Chapter 24

The State Design Pattern
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.
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 data structure for a binary tree using the state design pattern
Figure 24.2
The UML diagram for a state design pattern implementation of a binary search tree.
(a) The viewpoint of a Tree.

I am a tree.
I own an abstract node.

My owner.

(b) The viewpoint of an EmptyNode.

I am an empty node.
I own nothing.

My owner.

(c) The viewpoint of a NonEmptyNode.

I am a nonempty node.
I own two trees and a value.

My owner.

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

 value
 leftChild rightChild
MODULE  PboxTreeSta;
IMPORT StdLog;

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;

Figure 24.4
The implementation of the binary search tree with the state design pattern.
PROCEDURE (VAR tr: Tree) Clear
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 24.5
The empty tree in PboxTreeObj and PboxTreeSta.

(a) The empty tree from PboxTreeObj in Figure 22.14.
Figure 24.5
The empty tree in PboxTreeObj and PboxTreeSta.

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

(b) The empty tree from PboxTreeSta in Figure 24.4.
Figure 24.6
The viewpoint of a nonempty node in the environment of an Insert call.
PROCEDURE (IN node: EmptyNode) Insert (VAR owner: Tree; IN val: T);
    VAR
    p: POINTER TO NonEmptyNode;
    val
BEGIN
    NEW(p);
    p.leftChild.Clear;
    p.value := val;
    p.rightChild.Clear;
    owner.root := p (* Change the state of owner *)
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 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 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 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 node: EmptyNode) Insert (VAR owner: Tree; IN val: T);
    VAR
        p: POINTER TO NonEmptyNode;
    BEGIN
        sparrow
        NEW(p);
        p.leftChild.Clear;
        p.value := val;
        p.rightChild.Clear;
        owner.root := p (* Change the state of owner *)
    END Insert;

owner → p
EmptyNode → sparrow
EmptyNode → EmptyNode
The viewpoint of an empty node in the environment of an Insert call.
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.

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

The data structure for a linked list using the state design pattern
Figure 24.9
The UML diagram for a state design implementation of a linked list.
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;

Figure 24.10
The implementation of the linked list with the state design pattern.
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;
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.
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 (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 (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 (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 (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 (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;
Figure 24.11
Method InsertAtN for a nonempty node.