Patterns for Decoupling Data Structures and Algorithms

Dung ("Zung") Nguyen
Dept. of Mathematics/Computer Science
Pepperdine University
Malibu, CA 90263
dnguyen@pepperdine.edu

Stephen B. Wong
Computer Science Program
Oberlin College
Oberlin, OH 44074
stephen.wong@oberlin.edu

Abstract

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. To illustrate such a construction, this paper proposes a uniform object-oriented structural pattern for recursive data structures, and shows how external algorithms can be added without rewriting any code using the visitor design pattern. By presenting data structures in this manner, we can more effectively teach students about recursion, abstraction, design, and good software engineering practices.

1 Introduction

It is common practice to present data structures to students in some encapsulated form with a predefined set of operations. Unfortunately, even on common structures such as lists and trees, the set of operations differs from text to text. This arises because of the lack of delineation between the intrinsic and primitive behavior of the structure itself and the more complex behavior needed for a specific application. It thus becomes difficult for students to understand the proper abstraction of any particular structure.

The failure to decouple primitive and non-primitive operations also causes reusability and extensibility problems. The weakness in bundling a data structure with a predefined set of operations is that it presents a static non-extensible interface to the client that cannot handle unforeseen future requirements. Reusability and extensibility are more than just aesthetic issues; in the real world, they are driven by powerful practical and economic considerations. Computer science students should be conditioned to design code with the knowledge that it will be modified many times. In particular is the need for the ability to add features *after* the software has been delivered. An object-oriented approach to address this issue proceeds as follows.

i. Identify the variant and the invariant behaviors.
ii. Encapsulate the invariant behavior into a class.
iii. Add hooks to this class to define communication protocols with other classes.
iv. Encapsulate the variant behaviors into classes that comply with the above protocols.

The result is a flexible system of co-operating objects that is not only reusable and extensible, but also easy to understand and maintain.

This paper illustrates the above process by applying it to the design of recursive data structures and their algorithms. Here, the invariant is the intrinsic and primitive behavior of a structure, and the variants are the multitude of extrinsic and non-primitive algorithms that manipulate it. The recursive structures are implemented using the state design pattern [2] and encapsulated with a minimal and complete set of primitive structural operations. The visitor design pattern [1] is then applied to define the protocols for operating on the structure. Algorithms are simply implemented as visitors that can be sent to the appropriate structures for execution. A visitor only interacts with a structure via the structure's public interface. This approach turns a data structure into a (miniature) framework where control is inverted: one hands an algorithm to the structure to be executed instead of handing a structure to an algorithm to perform a computation. The structure is capable of carrying out any conforming algorithm, past, present, or future. This is how reusability and extensibility is achieved.

This paper also maintains that presenting data structures and algorithms in this manner makes it easier for students to implement data structures and develop algorithms, in particular recursion. It promotes thinking at the proper levels of abstraction, and exposes students to good design and software engineering practices.

Section 2 introduces the combination of state and visitor patterns for a linear recursive structure. Section 3 compares the binary tree structure with the linear recursive structure, and shows, abstractly, they are essentially the same. Section 4 describes this combination of state and visitor patterns with respect to general recursive structures and algorithms, and discusses its pedagogical implications. Section 5 demonstrates how the design easily accommodates the more complicated example of a binary search tree structure.

2 Linear Recursive Structure

A linear recursive structure (LRS) can be empty or non-empty. If it is empty, it contains no object. Otherwise, it contains an object called *first*, and a LRS object called *rest*. We model a LRS as a class in Java as follows.

```
public class LRStruct {
  public LRStruct() {…}
  public final Object getFirst () {…}
  public final void setFirst (Object dat) {…}
  public final LRStruct getRest () {…}
  public final void setRest (LRStruct tail) {…}
  public final void insertAsFirst (Object dat) {…}
  public final Object removeFirst () {…}
// other attributes and methods…  }
```

The constructor is to initialize a LRS to the empty state. The final keyword is used to express the fact that a method is an *invariant*. The get/set methods are for accessing and replacing the two different components (*first* and *rest*) of the LRS. insertAsFirst() corresponds to “cons” in Lisp. It allows the LRS to change state from empty to non-empty. removeFirst() is the (left) inverse of insertAsFirst(). It provides a way for the LRS to change state from non-empty to empty. This pair of methods defines the *state transitions* of a LRS. These seven methods expose the structure of a LRS to the client and constitute the pure structural behavior of a LRS. They form a minimal and complete set of methods for manipulating a LRS. Using them, one can create an empty LRS, store data in it, and remove/retrieve data from it at will.

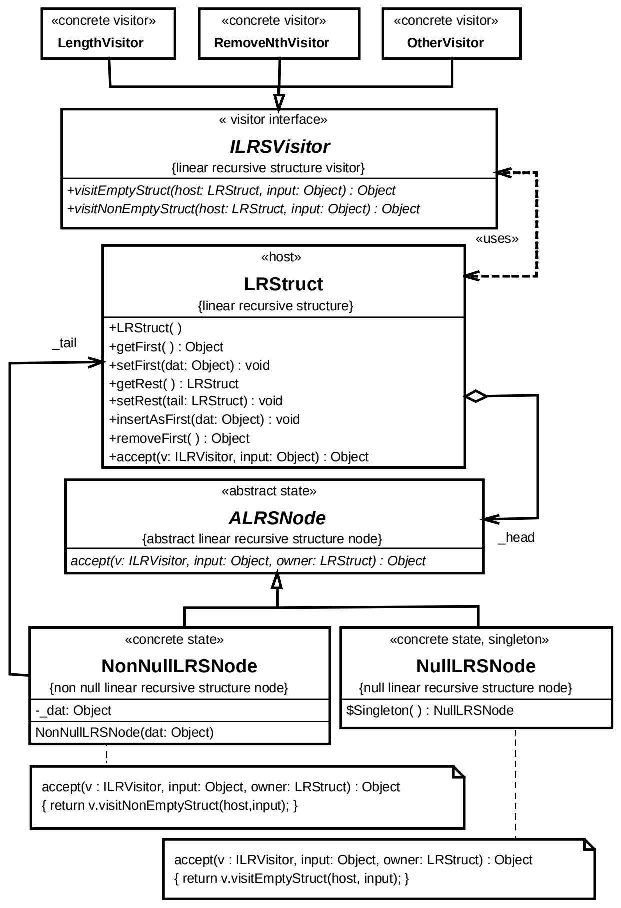
To endow a LRS the capability to perform an open-ended number of algorithms, we apply the visitor design pattern [1] by adding the following method to LRStruct:

```
public final Object accept (ILRSVisitor v, Object input),
```

where ILRSVisitor is an interface defined as

```
public interface ILRSVisitor {
  Object visitEmptyStruct (LRStruct host, Object input);
  Object visitNonEmptyStruct (LRStruct host, Object input);}
```

The visitor pattern decouples the structure from the extrinsic algorithms that operate on the structure. The pattern works as if a LRStruct announces to the outside world the following protocol:
“If you want me to execute your algorithm, encapsulate it into an object of type ILRSVisitor, hand it to me together with its input object as parameters for my accept(). I will send your algorithm object the appropriate message for it to perform its task, and return you the result.
visitEmptyStruct() should be the part of the algorithm that deals with the case where I am empty. visitNonEmptyStruct() should be the part of the algorithm that deals with the case where I am not empty.”



«concrete visitor» LengthVisitor
«concrete visitor» RemoveNthVisitor
«concrete visitor» OtherVisitor
« visitor interface» ILRSVisitor {linear recursive structure visitor}
+visitEmptyStruct(host: LRStruct, input: Object) : Object
+visitNonEmptyStruct(host: LRStruct, input: Object) : Object
«uses»
«host» LRStruct {linear recursive structure}
+LRStruct( )
+getFirst( ) : Object
+setFirst(dat: Object) : void
+getRest( ) : LRStruct
+setRest(tail: LRStruct) : void
+insertAsFirst(dat: Object) : void
+removeFirst( ) : Object
+accept(v: ILRVisitor, input: Object) : Object
_tail
_head
«abstract state» ALRSNode {abstract linear recursive structure node}
accept(v: ILRVisitor, input: Object, owner: LRStruct) : Object
«concrete state» NonNullLRSNode {non null linear recursive structure node}
-_dat: Object
NonNullLRSNode(dat: Object)
«concrete state, singleton» NullLRSNode {null linear recursive structure node}
$Singleton( ) : NullLRSNode
accept(v : ILRVisitor, input: Object, owner: LRStruct) : Object { return v.visitNonEmptyStruct(host,input); }
accept(v : ILRVisitor, input: Object, owner: LRStruct) : Object { return v.visitEmptyStruct(host, input); }

Figure 1: Linear recursive structure using state and visitor patterns.

Figure 1 depicts the above design in UML (Unified Modeling Language) notation. In this pattern, LRStruct is called the *host*, LRStruct.accept() is called the *hook* to the host, and all concrete implementations of ILRSVisitor are called *visitors*. A visitor interacts with the host only via the host’s structural behavior (public interface), and need not know how the host is implemented. The host intrinsically knows its own state, and is thus capable of making the appropriate callback on the visitor’s method.

How the host invokes the callback depends on how it is implemented. Nguyen [2] showed a representation of recursive structures using the state design pattern. In this particular implementation, an abstract class, ALRSNode, represents the state of a LRStruct. Two concrete subclasses of ALRSNode, NullLRSNode and NonNullLRSNode, represent the empty and non-empty states, respectively.

LRStruct maintains a reference to a ALRSNode object, say _head, and forwards all method calls to _head, in particular, the call to accept(). At run time, _head can be a NullLRSNode or a NonNullLRSNode. If it is a NullLRSNode, it calls visitEmptyStruct(), otherwise it calls visitNonEmptyStruct().

In Figure 1, LengthVisitor is the algorithm to compute the length of a LRStruct. OtherVisitor represents any other algorithm operating on LRStruct. RemoveNthVisitor is an example of a concrete implementation of ILRSVisitor. Its task to remove the n^{th} element from a LRS. It is defined as follows.

```
public class RemoveNthVisitor implements ILRSVisitor {
  public Object visitEmptyStruct (LRStruct host, Object input) {
      return null;  // the empty structure has no element.  }

  public Object visitNonEmptyStruct (LRStruct host, Object input) {
      int n = ((Integer) input).intValue();
      if (n < 0)  return null;  // there is no such n-th element.
      else if (0 == n)   return host.removeFirst();
      else  return (host.getRest()).accept(this, new Integer(n-1));  }}
```

Here, the RemoveNthVisitor class knows as a pre-condition the parameter input is of type Integer and can safely perform a type cast on input. RemoveNthVisitor accomplishes its task strictly through the public interface of LRStruct. Removing a node is an *extrinsic* operation. The recursive call, host.getRest().accept(), is easier to explain to students than the traditional recursive function calls. It is straightforward, clear and unambiguous because the host's *rest* is simply another LRStruct.

What we have is a framework for reusing a LRS. Here, the LRS is the unchanging entity from application to application (or over time). It is the particular algorithm used on the LRS that changes. The structure "drives" the algorithm and not the other way around. Presenting the LRS this way helps students focus on learning about the structure per se and appreciate what the abstraction of the structure is about. Building algorithms as separate entities more closely mimics the way that algorithms are discussed, thus is more "natural" and easier to understand. This is particularly true with recursive algorithms. To think about recursion is to think about the pattern of the base case (the empty case) *vs*. the non-base case (the non-empty case). The visitor interface not only provides students a clear picture of this pattern, but also distributes the code complexity over separate and independent methods, thereby simplifying the code, and making the overall complexity more manageable.

3 Binary Tree Structure

The above combination of state and visitor patterns can also be used to implement the binary tree structure (BiTS). A BiTS can be empty or non-empty. If it is empty, it contains no object. Otherwise, it contains an object called *root*, and a BiTS object called *left subtree*, and a distinct BiTS object called *right subtree*. This abstract definition of a BiTS is quite similar to that of a LRS. As in the case for a LRS, the class model for a BiTS consists of public methods that reflect the abstract view of the structure, and can be implemented using the state pattern [2].

```
public class BiTStruct {
  public BiTStruct () {…}
  public final Object getRootDat() {…}
  public final void setRootDat(Object dat) {…}
  public final BiTStruct getLeft() {   }
  public final void setLeft(BiTStruct t) {…}
  public final BiTStruct getRight() {…}
  public final void setRight(BiTStruct t) {…}
  public final void attachAsRoot(Object dat) {…}
  public final Object removeRoot() {…}
  public final Object accept(IBiTSVisiitor v, Object input) {…}
// other hidden methods…}
```

If the target BiTStruct is empty, attachAsRoot(dat) will change its state to non-empty, containing dat as the root, else nothing changes. removeRoot() changes the state of the target BiTStruct to empty. This pair of methods defines the *state transitions* of a BiTStruct. accept(), the visitor hook into the BiTStruct is identical to that of a LRStruct, save the appropriate change in the visitor type.

The visitor interface, IBiTSVisitor, is the same as that of a LRSVisitor, save the obvious change in the host type.

```
public interface IBiTSVisitor {
  public Object visitEmptyStruct (BiTStruct host, Object input);
  public Object visitNonEmptyStruct (BiTStruct host, Object input);}
```

Since both the BiTS and the LRS share the same state-visitor pattern, their underlying code is all but identical. For students, this emphasizes the abstract notions of recursive structures and their respective algorithms instead of confusing them with mountains of seemingly dissimilar code. To illustrate code structure similarity, we implement an algorithm, RemoveNMthVisitor, to remove a node from a BiTS, and compare it with the algorithm RemoveNthVisitor for a LRStruct.

There are many schemes for indexing the nodes of a binary tree, all of which are extrinsic to the tree structure. We use the following scheme described in [3]. A node's index consists of a pair of integers, *(n, m)*, where

- *n* denotes the depth of the node, and
- *m* is an integer where each bit starting from the least significant bit denotes whether a traversal to the right (1) or left (0) branch was taken to arrive at the node.

```
public class RemoveNMthVisitor implements IBiTSVisitor {
  public Object visitEmptyStruct (BiTStruct host, Object input)  {
      return null;  }
  public final Object visitNonEmptyStruct (BiTStruct host,
                                                        Object input) {
      BiTSIndex ix = (BiTSIndex) input;
      if (ix.n < 0) return null;
      else  if (0 == ix.n) {
        if (0 == ix.m) return host.removeRoot();
        else return null;       }
      else {
        if (1 == (ix.m&1)) return host.getRight().accept(this,
                                                new BiTSIndex(ix.n-1, ix.m/2));
       else return host.getLeft().accept(this,
                                              new BiTSIndex(ix.n-1, ix.m/2)); }
 }}
```

We see that since the depth transversal of a tree is the same as that through a linear structure, the code involving *n* is identical to the LRS code. The change to a binary topology results in a conditional statement, and the remaining returns and recursive call(s) are essentially identical as well. (Note: BiTSIndex is a class that holds both *n* and *m*.)

4 Recursive Structure Pattern

Traditionally, part of the code in a recursive algorithm is devoted to distinguishing between the base and the non-base cases. Students are often confused as they try to wend their way through the maze of conditional statements in many procedurally based algorithms. As illustrated in the preceding sections, the coding can be greatly simplified by decoupling the LRS and BiTS recursive structures from their respective extrinsic algorithms using a combination of state and visitor patterns. In general, this state-visitor pattern can be applied to any type recursive structure and its corresponding algorithms as shown in Figure 2. The public interfaces of a BiTS and of a LRS look very similar. They are special cases of a more general recursive structure pattern. In the design of recursive data structures and the development of their respective algorithms, this pattern enables the student to concentrate on the three building blocks of recursion, the base case, the non-base case, and the recursive call, and not be distracted by the details of control structures. This helps reduce confusion.

As shown in Figure 2, a recursive structure (RStruct) will have one or more base states, and one or more non-base states. In each of the states, the structure is composed of zero or one piece of data and zero or more children RStruct. Its public structural behavior consists of get/set methods to access the objects of its composition, and a changeStateTo() method for *state transitions*. This structural behavior is intrinsic to RStruct and can be implemented using the state pattern [2].

Because it is the recursive property of RStruct that gives rise to recursive algorithms, the visitor pattern prescribes encapsulating the algorithms on RStruct into classes whose common public interface, IRSVisitor, consists of methods that mirror the *states* (but *not* the behavior) of RStruct. RStruct.accept() is a hook method used to execute any IRSVisitor. At any point in time, RStruct knows what state it is in, and thus can drive the visiting algorithm by invoking the appropriate method call to IRSVisitor. The decision of when to call the various cases of an algorithm is defined *only once*, at the time when RStruct is designed, not every time a new algorithm is written as is done traditionally.

The variant behaviors of a system are built upon the invariant behaviors. Isolating the invariant behaviors and encapsulating them into the structure provides global and uniform access to them. As a result, the code for the variant behaviors is simplified, more easily understandable, and more robust. This is a powerful example to students of the advantages of isolating and properly encapsulating the invariant behaviors of a system.

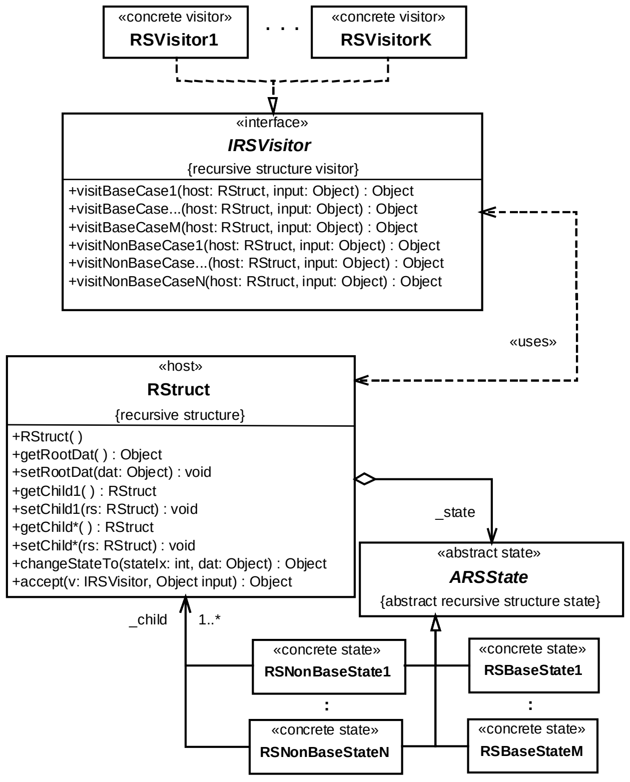

«concrete visitor»
RSVisitor1
. . .
«concrete visitor»
RSVisitorK
«interface»
IRSVisitor
{recursive structure visitor}
+visitBaseCase1(host: RStruct, input: Object) : Object
+visitBaseCase...(host: RStruct, input: Object) : Object
+visitBaseCaseM(host: RStruct, input: Object) : Object
+visitNonBaseCase1(host: RStruct, input: Object) : Object
+visitNonBaseCase...(host: RStruct, input: Object) : Object
+visitNonBaseCaseN(host: RStruct, input: Object) : Object
«uses»
«host»
RStruct
{recursive structure}
+RStruct( )
+getRootDat( ) : Object
+setRootDat(dat: Object) : void
+getChild1( ) : RStruct
+setChild1(rs: RStruct) : void
+getChild*( ) : RStruct
+setChild*(rs: RStruct) : void
+changeStateTo(stateIx: int, dat: Object) : Object
+accept(v: IRSVisitor, Object input) : Object
_state
«abstract state»
ARSState
{abstract recursive structure state}
_child
1..*
«concrete state»
RSNonBaseState1
«concrete state»
RSBaseState1
:
:
«concrete state»
RSNonBaseStateN
«concrete state»
RSBaseStateM

Figure 2: Generalized recursive structure using state and visitor patterns.

Reusability is achieved in both the data structure as well as the variant behaviors. All usages of a particular kind of data structure are based on the invariant behavior of that structure. Since the data structure object encapsulates its invariant behaviors, it is usable in any situation calling for it. Visitors are reusable because they can work with any instance of a data structure that presents the interface of the data structure for which they were built. Extensibility is achieved by the generality of the visitor-host protocol. A new visitor will work with a data structure as long as it presents methods for all the different states of the structure.

5 Binary Search Tree

A binary search tree (BST) can be built by reusing BiTStruct without modification and extending it with additional variant behaviors. A BST contains keyed objects whose keys are linearly ordered. So, suppose a client wants a BST with the capability to store, remove, and lookup data. We simply build a system consisting of (a) an interface, IkeyedObject, to specify the ordering strategy (b) concrete implementations of IBiTSVisitor for the store, remove, and lookup algorithms based on an ordering strategy (c) a BSTree class that wraps the BiTStruct to hide unneeded methods, exposing only the hook to accept visitors. Figure 3 shows the object model of the system.

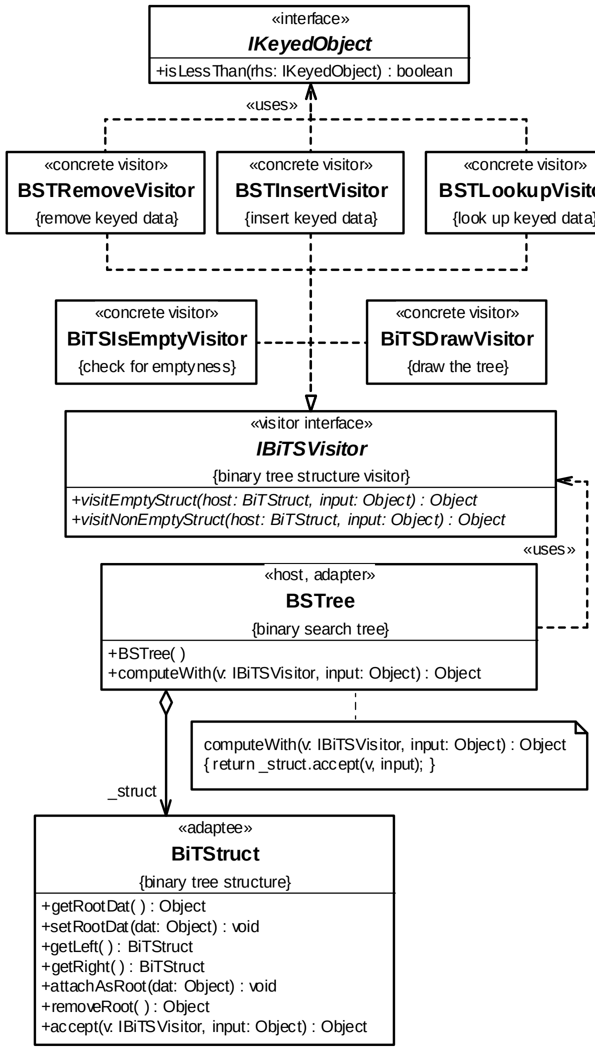


Figure 3: Binary search tree using visitors and the binary tree structure.

Suppose at a later time, the client wants to add to her BSTree the capability to check for emptiness, and to draw the tree. We can reuse the check empty and draw visitors that have already been built for BiTStruct but not used by this client, and add them to the client's system without changing or breaking any of the existing code. This is possible because these algorithms depend only on the invariant behavior of the binary tree structure. This demonstrates the reusability and extensibility of our BiTStruct and IBiTSVisitor setup.

6 Conclusion

In this paper we have discussed how data structures and their algorithms can be decoupled using the state and visitor design patterns. The intrinsic, invariant behaviors are encapsulated into an intelligent structure class. A hook is added, and a communication protocol with extrinsic algorithm classes is established. The extrinsic, variant behaviors on the data structure are encapsulated into visitor objects with a well-defined interface. The merits of this technique include enhanced reusability and unlimited extensibility. In addition, one finds that the coding is pleasantly simple. This does not happen by accident, for all the hard work has been shifted up front to formulate the appropriate abstraction for the problem at hand. Object-oriented design patterns provide ways to implement data structures in such a way that the gap between the abstraction and the computing model is minimal. Effective use of polymorphism eliminates most control structures and reduces code complexity. Data structures and algorithms built as such are elegant in their simplicity, universal in their applicability, and flexible in their extensibility.

It is important to teach students computer science concepts using solid software engineering practices. We maintain that using the state and visitor patterns to teach recursive data structures and their algorithms has tremendous benefits for students as well as technical value. The patterns force them to properly abstract the concepts by carefully delineating the variant and invariant behaviors. The students can focus on the key concepts and behaviors of the system without distraction from the complications of program control. Our proposed implementation simplifies their understanding of recursive data structures in general by emphasizing the invariant behaviors common to all types while discussing variant behaviors at their proper level of abstraction. This unification of their concepts and the resultant reduction in the traditional complexity of the subject will enable students to better understand the fundamentals of data structures and goals of quality software engineering.

7 References

1. Gamma, E, Helm, R, Johnson, R, Vlissides, J. *Design Patterns, Elements Of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

2. Nguyen, D. *Design Patterns for Data Structures*. SIGCSE Bulletin, 30, 1, March 1998, 336-340.

3. Wong, S. *Structure-encoded Indexing Schemes for Recursive Structures*, pre-publication, Oberlin College, stephen.wong@oberlin.edu.