Chapter 24

The State Design Pattern

The state design pattern is an object-oriented technique that uses inheritance and class composition. It is applicable to a variety of software design problems where an object needs to alter its behavior when its internal state changes. This chapter illustrates the state design pattern with implementations of a binary tree and a linked list.

Binary trees and linked lists have several things in common. Both structures are based on links between nodes. The state of each structure is defined as a pointer to its first node. That is, the state of a binary tree is defined as a pointer to its root node, while the state of a linked list is defined as a pointer to its head node. Furthermore, the definition of each data structure is inherently recursive. A binary tree is either empty or a pointer to a node that contains a value, a left tree, and a right tree. A list is either empty or a pointer to a node that contains a value and a list. These common properties are the basis of the state pattern implementation of the data structures.

Binary search trees

Figure 24.1 is the interface for a binary search tree implemented with the state design pattern. The methods should look familiar, as they are identical to the methods of the binary search tree class of Figure 22.9, which is also shown in the figure. Examine the interfaces for these two classes and you will find that they are identical in every detail except that the name of the module for the tree with the state design pattern is PboxTreeSta while that for the tree in Chapter 22 is PboxTreeObj.

Because the interfaces are identical, the programs that use them are identical as well. Rather than showing the dialog box that uses the tree with the state design pattern see Figure 22.10, which is identical. Rather than showing the program that implements the dialog box see Figure 22.11, which is identical except for the substitution of PboxTreeSta for every occurrence of PboxTreeObj. In all respects, a client module that uses the binary search tree implemented with the state design pattern is not aware of any difference between its behavior and that of the binary search tree as implemented in Chapter 22.

```
DEFINITION PboxTreeSta;

    TYPE
        T = ARRAY 16 OF CHAR;
        Tree = RECORD
            (VAR tr: Tree) Clear, NEW;
            (IN tr: Tree) Contains (IN val: T): BOOLEAN, NEW;
            (VAR tr: Tree) Insert (IN val: T), NEW;
            (IN tr: Tree) NumItems (): INTEGER, NEW;
            (IN tr: Tree) PreOrder, NEW;
            (IN tr: Tree) InOrder, NEW;
            (IN tr: Tree) PostOrder, NEW
        END;

END PboxTreeSta.


DEFINITION PboxTreeObj;

    TYPE
        T = ARRAY 16 OF CHAR;
        Tree = RECORD
            (VAR tr: Tree) Clear, NEW;
            (IN tr: Tree) Contains (IN val: T): BOOLEAN, NEW;
            (VAR tr: Tree) Insert (IN val: T), NEW;
            (IN tr: Tree) NumItems (): INTEGER, NEW;
            (IN tr: Tree) PreOrder, NEW;
            (IN tr: Tree) InOrder, NEW;
            (IN tr: Tree) PostOrder, NEW
        END;

END PboxTreeObj.
```

Figure 24.1
The interfaces for a binary search tree implemented with the state design pattern and as it is implemented in Chapter 22.

Figure 24.2 shows the UML diagram for the state design pattern applied to a binary search tree. The declarations of Tree and Node are

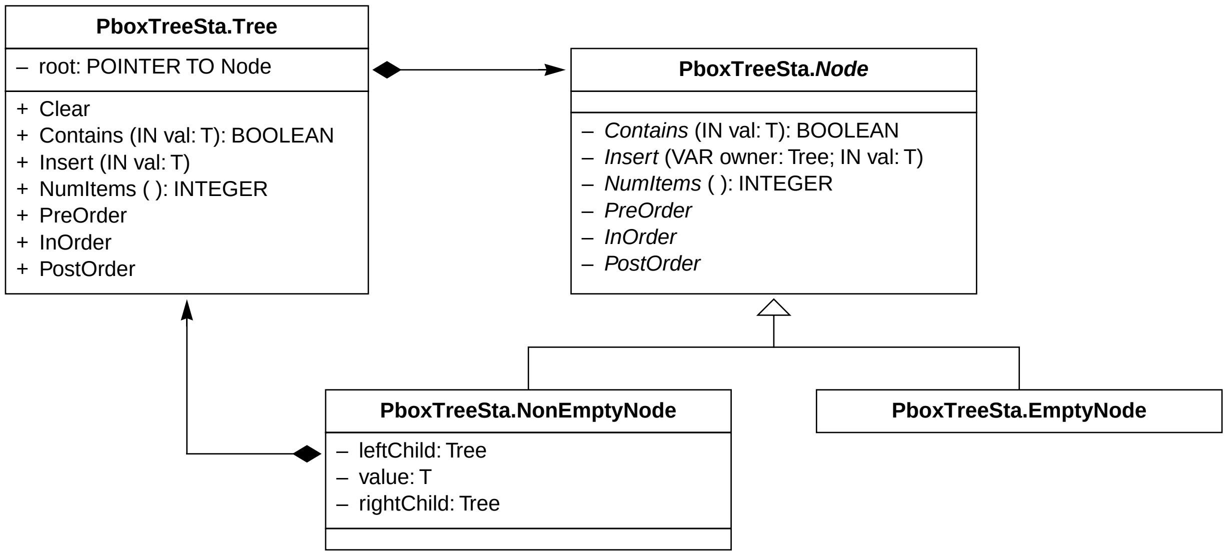
The data structure for a binary tree using the state design pattern

```
TYPE
    T* = ARRAY 16 OF CHAR;
    Tree* = RECORD
        root: POINTER TO Node
    END;

    Node = ABSTRACT RECORD END;
    EmptyNode = RECORD (Node) END;
    NonEmptyNode = RECORD (Node)
        leftChild: Tree;
        value: T;
        rightChild: Tree
    END;
```

The relation between a tree and an abstract node is class composition. A tree has a node. The relation between an empty node and an abstract node is inheritance. An empty node is an abstract node. The relation between a nonempty node and an abstract node is also inheritance. A nonempty node is an abstract node. The relation between a nonempty node and a tree is class composition. A nonempty node has two trees.

Figure 24.2
The UML diagram for a state design pattern implementation of a binary search tree.



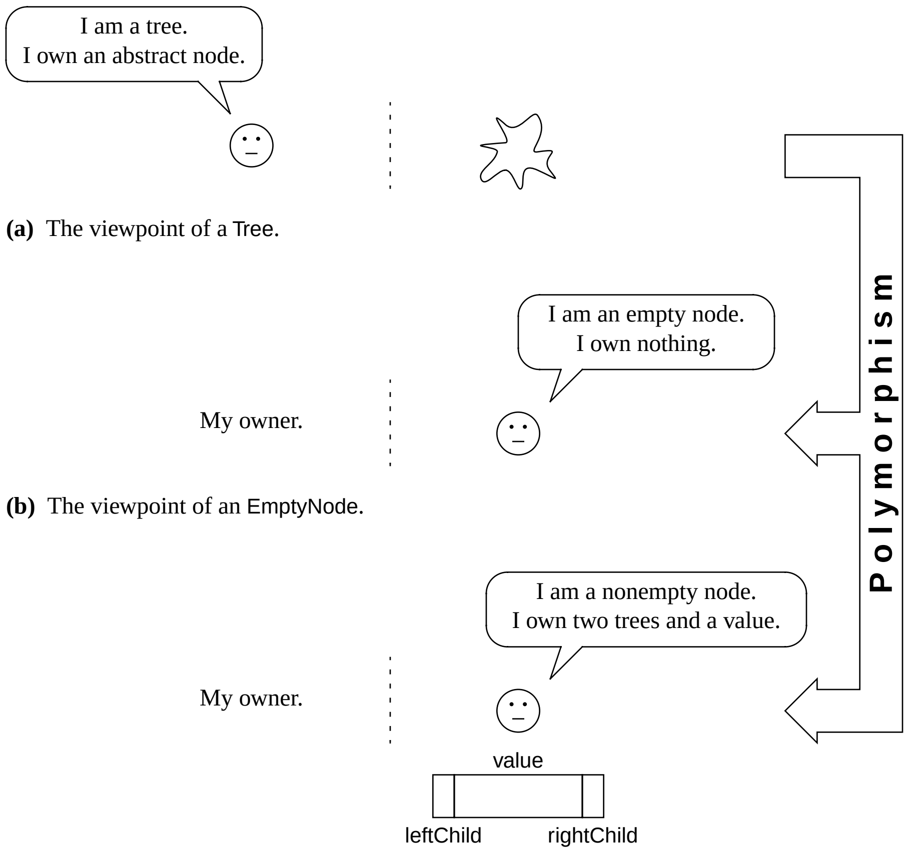
The module for the binary search tree contains the class for the tree as well as the class for the abstract node and each of its two subclasses. The details of the node structure are hidden from the client module by virtue of the fact that none of the node declarations are exported. In Figure 24.1, none of the details of the node or even the pointer to the head node are visible to the client. Unfortunately, many object-oriented languages are not based on the module concept, which is one of the great strengths of Component Pascal compared to them. If you had no way of packaging several classes into one module, you would need to export the methods of the abstract Node, making them public, so the Tree class could have access to them.

Object-oriented design is a collection of cooperating objects.

Class abstraction unifies attributes and operations. Because an instance of a class has both, it becomes an autonomous entity. You should think of OO design as a collection of cooperating objects, each one of which is autonomous. It sometimes helps to have an anthropomorphic view of the design in which each object is like an independent person who cooperates with the other people objects. Figure 24.3 is such a view for the state design pattern of the binary search tree.

The viewpoint of a tree

Figure 24.3(a) shows the viewpoint of a tree object. As far as the tree is concerned, he owns an abstract node. The tree looks through a window, represented by the dashed vertical line, and sees an abstract node, represented by the amorphous shape on the right side of the window. The tree does not know what kind of node he owns, that is, whether his node is empty or nonempty. The state of the tree is defined by its root. Figure 24.2 shows that root is a pointer to a Node. But type Node is abstract. That is, it can never be allocated. The only nodes that can be allocated are



(c) The viewpoint of a NonEmptyNode.

Figure 24.3
The cooperating objects in the state design pattern for the binary search tree.

its concrete subclasses, EmptyNode and NonEmptyNode.

The viewpoint of an empty node

Now consider the viewpoint of an empty node in Figure 24.3(b). An empty node has no attributes. Every node has an owner, but not in the same sense that the owner has a node. If myTree is a tree, then it can always refer to the node it owns by the expression myTree.root. But if myEmptyNode is an EmptyNode, there is no corresponding myEmptyNode.owner. Consequently, the empty node cannot look through the window directly to see his owner.

The viewpoint of a nonempty node

The viewpoint of a nonempty node is similar to that of the empty node. He cannot see his owner directly. However, he owns something that an empty node does not own, namely two trees and a value.

Delegation

The system of these three cooperating objects—trees, empty nodes, and nonempty nodes—works by delegation. Each autonomous object either knows how to perform a simple task itself or, if it cannot perform the task, delegates all or part of the task to another object. Typically, the client module gives a tree a task to perform by calling one of the tree's methods. The tree then delegates the task to its node.

For example, the PboxTree.Sta box in Figure 24.2 shows that a tree object has method NumItems(), which returns the number of items in the tree. Imagine that you are the tree in Figure 24.3(a). You are instructed to return the number of nodes contained in yourself. The problem is that you do not even know whether you are empty or not! The only thing you know is that you have an abstract node. When you look through the window at your node, you cannot tell what kind of node it is. You only see that abstract blob. Fortunately, the PboxTreeSta.Node box in Figure 24.2 shows that an abstract node also has a method named NumItems(). You own an abstract node, and your abstract node provides a method that will return the number of items in the tree. So, you simply call the method for your node.

NumItems for a tree

Now, which method executes? Certainly not the abstract method NumItems(), because an abstract method cannot be implemented. Figure 24.2 shows that EmptyNode and NonEmptyNode are concrete subclasses of the abstract class Node. They inherit from Node and each one implements its own version of NumItems(). The double-headed arrow on the right of Figure 24.3 indicates that polymorphic dispatch determines which of these two versions of NumItems() executes. You can see that polymorphism eliminates an IF statement here. As a tree, you do not know what kind of node you own, whether empty or nonempty. But you do not need to know or even to test what kind of tree you are with an IF statement. You simply delegate to your node the task of computing the number of items with a single call, which is polymorphically dispatched.

Polymorphism

Consider the implementation of the empty node's version of NumItems(). If you are the empty node in Figure 24.3(b), and you are the root of a tree, how many items do you contain? The answer should be obvious—none. Your version of NumItems() simply returns 0.

NumItems for an empty node

What about the implementation of the nonempty node's version of NumItems()? If you are the nonempty node in Figure 24.3(c) what do you return? That is, if you are the root of a tree, how many nodes does your tree contain? Remember, you own a value and two subtrees. So, the answer is one (yourself) plus the number of nodes in your left child plus the number of nodes in your right child.

NumItems for a nonempty node

Figure 24.4 shows the implementation of the binary search tree with the state design pattern. Implementation of some of the methods are left as problems for the student. Compare this tree class with that in Figure 22.14. As with that implementation, a Tree is a record that contains a single pointer to a node. With the state design pattern, however, the root of a tree is never NIL. It always points to something, either an empty node if the tree is empty or a nonempty node if it is not.

```
MODULE  PboxTreeSta;
    IMPORT StdLog;

    TYPE
        T* = ARRAY 16 OF CHAR;
        Tree* = RECORD
            root: POINTER TO Node
        END;
```

Figure 24.4
The implementation of the binary search tree with the state design pattern.

Figure 24.4
Continued.

```
        Node = ABSTRACT RECORD END;
        EmptyNode = RECORD (Node) END;
        NonEmptyNode = RECORD (Node)
            leftChild: Tree;
            value: T;
            rightChild: Tree
        END;

(* -------------------- *)
    PROCEDURE (VAR tr: Tree) Clear*, NEW;
        VAR
            p: POINTER TO EmptyNode;
    BEGIN
        NEW(p);
        tr.root := p
    END Clear;

(* -------------------- *)
    PROCEDURE (IN tr: Tree) Contains* (IN val: T): BOOLEAN, NEW;
    BEGIN
        (* A problem for the student *)
        RETURN FALSE
    END Contains;

(* -------------------- *)
    PROCEDURE (IN node: Node) Insert (VAR owner: Tree; IN val: T), NEW, ABSTRACT;

    PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    BEGIN
        tr.root.Insert (tr, val)
    END Insert;

    PROCEDURE (IN node: NonEmptyNode) Insert (VAR owner: Tree; IN val: T);
    BEGIN
        ASSERT(node.value # val, 20);
        IF node.value < val THEN
            node.rightChild.Insert(val)
        ELSE
            node.leftChild.Insert(val)
        END
    END Insert;

    PROCEDURE (IN node: EmptyNode) Insert (VAR owner: Tree; IN val: T);
        VAR
            p: POINTER TO NonEmptyNode;
    BEGIN
        NEW(p);
        p.leftChild.Clear;
        p.value := val;
        p.rightChild.Clear;
        owner.root := p (* Change the state of owner *)
    END Insert;
```

```
(* -------------------- *)
    PROCEDURE (IN tr: Tree) NumItems* (): INTEGER, NEW;
    BEGIN
        (* A problem for the student *)
        RETURN 999
    END NumItems;

(* -------------------- *)
    PROCEDURE (IN node: Node) PreOrder, NEW, ABSTRACT;

    PROCEDURE (IN tr: Tree) PreOrder*, NEW;
    BEGIN
        tr.root.PreOrder
    END PreOrder;

    PROCEDURE (IN node: EmptyNode) PreOrder;
    BEGIN
        (* Do nothing *)
    END PreOrder;

    PROCEDURE (IN node: NonEmptyNode) PreOrder;
    BEGIN
        StdLog.String(node.value); StdLog.String("  ");
        node.leftChild.PreOrder;
        node.rightChild.PreOrder
    END PreOrder;

(* -------------------- *)
    PROCEDURE (IN tr: Tree) InOrder*, NEW;
    BEGIN
        (* A problem for the student *)
    END InOrder;

(* -------------------- *)
    PROCEDURE (IN tr: Tree) PostOrder*, NEW;
    BEGIN
        (* A problem for the student *)
    END PostOrder;

END PboxTreeSta.
```

Figure 24.4
Continued.

Compare Figure 24.4 with Figure 22.14 and you will see that with the state design pattern there is no setting of any pointer to NIL, nor is there a comparison of any pointer to NIL. The concept of NIL is hidden at a lower level of abstraction with the state design pattern. There are fewer IF statements because polymorphic dispatch takes their place.

For example, method Clear from PboxTreeSta in Figure 24.4 declares p to be a local pointer to an empty node as

```
p: POINTER TO EmptyNode;
```

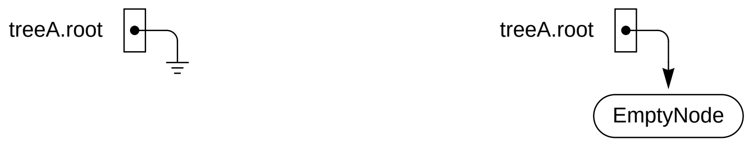
It clears tree tr by setting its root to a pointer to an empty node as follows.

```
NEW(p);
tr.root := p
```

Compare this with the corresponding implementation of Clear from PboxTreeObj in Figure 22.14.

```
tr.root := NIL
```

This version of Clear clears tree tr by setting its root to NIL. In contrast, there is no concept of NIL in the PboxTreeSta version. Figure 24.5 shows the difference between empty trees in these two versions. There is really nothing in the oval box labeled EmptyNode in Figure 24.5(b), because an empty node has no attributes.



(a) The empty tree from PboxTreeObj in Figure 22.14.

(b) The empty tree from PboxTreeSta in Figure 24.4.

Figure 24.5 The empty tree in PboxTreeObj and PboxTreeSta.

To see how the state design pattern eliminates the IF statement and the NIL value, consider the implementation of method PreOrder. In PboxTreeObj, the implementation of method PreOrder in Figure 22.14 begins with the statement

```
IF tr.root # NIL THEN
```

followed by the recursive calls to PreOrder in the body of the IF statement. But, the PreOrder implementation in PboxTreeSta has no IF statement and no reference to NIL. Imagine you are the tree in Figure 24.3(a). You do not know what kind of tree you are, whether empty or nonempty. Only your root node knows. But you cannot tell by looking at your root node, because it is abstract. All you see is that abstract blob. So, you delegate. The implementation of PreOrder for a tree is the one liner

```
tr.root.PreOrder
```

PreOrder for a tree

which is a call to the PreOrder method of a node. Which version of PreOrder gets called—the one for an empty node or the one for a nonempty node? Polymorphism decides with no IF statement or NIL test. The implementation of the empty node version is simply the comment

```
(* Do nothing *)
```

PreOrder for an empty node

The implementation of the nonempty version is an output of the nonempty node's value with

```
StdLog.String(node.value); StdLog.String("  ")
```

followed by the usual recursive calls to PreOrder for the left and right children. In the first recursive call,

PreOrder for a nonempty node

```
node.leftChild.PreOrder
```

node is the current nonempty node, and node.leftChild is its left child, which is a tree. Therefore, node.leftChild.PreOrder is a method call for a tree, not for a node.

The implementation of method Insert in PboxTreeSta shows the power of object-oriented programming with polymorphism. Consider implementation of the method from PboxTreeObj in Figure 22.14. It has a WHILE loop to determine the leaf where the new value is to be attached and tests for NIL all over the place. Contrast the complexity of that implementation with the simplicity of the one from PboxTreeSta in Figure 24.4. There are no loops, and there is only one simple IF statement that compares the value to be inserted with the value of the current nonempty node. Here is how it works.

The heading for the Insert method for a tree is

```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
```

Formal parameter tr is the tree into which the value is to be inserted, and val is the formal parameter of the value to insert. Imagine you are the tree in Figure 24.3(a). You have a value val to insert into yourself, but you do not even know what kind of tree you are, whether empty or nonempty. So, you delegate the task to your root node, which knows what kind of node it is. If your root node is nonempty, it will simply pass the request down to one of its children.

Insert for a tree

But, there is a slight complication if your root node is empty. In that case, your root node must change your state (hence, the name *state* design pattern). Your current state is empty, but after the insertion your state will be changed to nonempty. That is, your root attribute will need to point to a new nonempty node after the insertion rather than the empty node to which it currently points. Your empty node will need to change your root. So, you the owner of the node must pass yourself to your node so it can change your state. The heading for the Insert method for an abstract node is

Insert for an abstract node

```
PROCEDURE (IN node: Node) Insert (VAR owner: Tree; IN val: T), NEW, ABSTRACT;
```

Not only must the tree pass the value via parameter val to its node, it must also pass itself via parameter owner to its node. Formal parameter owner is passed by reference, because the method may use its value and change the corresponding actual parameter.

The heading for the Insert method for a nonempty node is

```
PROCEDURE (IN node: NonEmptyNode) Insert (VAR owner: Tree; IN val: T);
```

Figure 24.6 shows the perspective of a nonempty node object who owns a value and two children. The value it owns is robin. It does not know what kind of children it owns, because each child contains a root that points to an abstract node. It has access to its owner and to val through its parameter list. The figure assumes that sparrow is passed as val. The implementation of Insert for the nonempty node is simple. First, the statement

```
ASSERT(node.value # val, 20)
```

verifies that a duplicate value is not being inserted into the tree. Then,

```
IF node.value < val THEN
    node.rightChild.Insert(val)
ELSE
    node.leftChild.Insert(val)
END
```

executes, which tests if robin is less than sparrow. Because robin is indeed less than sparrow in alphabetical order, the nonempty node simply delegates the insertion task by passing sparrow to be inserted into its right child with the call

```
node.rightChild.Insert(val)
```

Because node.rightChild is a tree, the method call is for a tree with no owner in the parameter list.

The heading for the Insert method for an empty node is

```
PROCEDURE (IN node: EmptyNode) Insert (VAR owner: Tree; IN val: T);
```

Because the node is empty, a new nonempty node must take its place with the value from val in the new node's value field. Also, the left and right children of the new node must be empty trees. The code is straightforward as Figure 24.7 shows. Initially the empty node is in an environment that provides access to its owner, the value passed to it in parameter val, and the local variable p, as Figure 24.7(a) shows. When the statement

```
NEW(p)
```

executes in Figure 24.7(b), a nonempty node is allocated because p is declared to be a pointer to a nonempty node. Then,

```
p.leftChild.Clear
```

clears the left child in Figure 24.7(c),

```
p.value := val
```

puts the value from the parameter into the value part of the new nonempty node in Figure 24.7(d), and

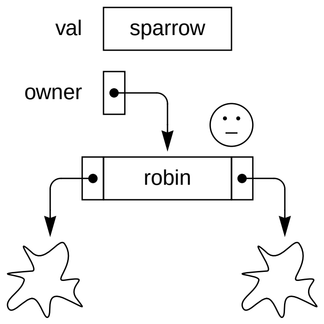
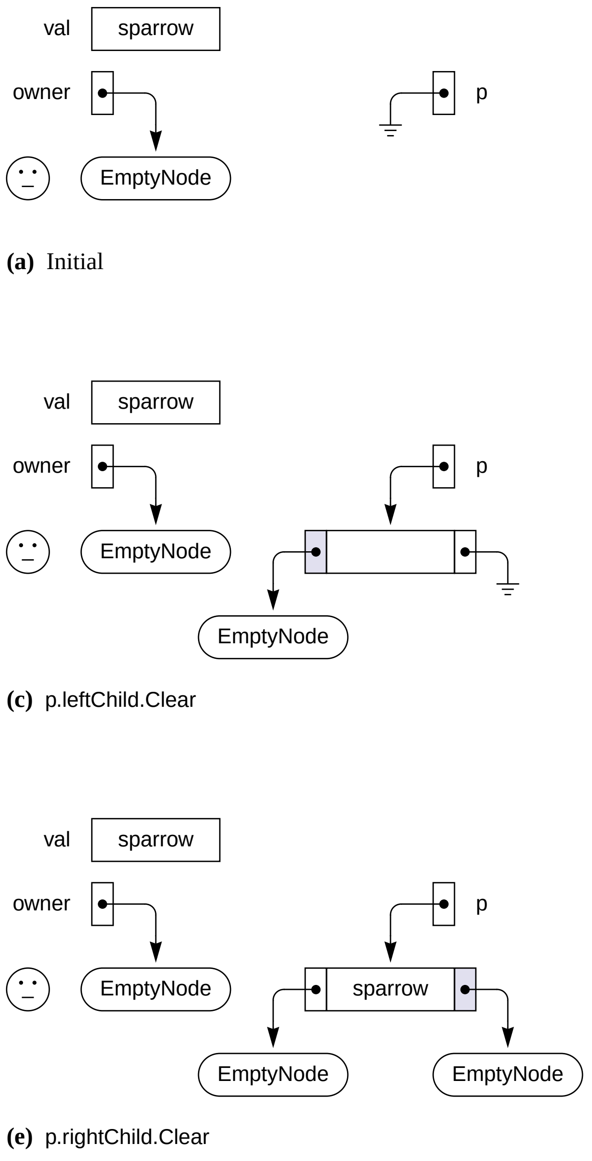


Figure 24.6
The viewpoint of a nonempty node in the environment of an



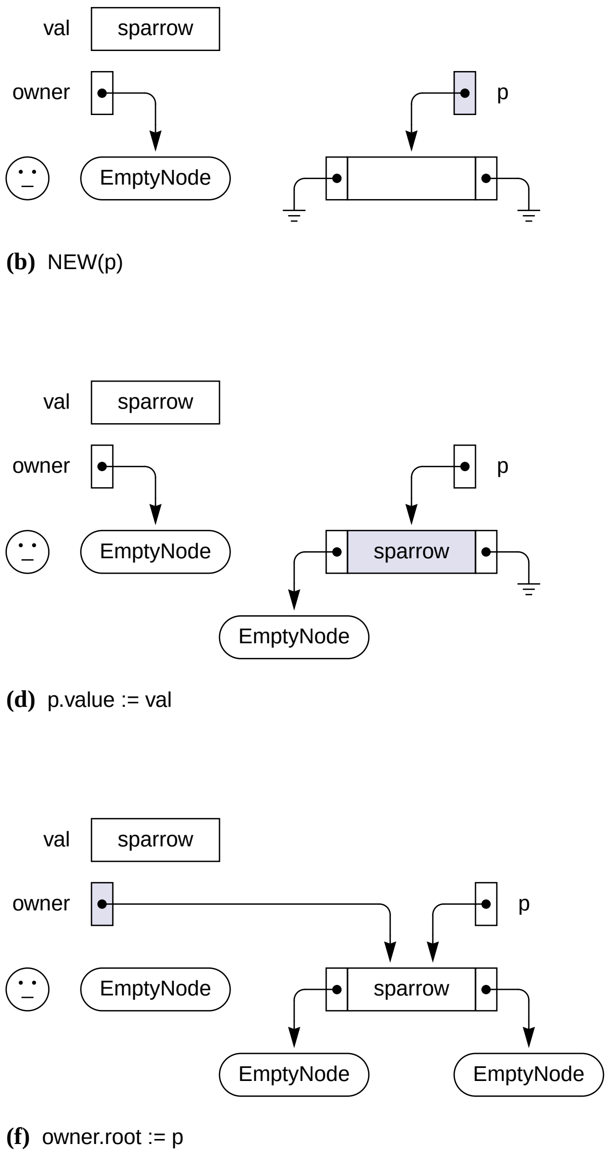


Figure 24.7 The viewpoint of an empty node in the environment of an Insert call.

p.rightChild.Clear

clears the right child in Figure 24.7(e). The last statement in the method

owner.root := p

shown in Figure 24.7(f) changes the state of the owner. Whereas the owner used to be an empty tree, it is now a nonempty tree.

A striking feature of method Insert for the state design pattern in Figure 24.7 compared to the original method illustrated in Figure 22.15 is the locality of the environment. Figure 22.15 shows how that algorithm must view the entire tree, keeping track of the parent of each node as it works its way to the proper leaf. But, the algorithm with the state design pattern is separated into three parts—one for the tree, one for a nonempty node, and one for an empty node. Each part is a separate method. The implementation of the tree method is a single line. The simplicity of the method is due to the fact that the environment of the tree is local. The tree does not even know what kind of tree it is because it cannot see past its abstract root. There is no larger picture of the tree as a whole. The implementation of the nonempty node method is a single IF statement. It cannot see past its children, because their roots are abstract. The nonempty node cannot see past its owner above or past its children below. The environment in which it accomplishes its task is strictly local. The same can be said for the empty node. It works in a local environment without even the concept of a parent.

Locality of the environment

In effect, the original insertion algorithm is distributed among three kinds of objects—one tree and two nodes—each acting in its own local environment. In each environment, abstraction hides the details from the other environments. The problem is subdivided into smaller problems in a natural way. Each smaller problem is easier to solve that the larger problem of which it is a part. A distributed algorithm using polymorphism in a system of cooperating objects is the hallmark of object-oriented thinking. An example of the utility of such an approach is programming for a network of computers. It is possible to have the different objects of the system exist on different computers in the network. In such an environment, the distribution of the algorithm is not just a logical construction among objects executing on the same computer, but is a literal distribution of objects executing on physically different computers.

Distributed algorithms

Linked lists

The state design pattern is a general technique not limited to binary trees. Figure 24.8 shows the interfaces for a linked list implemented with the state design pattern and for the linked list implemented in Chapter 21.

```
DEFINITION PboxLListSta;

   TYPE
      T = ARRAY 16 OF CHAR;
      List = RECORD
         (VAR lst: List) Clear, NEW;
         (IN lst: List) Display, NEW;
         (IN lst: List) GetElementN (n: INTEGER; OUT val: T), NEW;
         (VAR lst: List) InsertAtN (n: INTEGER; IN val: T), NEW;
         (IN lst: List) Length (): INTEGER, NEW;
         (VAR lst: List) RemoveN (n: INTEGER), NEW;
         (IN lst: List) Search (IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN), NEW
      END;

END PboxLListSta.
```

Figure 24.8
The interfaces for a linked list implemented with the state design pattern and as it is implemented in Chapter 21.

```
DEFINITION PboxLListObj;

   TYPE
      T = ARRAY 16 OF CHAR;
      List = RECORD
         (VAR lst: List) Clear, NEW;
         (IN lst: List) Display, NEW;
         (IN lst: List) GetElementN (n: INTEGER; OUT val: T), NEW;
         (VAR lst: List) InsertAtN (n: INTEGER; IN val: T), NEW;
         (IN lst: List) Length (): INTEGER, NEW;
         (VAR lst: List) RemoveN (n: INTEGER), NEW;
         (IN lst: List) Search (IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN), NEW
      END;

END PboxLListObj.
```

Figure 24.8
Continued.

There is no difference between these two interfaces other than the name of the module. Consequently, the program that uses PboxLListSta is identical in all respects to the program that uses PboxLListObj except for the textual substitution of the name of one module for the other. Refer to Figure 21.28 for a dialog box that uses the linked list and Figure 21.29 for a program that implements the dialog box.

The state design pattern for the linked list uses the same kind of abstract node that is used in the state design pattern for the binary tree. The state of a list is defined as a pointer to a head node, which is abstract. An empty node and a nonempty node are type extensions of an abstract node. An empty node contains no attributes. A nonempty node contains a field for the value and a field named next, which is a list. Here is the declaration of a list and its associated nodes.

The data structure for a linked list using the state design pattern

```
TYPE
   T* = ARRAY 16 OF CHAR;
   List* = RECORD
      head: POINTER TO Node
   END;

   Node = ABSTRACT RECORD END;
   EmptyNode = RECORD (Node) END;
   NonEmptyNode = RECORD (Node)
      value: T;
      next: List
   END;
```

Figure 24.9 is the UML diagram for the state design pattern of the linked list, which you should compare with Figure 24.2 for the binary tree. Figure 24.9 shows that seven public methods are exported by PboxLListSta, corresponding to the seven methods provided by the interface in Figure 24.8. As usual, all the methods for the nodes, as well as the nodes themselves, are private. The user of the module has no concept of the internal workings of the list. Unlike the binary tree, however, Figure 24.9 shows that List has two private methods that are helpers for their corresponding public methods. Method DisplayN is a helper for Display and SearchN is a helper for

Search. The abstract node provides six methods that correspond to six of the seven methods provided by the list. Clear is the one method that List can implement without delegating the task to a corresponding Node method. As usual, each concrete node implements all the abstract methods inherited from the abstract node.

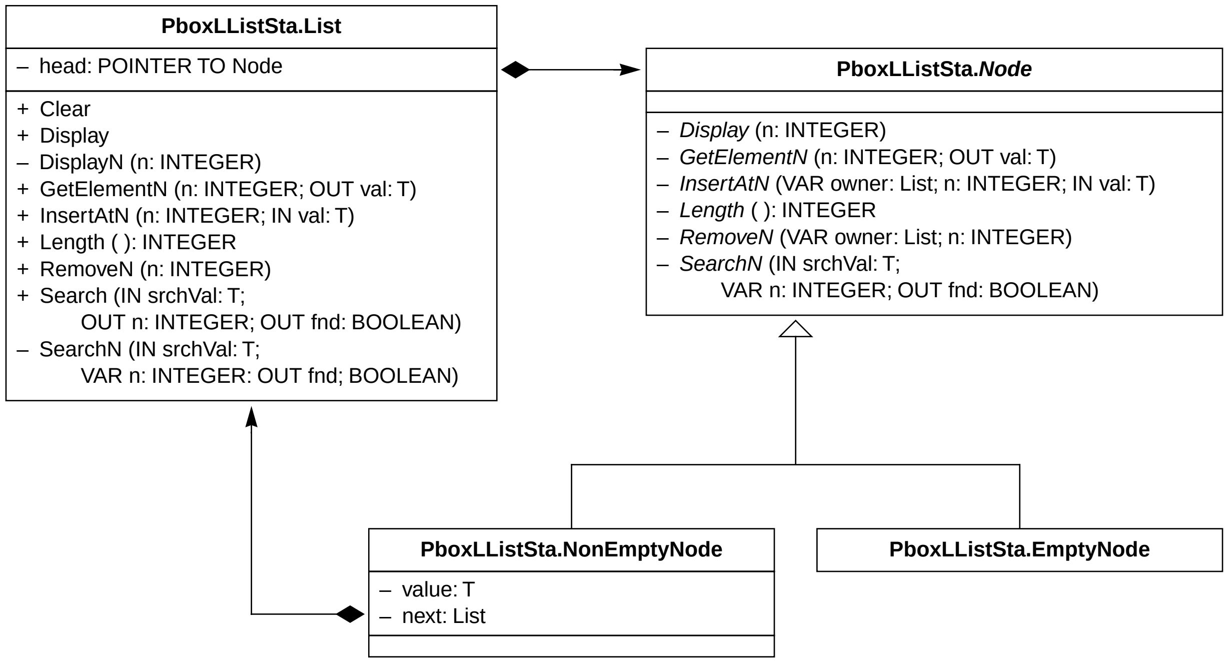

PboxLListSta.List
– head: POINTER TO Node
+ Clear
+ Display
– DisplayN (n: INTEGER)
+ GetElementN (n: INTEGER; OUT val: T)
+ InsertAtN (n: INTEGER; IN val: T)
+ Length ( ): INTEGER
+ RemoveN (n: INTEGER)
+ Search (IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN)
– SearchN (IN srchVal: T; VAR n: INTEGER: OUT fnd; BOOLEAN)
PboxLListSta.*Node*
– *Display* (n: INTEGER)
– *GetElementN* (n: INTEGER; OUT val: T)
– *InsertAtN* (VAR owner: List; n: INTEGER; IN val: T)
– *Length* ( ): INTEGER
– *RemoveN* (VAR owner: List; n: INTEGER)
– *SearchN* (IN srchVal: T; VAR n: INTEGER; OUT fnd: BOOLEAN)
PboxLListSta.NonEmptyNode
– value: T
– next: List
PboxLListSta.EmptyNode

Figure 24.9
The UML diagram for a state design implementation of a linked list.

The viewpoint of each object in the state design pattern for the linked list is identical to the viewpoint of the corresponding object in the state design pattern for the binary tree in Figure 24.3. A List is an owner of an abstract head node. Because the node is abstract, the list cannot see past it and does not know whether it is an empty list or a nonempty list, similar to the tree in Figure 24.3(a). An empty node does not own anything, similar to the empty node in Figure 24.3(b). A nonempty node owns a value and a next list, similar to the way the nonempty node in Figure 24.3(c) owns a value and two children.

As with a tree, the system of cooperating objects works by delegation. A client module typically calls a method for the list. Because the list does not know what kind of list it is, it simply delegates the task to its head node by calling the corresponding method for the node. Polymorphism determines whether the method for an empty node or for a nonempty node executes, with no recourse to an IF statement to determine which. For two methods—Display and Search—the list delegates the task to its helper function. The helper function then delegates the task polymorphically to the corresponding method for the node. As with the binary tree, usually the method for the empty node can execute without further calls, and the method for the nonempty node makes a further call to a method of its next list.

Figure 24.10 is the implementation of the linked list with the state design pattern.

Methods Length, RemoveN, and Search are left as problems for the student.

```
MODULE  PboxLListSta;
    IMPORT StdLog;

    TYPE
        T* = ARRAY 16 OF CHAR;
        List* = RECORD
            head: POINTER TO Node
        END;

        Node = ABSTRACT RECORD END;
        EmptyNode = RECORD (Node) END;
        NonEmptyNode = RECORD (Node)
            value: T;
            next: List
        END;

(* -------------------- *)
    PROCEDURE (VAR lst: List) Clear*, NEW;
        VAR
            p: POINTER TO EmptyNode;
    BEGIN
        NEW(p);
        lst.head := p
    END Clear;

(* -------------------- *)
    PROCEDURE (IN node: Node) DisplayN (n: INTEGER), NEW, ABSTRACT;

    PROCEDURE (IN lst: List) DisplayN (n: INTEGER), NEW;
    BEGIN
        lst.head.DisplayN(n)
    END DisplayN;

    PROCEDURE (IN lst: List) Display*, NEW;
    BEGIN
        lst.DisplayN (0)
    END Display;

    PROCEDURE (IN node: EmptyNode) DisplayN (n: INTEGER);
    BEGIN
        (* Do nothing *)
    END DisplayN;

    PROCEDURE (IN node: NonEmptyNode) DisplayN (n: INTEGER);
    BEGIN
        StdLog.Int(n); StdLog.String(" "); StdLog.String(node.value); StdLog.Ln;
        node.next.DisplayN(n+1)
    END DisplayN;
```

Figure 24.10
The implementation of the linked list with the state design pattern.

Figure 24.10
Continued.

```
(* -------------------- *)
    PROCEDURE (IN node: Node) GetElementN (n: INTEGER; OUT val: T), NEW, ABSTRACT;

    PROCEDURE (IN lst: List) GetElementN* (n: INTEGER; OUT val: T), NEW;
    BEGIN
        ASSERT(0 <= n, 20);
        lst.head.GetElementN(n, val)
    END GetElementN;

    PROCEDURE (IN node: EmptyNode) GetElementN (n: INTEGER; OUT val: T);
    BEGIN
        HALT(21)
    END GetElementN;

    PROCEDURE (IN node: NonEmptyNode) GetElementN (n: INTEGER; OUT val: T);
    BEGIN
        IF n = 0 THEN
            val := node.value
        ELSE
            node.next.GetElementN(n - 1, val)
        END
    END GetElementN;

(* -------------------- *)
    PROCEDURE (VAR node: Node) InsertAtN (VAR owner: List; n: INTEGER; IN val: T), NEW, ABSTRACT;

    PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    BEGIN
        ASSERT(n >= 0, 20);
        lst.head.InsertAtN(lst, n, val)
    END InsertAtN;

    PROCEDURE (VAR node: EmptyNode) InsertAtN (VAR owner: List; n: INTEGER; IN val: T);
        VAR
            p: POINTER TO NonEmptyNode;
    BEGIN
        NEW(p);
        p.value := val;
        p.next.Clear;
        owner.head := p (* Change the state of owner *)
    END InsertAtN;
    PROCEDURE (VAR node: NonEmptyNode) InsertAtN (VAR owner: List; n: INTEGER; IN val: T);
        VAR
            p: POINTER TO NonEmptyNode;
    BEGIN
        IF n > 0 THEN
            node.next.InsertAtN(n - 1, val)
        ELSE
            NEW(p);
            p.value := val;
            p.next := owner; (* Change the state of p.next *)
            owner.head := p (* Change the state of owner *)
        END
    END InsertAtN;

(* -------------------- *)
    PROCEDURE (IN lst: List) Length* (): INTEGER, NEW;
    BEGIN
        (* A problem for the student *)
        RETURN 999
    END Length;

(* -------------------- *)
    PROCEDURE (VAR lst: List) RemoveN* (n: INTEGER), NEW;
    BEGIN
        (* A problem for the student *)
    END RemoveN;

(* -------------------- *)
    PROCEDURE (IN lst: List) Search* (IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN), NEW;
    BEGIN
        (* A problem for the student *)
        fnd := FALSE
    END Search;

END PboxLListSta.
```

Figure 24.10
Continued.

As with the binary tree, the implementation of a linked list with the state design pattern relegates NIL to a lower level of abstraction. Nowhere is there any test for NIL in the implementation of Figure 24.10. Nor does the implementation contain any loops, all of which are replaced by recursion. There are also fewer IF statements, many of which are replaced by polymorphic dispatch.

The implementation of method Clear for the linked list is identical to its counterpart for the binary tree. The method has a local pointer, which it uses to allocate a new empty node. It sets its head pointer to point to the new empty node as does the tree in Figure 24.5.

Display requires a helper function because each item of a list is printed on the Log prefixed by its position in the list. For example, a list of items might be displayed as

```
 0 trout
 1 tuna
 2 cod
 3 salmon
```

So, a nonempty node needs to know its position in the list so it can display the position before it displays its value. The idea is for the helper method DisplayN to contain an additional parameter n that signifies the position of the first item in the current list. When the client module calls Display for a list, the list simply calls the helper function DisplayN with an actual parameter of 0 corresponding to formal parameter n.

Display for a list

```
lst.DisplayN (0)
```

The 0 indicates that the first item in the client's list is at position 0. The helper method delegates the display task polymorphically to the list's head node, passing along the current position.

DisplayN for a list

```
lst.head.DisplayN(n)
```

If the head node is an empty node, there is nothing to print and no more processing to be done. DisplayN for an empty node is simply

DisplayN for an empty node

```
(* Do nothing *)
```

If the head node is a nonempty node, it prints the value of n followed by the value it owns. Then, it delegates the task of printing the rest of the list by calling the helper method for its next list. Because the position of the first item in the next list is the position of the current item plus 1, it supplies n + 1 for the actual parameter.

DisplayN for a nonempty node

```
StdLog.Int(n); StdLog.String(" "); StdLog.String(node.value); StdLog.Ln;
node.next.DisplayN(n+1)
```

You should compare this implementation of Display with the implementation of Display in Figure 21.30 for PboxLListObj. This implementation divides the algorithm into four methods, two of which consist of a single statement and one of which contains no statements! It is necessary to have an implementation for the empty node even if it does nothing, because the method does get called polymorphically. The implementation for PboxLListSta exhibits the object-oriented features of locality of environments in a system of cooperating objects.

When a client calls InsertAtN for a list it supplies n, the position in the list to insert, and val, the value to be inserted. The implementation for the list version implements the precondition with an ASSERT statement, then delegates as usual.

InsertAtN for a list

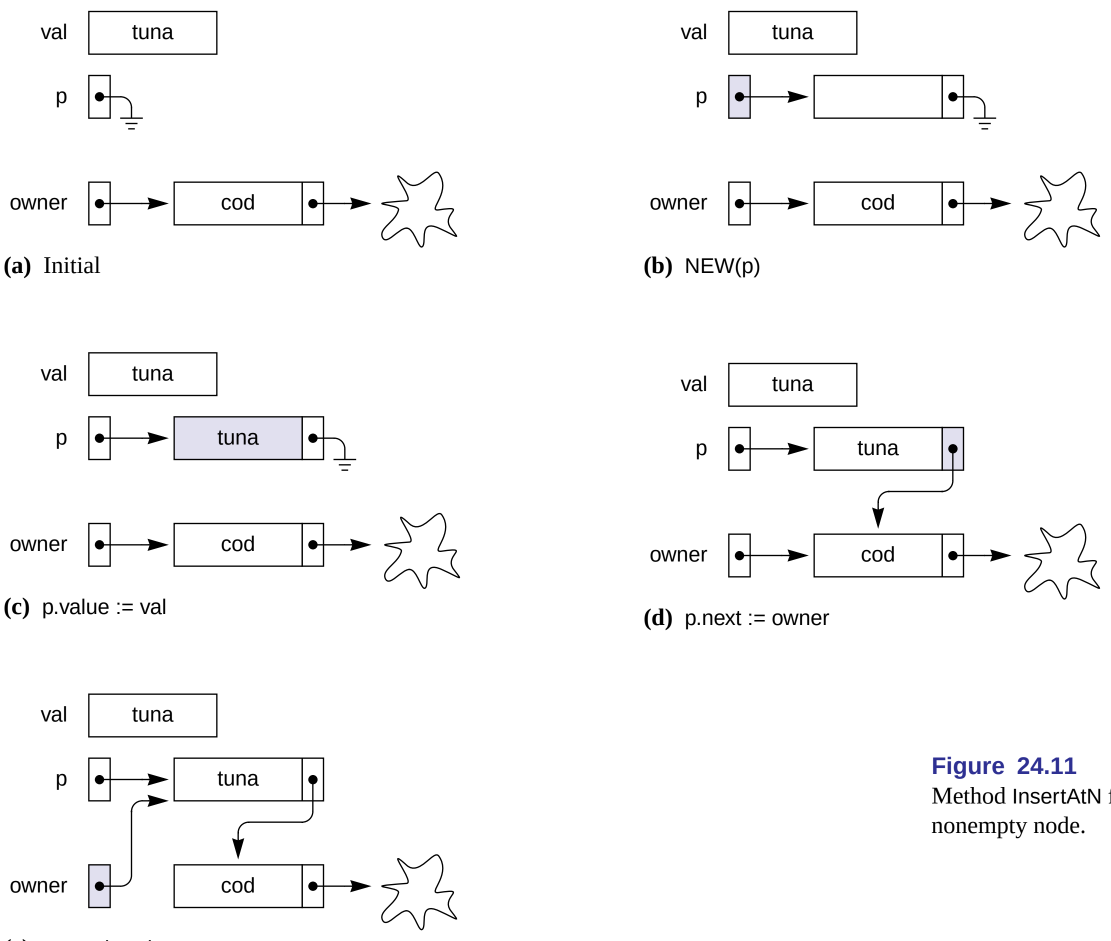
```
ASSERT(n >= 0, 20);
lst.head.InsertAtN(lst, n, val)
```

A third parameter is included in the corresponding method for a node. lst is the actual parameter and owner is the formal parameter. A head node needs to have access to its owner, because the owner's state will change if n is 0.

The implementation of InsertAtN for a nonempty node must first decide if the value is to be inserted at the current position or at a position further down the list. If n is greater than 0, it belongs further down the list. So, the method delegates with

```
node.next.InsertAtN(n - 1, val)
```

Because node.next is a list, the method call is for the list version of InsertAtN, which has only two parameters. The implementation supplies n - 1 for the actual parameter, because the position of the first item in the next list is one less than the current position.



(a) Initial
(b) NEW(p)
(c) p.value := val
(d) p.next := owner
(e) owner.head := p

Figure 24.11
Method InsertAtN for a nonempty node.

If n equals 0, the four statements in Figure 24.11 execute. Figure 24.11(a) shows the initial environment for a nonempty node who owns value cod and next, which is a list. The nonempty node has access to its owner as a formal parameter. The figure assumes that tuna is passed in parameter val from the list. Figure 24.11(b) shows the effect of

```
NEW(p)
```

where p is a local variable that points to a nonempty node. Storage for the node is allocated from the heap. The statement

```
p.value := val
```

sets the value field of the new node to tuna from parameter val. Figure 24.11(d) shows the effect of

```
p.next := owner
```

which changes the state of p.next to point to the same nonempty node to which owner points. Finally, Figure 24.11(e) shows the effect of

```
owner.head := p
```

which changes the state of owner to point to the inserted node.

The implementation of InsertAtN for an empty node is similar to the above implementation for a nonempty node. No IF statement is required, because the specification requires the value to be inserted at the end of the list if the position supplied exceeds the length of the list. The empty node simply executes

```
NEW(p);
p.value := val;
p.next.Clear;
owner.head := p
```

where p is a local nonempty node. The only difference between this sequence of statements and the sequence for a nonempty node is that p.next is cleared instead of being set to point to the following node. There is no following node that must be linked to the inserted node.

You should compare this algorithm to the implementation of InsertAtN in Figure 21.30 for PboxLListObj. This version for PboxLListSta with polymorphism exhibits the object-oriented locality of environment for its distributed system of cooperating objects. Because the environment is local, the code for each method is easier to write and to understand compared to the version for PboxLListObj.

Method Search for a list is implemented with the help of method SearchN also for a list, whose signature differs from that of Search only by n being called by reference (VAR) instead of called by result (OUT).

```
PROCEDURE (IN lst: List) SearchN (IN srchVal: T; VAR n: INTEGER; OUT fnd: BOOLEAN), NEW;
```

The programmer of the client sees the same interface for PboxLListSta as for PboxLListObj and does not need to initialize the value of n before she calls the server method. Inside the PboxLListSta server, Search initializes n to 0 then calls SearchN, which assumes that the initial value of n is defined. The idea is for SearchN to delegate to its head node the request to search for srchVal. The head node executes SearchN for an empty node or for a nonempty node with polymorphic dispatch. The nonempty node reasons that if srchVal is not equal to its value field, it must further delegate the task to the list in its next field. If srchVal is at position n in the next list, then it is at position n + 1 in the nonempty node's owner's list. In that case, the nonempty node must increment n for its owner. The details are a problem for the student.

Problems

1. Complete the methods for the binary search tree of Figure 24.4. You will need to write the methods for the abstract node, and the corresponding implementations for the concrete empty node and nonempty node. Test your implementation with a client program identical to that in Figure 22.11 but importing your server module instead of PboxTreeObj.

(a) PROCEDURE (IN tr: Tree) **Contains*** (IN val: T): BOOLEAN, NEW
(b) PROCEDURE (IN tr: Tree) **NumItems*** (): INTEGER, NEW
(c) PROCEDURE (IN tr: Tree) **InOrder***, NEW
(d) PROCEDURE (IN tr: Tree) **PostOrder***, NEW

2. Complete the methods for the linked list of Figure 24.10. You will need to write the methods for the abstract node, and the corresponding implementations for the concrete empty node and nonempty node. Test your implementation with a client program identical to that in Figure 21.29 but importing your server module instead of PboxLListObj.

(a) PROCEDURE (IN lst: List) **Length*** (): INTEGER, NEW
(b) PROCEDURE (VAR lst: List) **RemoveN*** (n: INTEGER), NEW
(c) PROCEDURE (IN lst: List) **Search*** (IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN), NEW