the underlying foundational data structure or using a pointer-based implementation. E.g., stacks, queues, deques, lists, hash tables, sets, multisets, trees, and graphs all have array- and pointer-based implementations.

- Stacks, queues, priority queues, trees, and graphs are viewed as specialized containers; whereas lists, search trees, hash tables, and sets are viewed as specialized *searchable* containers.

- Despite well-documented objections to the use of multiple inheritance, multiple inheritance is used to show the relationships between data structures that cannot be captured otherwise. E.g., a search tree is both a tree and a searchable container; and a leftist heap is both a tree and a priority queue.

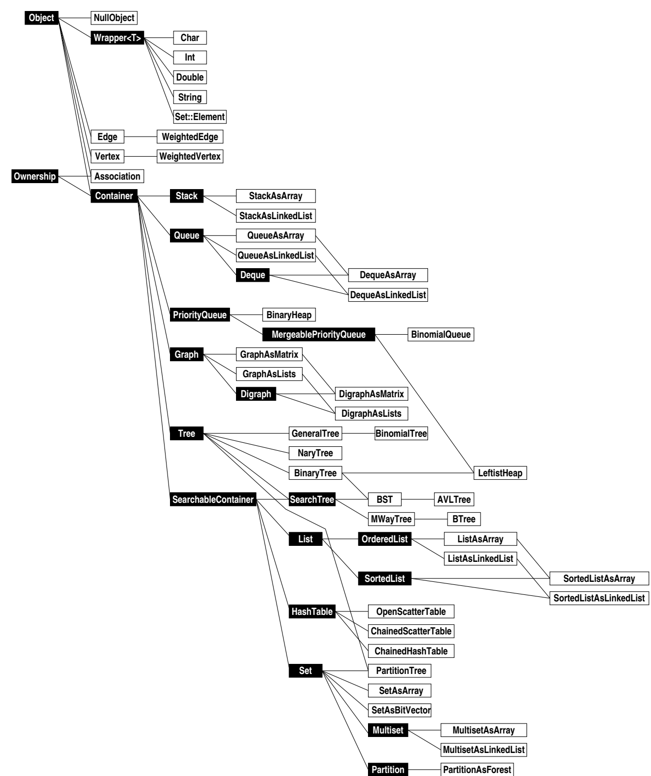

Figure 1: Object class hierarchy

2.7 Example: A C++ Language Binding

This section presents C++ code fragments which illustrate how the class hierarchy described above can be implemented. Figure 2 gives excerpts from the `Object`, `Container`, and `Visitor` class declarations. Notice that these classes are *abstract* classes that comprise only pure virtual functions. They define the interfaces shared by many of the data structures and, when combined with a precise description of the required semantics, serve as the specifications for the various abstract data types (ADTs).

Consider the `Object` class declared in Figure 2. Every object provides the following methods:

```
class Object
{
public:
    virtual int Compare (Object const&) const = 0;
    virtual HashValue Hash () const = 0;
    virtual ostream& Put (ostream&) const = 0;
    // ...
};

class Container : public virtual Object
{
public:
    virtual bool IsEmpty () const;
    virtual bool IsFull () const;
    virtual Iterator& NewIterator () const;
    virtual void Accept (Visitor&) const = 0;
    // ...
};

class Visitor
{
public:
    virtual void PreVisit (Object&) = 0;
    virtual void Visit (Object&) = 0;
    virtual void PostVisit (Object&) = 0;
    virtual bool IsDone () const = 0;
};
```

Figure 2: Object, container, and visitor classes

`Compare` compares this object with a given object;

`Hash` computes a hash value for this object; and

`Put` prints this object on the given output stream.

Similarly, every container provides the following methods:

`IsEmpty` returns true if this object is empty;

`IsFull` returns true if this object is full;

`NewIterator` returns a new instance of an iterator that enumerates the contents of this container; and,

`Accept` causes the given visitor to visit all the objects in this container.

```
class HashingVisitor : public Visitor
{
    HashValue value;
public:
    HashingVisitor ()
        { value = 0; }
    void Visit (Object& object)
        { value += object.Hash (); }
    HashValue Value () const
        { return value; }
};

HashValue Container::Hash () const
{
    HashingVisitor visitor;
    Accept (visitor);
    return visitor.Value ();
}
```

Figure 3: Hashing visitor

Figure 3 shows how these interfaces can be used to implement a "hashing visitor." The purpose of the hashing visitor is to compute a hash code for a given container by visiting one-by-one all the objects in that container and accumulating their hash codes.

Figure 3 illustrates several pedagogic benefits which accrue from the class hierarchy and design patterns:

- The use of a visitor separates the *application-specific code* (i.e., accumulate hash codes) from the *data structure-specific code* (i.e., the `Accept` function of the container). Consequently, we can talk about the implementation of a method that visits all the objects in container without having to be concerned with what it means to "visit" an object.

- The use by the visitor of the *abstract* container interface isolates the implementation of the visitor completely from the implementation of the container. This means we can talk about the computations required to perform a task that visits all the objects in a container without having to be concerned with how to visit all the objects.

3 Algorithmic Abstraction

Algorithmic abstraction is the idea that by using well-defined generic interfaces to underlying data structures we may write algorithms that are completely independent of the implementation of those underlying data structures. The pedagogic benefit of doing so accrues from this separation. I.e., an abstract algorithm captures the *essence* of the algorithm—the rest is just implementation detail.

For example, Figure 3 illustrates the use of *algorithmic abstraction*. Since every container is an object, it must provide a `Hash` function. In this case, the `Hash` function is implemented by using a hashing visitor. However, notice that the given hash function works for any derived container class (provided that class implements the required semantics). In effect the algorithm we have written is completely independent of the underlying data structures—it is an *abstract algorithm*.

The following sections describe four additional scenarios in which algorithmic abstraction can be applied.

3.1 Abstract Sorters vs. Sorting Functions

The usual way to implement a sorting algorithm is to write a function or procedure that sorts an array of data. Traditionally, sorting is taught as a collection of algorithms. Typically, a motivation for each particular approach is given, and then an algorithm is developed and analyzed. While the relationships between the various algorithms are usually discussed, they are not explicitly present in the algorithms developed. However, there is an alternate, object-oriented approach that is based on the notion of an *abstract sorter*.

Think of a sorter as an abstract machine, the purpose of which is to sort arrays of data. A machine is an object. Therefore, it makes sense that we represent it as an instance of some class. The machine sorts data. Therefore, the class interface contains a member function which sorts an array of data.

The pedagogic benefit of organizing abstract sorters in a class hierarchy, is that the relationships between the various sorting algorithms is explicit. Furthermore, by using derivation and inheritance, we are able to demonstrate clearly the underlying commonalities between the various sorting algorithms.

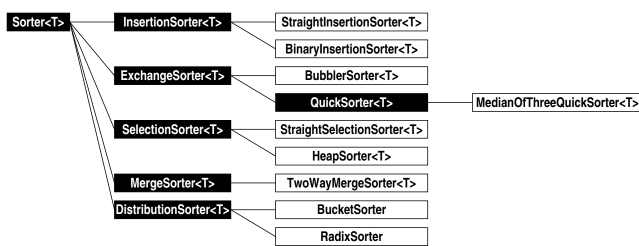

Figure 4: Sorter class hierarchy

As shown in Figure 4, the sorting algorithms normally discussed in the second course can be organized into a class hierarchy. Indeed the abstract sorters of this hierarchy correspond to way in which sorting algorithms have been traditionally categorized[3].

For example, consider the *quicksort* algorithm:

1. Select a pivot; and then
2. Using the pivot split the data to be sorted into two sublists and then recursively quicksort the sublists.

As shown in Figure 4 we can define the abstract class `AbstractQuickSorter` and the concrete class `MedianOfThreeQuickSorter`. The abstract class defers step 1 to the derived (concrete) class but implements step 2. This approach explicitly captures the notion that, after having somehow selected a pivot, all quicksorters are the same. The use of the class hierarchy makes this plainly evident.

3.2 Abstract Trees and Tree Traversals

As shown in Figure 1, we may view a *tree* as a specialized container. The C++ code given in Figure 5 illustrates how a container is specialized to obtain a tree by augmenting the generic container interface with functions specifically defined for manipulating trees. Specifically, every tree provides the following functions:

`Key` returns the object (key) in the root node of this tree;

`Subtree` returns the specified subtree of this tree;

`IsEmpty` returns true if this tree is the empty tree;

`Degree` returns the degree (number of subtrees) of this tree; and,

`DepthFirstTraversal`

```
class Tree : public virtual Container
{
public:
    virtual Object& Key () const = 0;
    virtual Tree& Subtree (unsigned int) const = 0;
    virtual bool IsEmpty () const = 0;
    virtual unsigned int Degree () const = 0;
    virtual void DepthFirstTraversal (Visitor&) const;
    // ...
};
```

Figure 5: Tree class

Many algorithms involve the systematic visiting one-by-one the keys contained in the nodes of a tree. Visiting all the nodes in a tree is called a tree traversal. The common tree traversals are *depth-first traversal* and *breadth-first traversal*.

For example, using the abstract tree interface described above we can write an abstract depth-first tree traversal function shown in Figure 6. Notice how the depth-first traversal routine is based on the *visitor* design pattern—it takes an abstract visitor and causes it to systematically visit all the keys in this tree.

```
void Tree::DepthFirstTraversal (Visitor& visitor) const
{
    if (visitor.IsDone ())
        return;
    if (!IsEmpty ())
    {
        visitor.PreVisit (Key ());
        for (unsigned int i = 0; i < Degree (); ++i)
            Subtree (i).DepthFirstTraversal (visitor);
        visitor.PostVisit (Key ());
    }
}
```

Figure 6: Depth-first tree traversal

The principal pedagogic benefit of this approach is that the notion of tree traversal is presented without the need to consider explicitly the representation of a tree. In fact, the algorithm given in Figure 6 works for all of the trees in the class hierarchy. Furthermore, by using a visitor we remove the application-specific activities from the traversal routine. What remains is the *essence* and nothing more.

3.3 Abstract Graphs and Graph Traversals

As shown in Figure 1, a *graph* as a specialized container. Specifically, we view a graph as a container that holds two distinct kinds of objects—*vertices* and *edges*.

The essence of many graph algorithms is a systematic visiting of either the edges or the vertices of a graph. For example, three of the most common graph traversal algorithms are *depth-first traversal*, *breadth-first traversal*, and *topological-order traversal* of the vertices of a graph.

By defining and using a generic graph interface it is possible to write abstract graph traversal algorithms in much the same way as the tree traversal algorithms described in the preceding section. In fact, the algorithms thus written are (not surprisingly) very similar to the corresponding algorithms for trees. The principal pedagogic benefit of this approach is that the notion of graph traversal can be presented without the need to consider explicitly the representation of a graph.

3.4 Abstract Solution Spaces and Problem Solvers

A backtracking algorithm finds the solution to a problem by systematically traversing the nodes of the solution space for that problem. In general, the solution space is an arbitrary graph. However, for many important applications the nodes form a tree (e.g., the 0/1 knapsack problem).

Figure 7 defines an abstract base class for representing the nodes of a solution space. By defining an abstract interface, it is possible to hide the details of the specific problem to be solved from the backtracking algorithm. In so doing, it is possible to implement completely generic backtracking problem solvers.

Each node of a solution space supports the following functions:

```
class Solution : public Object
{
public:
    virtual bool IsFeasible () const = 0;
    virtual bool IsComplete () const = 0;
    virtual int Objective () const = 0;
    virtual int Bound () const = 0;
    virtual Iterator& Successors () const = 0;
};
```

Figure 7: Abstract solution class

`IsFeasible` returns true if the partial solution that this node represents is a feasible solution;

`IsComplete` returns true if this partial solution is in fact a complete solution;

`Objective` returns the value of the objective function for this partial solution;

`Bound` returns a lower bound on the objective function for all the complete solutions to the problem that are descendents of this partial solution; and,

`Successors` returns an iterator that enumerates the successors of this partial solution, i.e., all the nodes in the solution space which are adjacent to this node.

Given this abstract `Solution` interface, it is possible to develop a generic backtracking algorithm that systematically explores a solution space, looking for the feasible solution which minimizes (or maximizes) the value of the objective function.

The usual way to implement a backtracking algorithm is to write a function or procedure which traverses the solution space. However, there is an alternate, object-oriented approach that is based on the notion of an *abstract solver*. Think of a solver as an abstract machine, the purpose of which is to search a given solution space for the best possible solution. Since a machine is an object we represent it as an instance of an abstract the `Solver` some class shown in Figure 8.

```
class Solver
{
public:
    virtual Solution& Solve (Solution const&);
    // ...
};
```

Figure 8: Abstract solver class

Given that the solution space is a tree, a backtracking problem solver simply does a tree traversal. By separating representation of the problem from the implementation of the problem solver, it is possible to develop *abstract* backtracking solvers. For example, Figure 9 gives the implementation of a problem solver that does a depth-first traversal of the solution space. Furthermore, with only a relatively minor modification to the code, the solver can be transformed into an abstract branch-and-bound problem solver[4].

```
void DepthFirstSolver::Solve (Solution const& solution)
{
    if (solution.IsComplete ())
        UpdateBest (solution);
    else
    {
        Iterator& i = solution.Successors ();
        while (!i.IsDone ()) {
            Solution& successor = (Solution&) (*i);
            DoSolve (successor);
            delete &successor;
            ++i;
        }
        delete &i;
    }
}
```

Figure 9: Depth-first solver

4 Summary and Conclusions

The advent of object-oriented methods and the emergence of object-oriented design patterns enables a profound change in the pedagogy of data structures and algorithms. The examples presented in this paper show that the successful application of these techniques has important pedagogic benefits. That is, they give rise to a kind of cognitive unification: Ideas that are disparate and apparently unrelated come together when the appropriate design patterns and abstractions are used.

This paper espouses the view that in teaching the second course one should preach design patterns from day one:

- Speak the *vocabulary* of objects and design patterns.
- Use appropriate *abstractions* including algorithmic abstraction.
- Demonstrate common *design patterns*.
- Explicate *relationships* between classes.
- Exploit *commonalities* between classes.
- Illustrate the salient *differences* between classes.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science*. Computer Science Press, New York, NY, 1992.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[3] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[4] Bruno Richard Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley & Sons, 1999. 660 pp. ISBN 0471-24134-2.

[5] Allen B. Tucker, Bruce H. Barnes, Robert M. Aiken, Keith Barker, Kim B. Bruce, J. Thomas Cain, Susan E. Conry, Gerald L. Engel, Richard G. Epstein, Doris K. Lidtke, Michael C. Mulder, Jean B. Rogers, Eugene H. Spafford, and A. Joe Turner. *Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force*. ACM/IEEE, 1991.