Teaching Data Structure Design Patterns

Natasha Gelfand* — Dept. of Comp. Sci., Brown Univ., Providence, RI 02912, ng@cs.brown.edu

Michael T. Goodrich† — Dept. of Comp. Sci., Johns Hopkins Univ., Baltimore, MD 21218, goodrich@cs.jhu.edu

Roberto Tamassia* — Dept. of Comp. Sci., Brown Univ., Providence, RI 02912, rt@cs.brown.edu

Abstract In this paper we present an approach for teaching the Freshman-Sophomore introduction to data structures course (CS2) in a way that provides an introduction to object-oriented software engineering patterns in addition to the theory of data structures. We survey in this paper several design patterns and describe how they can be naturally integrated in the CS2 curriculum.

1 Introduction

One of the main advantages of object-oriented design is to encourage well-organized code development for building software that is reusable, robust, and adaptable (e.g., see [2,3,6]). Designing quality object-oriented code takes more than simply understanding the object-oriented design methodologies, however. It requires the effective use of these and other object-oriented techniques in powerful and elegant ways.

Design Patterns Software engineering researchers and practitioners are developing sets of organizational concepts for designing quality object-oriented software. These concepts, called *design patterns* [4], are frameworks that one might follow for producing object-oriented software that is concise, correct, and reusable. Such patterns are important, but probably neglected by most instructors in the introduction to data structures course (CS2) and usually not taught until the software engineering course. We briefly survey several object-oriented design paradigms in this paper, and we describe how these paradigms can be consistently integrated into the curriculum of CS2, teaching students how to design quality implementations of data structures. The design patterns we discuss include the following: *adapters*, *template method*, *comparators*, *decorators*, *iterators* and *enumerations*, *positions*, and *locators*. Incorporating these patterns doesn't require any major revisions to the CS2 curriculum, for, as we describe below, they fit in naturally with the discussions of several components of CS2. These patterns apply to a course taught in any object-oriented language, we will give our examples in Java.

Related Work Software engineers have used design paradigms, or *patterns*, for some time now. Even so, it wasn't until the recent, yet seminal, cataloging effort of the so-called "gang of four," Gamma *et al.* [4], that the subject of design patterns became a topic of study in its own right. The efforts of these four, and other software engineering researchers, have shown that design patterns can save development time and result in software that is robust and reusable.

At Brown University design patterns have been taught as early as the the introductory computer science course (CS1) [7]. The patterns presented included *state*, *proxy*, *chain of responsibility*, and *factory*. Most of the patterns we describe in this paper are also described in the book by Gamma *et al.*, including *adapters*, *iterators*, *template methods*, and *decorators*. Other patterns we describe here, including *positions*, *locators*, and *comparators*, and the way that they can be integrated in the CS2 curriculum, are included in the recent book by Goodrich and Tamassia [5].

In the remainder of this paper we describe these patterns and how they can naturally be included in the CS2 curriculum. We break the patterns into two groups: those that primarily act on classes and those that primarily act on objects.

2 Class Patterns

Many patterns act on classes. That is, they provide extra capabilities for a class of objects, which the designer of that class need not be directly aware of.

*The work of this author is supported by the U.S. Army Research Office under grant DAAH04–96–1–0013, and by the National Science Foundation under grant CCR–9423847.

†The work of this author is supported by the U.S. Army Research Office under grant DAAH04–96–1–0013, and by the National Science Foundation under grant CCR–9625289.

We describe some of these patterns in this section.

Adapter The *adapter* pattern adjusts methods from one class so they can be used to implement methods of another class. The adaptation is expected to be a simple one, involving what are essentially one-line method calls to implement each method. This sometimes also involves *not* using several of the methods from a more general class. This adaptation is commonly done in design of data structures when we want to implement a new data structure in terms of another data structure that has a similar functionality, but different interface. A natural place to introduce this pattern is in the discussion of implementation of stacks, queues, and double-ended queues (or *deques*).

The adapter pattern can naturally be included early in the CS2 curriculum, in the implementation of DequeStack class shown in Code Fragment 1. This class demonstrates how to adapt a deque class so that it can be used to implement the stack abstract data type (ADT). That is, if we have an implementation MyDeque of the deque ADT, then we can easily implement the Stack interface with the class DequeStack shown in Code Fragment 1. All the methods of DequeStack are essentially one-line calls to methods of the Deque interface, with the slight added complication of converting deque exceptions into stack exceptions.

```
public class DequeStack implements Stack {
  Deque D; // holds the elements of the stack
  public DequeStack() { D = new MyDeque(); }
  public int size() { return D.size(); }
  public boolean isEmpty() { return D.isEmpty(); }
  public void push(Object obj) { D.insertLast(obj); }
  public Object top() throws StackEmptyException {
    try { return D.lastElement(); }
    catch (DequeEmptyException err)
        { throw new StackEmptyException(); }
  }
  public Object pop() throws StackEmptyException {
    try { return D.removeLast(); }
    catch (DequeEmptyException err)
        { throw new StackEmptyException(); }
  }
}
```

Code Fragment 1: Implementation of the Stack interface by means of a deque.

Another useful application of the adapter pattern in design of data structures is to specialize the types of objects that are used by a general class. This allows us to design general data structures which can store objects of any type. We can then make the data structure type-safe by writing an adapter that only accepts objects of a certain type and then forward all calls to the generic class. We can use this kind of adapter, for example, to define an IntegerArrayStack class that adapts an array-based ArrayStack class so that the stack only stores Integer objects. Such a class can then be used to to avoid the extra typing and possible confusion associated with casting.

Template Method Often several algorithms have the same overall structure but differ in the actions they take at specific steps. For example, many algorithms have as a base a simple tree traversal, but differ in the actions they perform at the nodes of a tree. In such cases it is desirable to implement the algorithm only once and then specialize it for the different applications.

The design pattern that can be used in such situations is called *template method*. This pattern provides a class which implements a skeleton of an algorithm, and delegates the steps that will vary in different implementations to its subclasses.

Template methods can be introduced in CS2 during the discussion of tree and graph traversals. We can generalize different tree traversals, such as preorder and postorder visit, to one generic visit of the tree, called *Euler tour*, where we start by going from the root towards the left child viewing the edges of the tree as being walls that we always keep to our left. Each node, therefore, will be encountered three times by the Euler tour: from the left, from below, and from the right. Since all algorithms using an Euler tour will have the same general structure, we can define an abstract class BinaryTreeTraversal, shown in Code Fragment 2, which executes the traversal, but does not take any specific action when it encounters a node. Instead, it calls auxiliary methods which are left empty in the abstract class, but are defined in the subclasses of the traversal to perform some actions. For example, we can produce preorder and postorder traversals of the tree by performing an action when a node is encountered from the left and from the right respectively.

An alternative approach to the problem of generalized algorithms is to defer the specific actions to separate objects instead of the subclasses of the abstract class and use the *strategy* pattern [4] instead of the template method pattern.

Comparator Another useful pattern that acts upon a class is the *comparator* pattern, which is an instance of a more general *strategy* pattern. This pattern provides a class of objects that are used for comparing pairs of objects in a totally-ordered container. An alternative approach is to require that objects be able to compare themselves to one another, but there

```
public abstract class BinaryTreeTraversal {
    public void traverseNode(Position p) {
        left(p);
        traverseNode(tree.leftChild(p));
        below(p);
        traverseNode(tree.rightChild(p));
        right(p);
    }
    // specific actions to take will be defined here
    protected void left(Position p) {}
    protected void below(Position p) {}
    protected void right(Position p) {}
}
```

Code Fragment 2: Generalized Euler tour of a binary tree

are contexts in which this solution is not applicable. Often objects do not need to "know" how they ought to be compared, or there may be multiple comparison methods that will add unnecessary complexity to the interface of those objects. For example, for two-dimensional data, it is not clear whether we should use the first coordinate or the second as the primary comparison value (or some other rule altogether). Indeed, there are several contexts in geometric algorithms where we might want to dynamically switch between different comparison functions. Thus, the data structures that need to compare objects should not expect the objects to provide their own comparison rules, but instead delegate this task to a *comparator* object.

Comparators are most naturally introduced in CS2 during the discussion of comparison-based data structures, such as priority queues and dictionaries. For example, a priority queue Q that is designed with comparators in mind is initialized with a given comparator, which is then used by Q to compare two objects. We can even imagine the ability for a priority queue to be given a new comparator if the old one even becomes "out-of date". Thus, a programmer can write a general priority queue implementation that can work correctly in a wide variety of contexts (including some the programmer has probably not even thought about). Formally, a comparator interface provides methods, isLess, isLessOrEqual, areEqual, isGreater, and isGreaterOrEqual. We provide an example implementation of the Comparator interface in Code Fragment 3.

Decorator The final class pattern we discuss in this section is the *decorator* pattern. This pattern is used to add extra attributes or "decorations" to objects with a certain interface (one possible interface is shown in Code Fragment 4). The use of decorators is motivated by the need of some algorithms and

```
public class Lexicographic implements Comparator {
  // Assumes Point2D objects have getX() and
  // getY() methods for coordinates.
  public boolean isLess(Point2D a, Point2D b) {
    if (a.getX() == b.getX())
        return (a.getY() < b.getY());
    return (a.getX() < b.getX());
  }
  // other methods are implemented in a similar fashion
}
```

Code Fragment 3: An implementation of the Comparator interface for 2-dimensional points.

data structures to add extra variables or temporary scratch data to the objects that will not normally need to have such variables.

```
public interface Decorable {
    public void create (Object key, Object value);
    public Object destroy(Object key);
    public boolean has(Object key);
    public void set (Object key, Object value);
    public Object get(Object key);
    public Enumeration attributes();
}
```

Code Fragment 4: An interface for objects that support adding decorations. Here *key* is a reference to the new decoration.

```
if (v.get(VISITED) == Boolean.FALSE) {
    v.set(VISITED, Boolean.TRUE);
    visit(v);
}
```

Code Fragment 5: An example of vertex visit in depth-first search using a decoration to store whether the vertex has been explored.

Decorators can be introduced in the CS2 curriculum in the discussion of balanced binary search trees and graph algorithms. 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 as 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 search and breadth-first search, we can use the decorator pattern to store temporary information about whether a certain vertex of a graph has been visited (see Code Fragment 5). The decorator pattern can be used in conjunction with the *position* pattern described in Section 3.

3 Object Patterns

Other patterns act primarily on objects. We describe some of them in this section.

Iterator Often we are interested in accessing the elements of a collection in certain order, one at a time, without changing the contents of the collection, e.g. to look for a specific element or to sum the values of all its elements. An *iterator* is an object-oriented design pattern that abstracts the process of scanning through a collection of elements, one element at a time, without exposing the underlying implementation of the collection. A typical interface of an iterator will include methods isDone(), firstElement(), nextElement(), and currentElement(). This ADT allows us to visit each element in a collection in order, keeping track of the "current" element.

Iterators can be discussed in the CS2 curriculum as soon as elementary data structures introduced. Java provides the simplified "streamlined" version of the iterator pattern in its Enumeration interface. Any time several objects need to be *examined* by some class, they can be given to that class in an enumeration. It is often useful to be able to run through or *enumerate* all the objects in a particular collection, so it can be useful to require all collections to support a method for returning their elements in an enumeration. Some collections, such as trees, are not linearly ordered, and there may be several different ways to enumerate their elements (e.g. preorder and postorder traversal). Using enumerations to traverse collections does not require knowledge of the internal details of how the collection is implemented. For example, one may wish to write a generic printCollection method, shown in Code Fragment 6, that can print out the contents of a collection of objects.

```
public printCollection(Collection c) {
    Enumeration enum = c.elements();
    while (enum.hasMoreElements()) {
        System.out.println(enum.nextElement());
    }
}
```

Code Fragment 6: An example of using an enumeration.

When it is created, an Iterator or Enumeration object may or may not be a "snapshot" of the collection at that time, so it is not a good idea to use iterator objects while modifying the contents of a collection.

Position In order to abstract and unify the different mechanisms for storing elements in various implementations of data structures, we introduce the concept of *position* in the data structure, which formalizes the intuitive notion of "place" of an element in a collection. A collection then stores its elements in positions and keeps the positions arranged in some specific order. The Position interface, then, provides methods for accessing the element stored at that position and the collection that the position belongs to.

Some examples of positions are nodes in such data structures as sequences and trees. Usually, the nodes are a part of the implementation of these data structures and therefore are not visible to the user. In an array-based implementations of sequences, there are no nodes, so positions are represented by the array indices. The position pattern provides a uniform interface for different implementations of positions in various data structures and makes the positions part of the interface of a data structure. For example, we can provide a method insertAfter(Position p, Object element) in the interface of a sequence that allows us to insert a new element into the sequence immediately after a given position (node).

A big advantage of being able to refer to individual positions is that it allows us to perform several operations on collections more efficiently. For example, given an implementation of a doubly linked list with nodes that have *next* and *prev* pointers, we can insert an arbitrary node v in $O(1)$ time, provided we are given a reference to the node preceding (or following) v. We can just "link in" the node new by updating its *next* and *prev* references, as well as those of its neighbors. Some possible places in the CS2 where positions can be introduced include the discussions of sequences and binary trees, where positions abstract the concept of nodes, and discussion of graphs where positions represent vertices and edges. Positions can be used in conjunction with the decorator pattern discussed in Section 2, since positions are the objects to which decorations can be added (in fact Position interface can extend Decorable). Not all data structures support a natural notion of position, however, and for those structures we can use the pattern we discuss next.

Locator The Position interface, described above, allows us to identify a specific "place" that can store an element. The element at some position can change, but the position stays the same. However, just having positions is not enough. When discussing in CS2 priority queues, dictionaries, and (in a fast-paced course) Dijkstra's shortest path algorithm, there are applications where one needs to keep track of elements as they are being moved from position to position inside a collection. In order to keep track of the location of each such object in an object-oriented manner, we need an abstraction of "location" that follows an element around, rather than being associated with a fixed position. This need is particularly important for data structures where there is no real concept of "positions" in the structure (e.g., key-based structures). A simple design pattern that fulfills this need is the *locator*.

The Locator interface is a simple ADT that abstracts the location of a specific element in a collection. A locator "sticks" with is associated element as long as that element remains in the collection, i.e., a locator remains valid until its associated element is removed or replaced. Like the Position ADT, the Locator ADT supports a method for returning its element. Even though they both support such a method, the Locator and Position interfaces are actually complements of each other: a Locator object stays with a specific element, even if it changes from position to position, and a Position object stays with a specific position, even if it changes the elements it holds. A locator is therefore like a coat check: we can give our coat to a coat room attendant, and we receive back a coat check, which is a "locator" for our coat. The position of our coat relative to the other coats can change, as other coats are added and removed, but our coat check can always be used to retrieve our coat. Like a coat check, then, we can now imagine getting something back when we insert an element into a collection: we can get back a locator to that element. This in turn can then be used to provide quick access to the position of this element in the collection, to say, remove this element or replace it with another element.

We can use locators in a very natural way in the context of a priority queue. A locator in such a scenario stays attached to an element inserted in the priority queue and allows us to refer to the element and its key in a generic manner that is independent from the specific implementation of the priority queue. This ability is important for a priority queue implementation, for there are no positions *per se* in a priority queue, since we do not refer to elements or keys by any notions of "rank," "index," or "node." By using locators, we can define methods for a priority queue Q that refer to elements stored in Q in a way that abstracts from the specific implementation of Q. Such methods include remove(ℓ), which removes the element with locator ℓ, and replaceKey(ℓ, k), which changes to k the priority of the element with locator ℓ. For example, Code Fragment 7, shows two fragments from an implementation of Dijkstra's algorithm in Java. The first fragment inserts a vertex u into a priority queue Q, using u's distance as its key, and associates with u the locator returned by Q (e.g., storing the locator as a decoration of u). The second fragment shows the relaxation of edge (u, z), and the update of the priority of vertex z in Q, which is performed with operation replaceKey.

```
Locator u_loc = Q.insert(new Integer(u_dist), u);
setLocator(u, u_loc);
...
if ( u_dist + e_weight < z_dist ) // relaxation
    Q.replaceKey(z_loc, new Integer(u_dist + e_weight));
```

Code Fragment 7: Fragments from the implementation of Dijkstra's algorithm

4 Conclusion

In this paper we survey a number of useful object-oriented software design patterns and describe natural places where they can be introduced in the standard curriculum for the Freshman-Sophomore data structures course (CS2). We summarize our suggestions in Table 1. We feel that introducing such design principles early in the computer science curriculum helps students form a framework for engineering software that will complement the theoretical foundation they receive in CS2.

Design Pattern	CS2 Topic
adapters	stacks and queues
template methods	tree and graph traversals
comparators	priority queues
decorators	balanced trees, graphs
iterators	sequences, trees, graphs
positions	sequences, binary trees, graphs
locators	priority queues, dictionaries

Table 1: Some design patterns and natural places in the CS2 curriculum where they can be introduced.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 1994.

[3] T. Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, 1991.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley and Sons, 1997.

[6] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. The MIT Press/McGraw-Hill, 1986.

[7] *Computer Science 15 Homepage*, Brown University. http://www.cs.brown.edu/courses/cs015