Recitation 9 Solution

Design Patterns

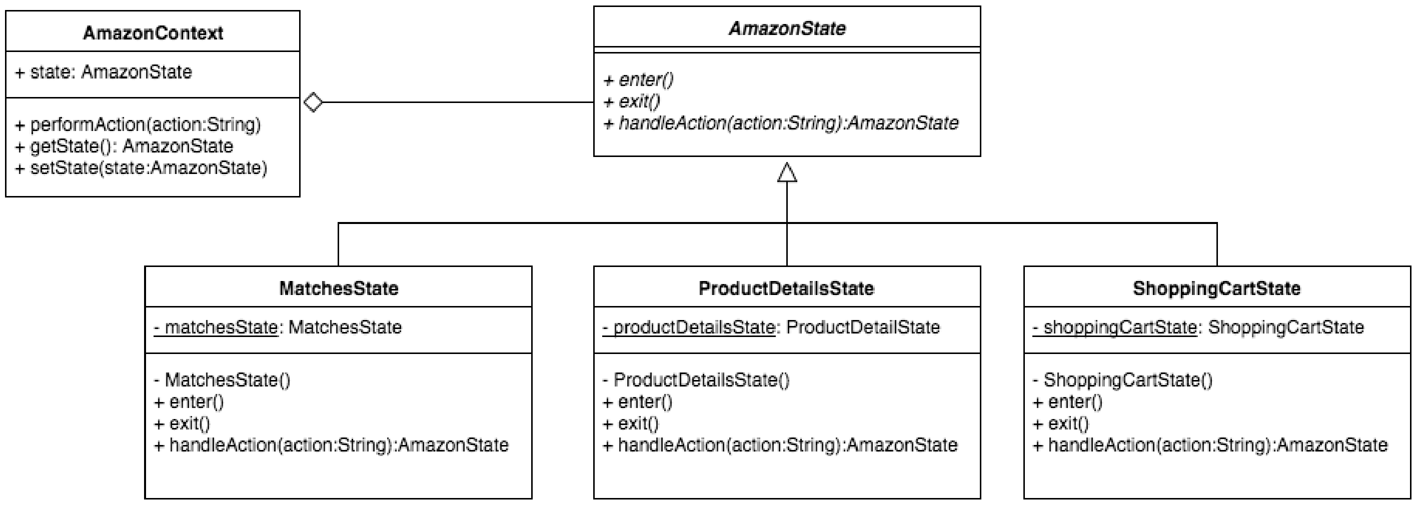
1. Draw a state diagram that models a user's shopping session at amazon.com, starting with a search. Show the UML for an implementation using the state design pattern including key fields and headers for the methods in the states.

SOLUTION

As an example, here are three states: **matches**, **product detail** and **shopping cart**. When a user searches for a product, all matches are shown. When the user clicks on a match, the product detail is shown, and when the user adds a product to the shopping cart, the shopping cart is shown. The following shows a state transition table (a search can be done on any of these pages). For any cell (state i, state j), the action in that cell leads to a transition from state i to state j.

	Matches	Product Detail	Shopping Cart
Matches	Search	Click on a product	Click on shopping cart
Product Detail	Search	-	Click on Add to cart
Shopping Cart	Search	Click on a product	Delete a product

When a state is entered, a web page is displayed that is specific to that state. Each of the above states will be a concrete state class in the application, with methods to enter and process event. Here's a sample UML and some illustrative code.


AmazonContext
+ state: AmazonState
+ performAction(action:String)
+ getState(): AmazonState
+ setState(state:AmazonState)
AmazonState
+ enter()
+ exit()
+ handleAction(action:String):AmazonState
MatchesState
- matchesState: MatchesState
- MatchesState()
+ enter()
+ exit()
+ handleAction(action:String):AmazonState
ProductDetailsState
- productDetailsState: ProductDetailState
- ProductDetailsState()
+ enter()
+ exit()
+ handleAction(action:String):AmazonState
ShoppingCartState
- shoppingCartState: ShoppingCartState
- ShoppingCartState()
+ enter()
+ exit()
+ handleAction(action:String):AmazonState

```
public abstract class AmazonState {
public void enter(AmazonContext context){}
public void exit(AmazonContext context){}
   public AmazonState handleAction(String action) throws Exception;
}

public class ProductDetailState extends AmazonState { ... }
public class ShoppingCartState extends AmazonState { ... }
public class MatchesState extends AmazonState {
   private static MatchesState matchesState = null;
   private MatchesState(){}
   public AmazonState handleAction(String action) throws Exception{
      if (action.equals("search")){
        // rest of the search behaviour
        return matchesState;
      }else if (action.equals("clickProduct")){
        // rest of the clickProduct behaviour
        return productDetailState;
      }
      // other valid actions
      else {
        // unsupported action
        // throw exception
      }
   }
}
```

```
public class AmazonContext {
  private AmazonState state;
  private AmazonContext(AmazonState state) {
    this.state= state;
  }
  public void performAction(String action) throws Exception {
    AmazonState newState = state.handleAction(action);
    if (newState != state) {
      state.exit();
      newState.enter();
      this.state = newState;
    }
  }
  public void setState(AmazonState state) {
    this.state = state;
  }
  public AmazonState getState() {
    return state;
  }
}
```

---

2. Show how you would enhance the Singleton pattern to allow up to a maximum number of instances of an object. There should be a way for clients to recycle instances, i.e. when a client is finished with an instance, it gives it up, and this instance can be later dealt out in response to a new instance request.

   SOLUTION

   Keep an array of instances, and each time an instance is requested, if any member of the array is null, create a new one and return it, otherwise return null (meaning no more instances are available).

   Also keep a boolean array equal in size to the number of instances. When an instance is "checked out", set the associated spot to true, and when an instance is "recycled", set the associated spot to 0. When a client requests a requests a new instance, return one of the available instances, or else return null if none are available.

---

3. Say you design a BinaryTree class. How will you use the Iterator design pattern to implement preorder, inorder, and postorder traversals, each of which just prints the data stored at each node? Sketch your design, and show how a client can call on the different traversals.

   SOLUTION

   Since the inorder, preorder, and postorder traversals use recursion, the state is maintained by the compiler. Since the iterator needs to keep state (where it is currently at) the solution is to use a stack in the tree traversal code itself, instead of doing recursion. Here is the relevant code to implement the iterators, with the complete code for inorder traversal.

```
public class BinaryTree<T> {
 static class BTNode<E> {
   BTNode<E> left, right;
   E data;
   BTNode(BTNode<E> left, BTNode<E> right, E data) {
     this.left = left; this.right = right; this.data = data;
   }
 }
 // root of tree
 BTNode<T> root;

 ...

 Iterator<T> inorderIterator() { return new BTInorderIterator<T>(root); }
 Iterator<T> preorderIterator() { return new BTPreoderIterator<T>(root); }
 Iterator<T> postorderIterator() { return new BTPostorderIterator<T>(root); }
 ...
}

class BTInorderIterator<E> implements Iterator<E> {

 Stack s;

 BTInorderIterator(BinaryTree.BTNode<E> root) {
   s = new Stack();
   addAllLeftChildren(root);
 }
```

```
  void addAllLeftChildren(BinaryTree.BTNode<E> subtree) {
    if (subtree == null) return;
    s.push(subtree);
    addAllLeftChildren(subtree.left);
  }

  public boolean hasNext() {
    return !s.isEmpty();
  }

  public E next() {
    BinaryTree.BTNode<E> t = s.pop();
    addAllLeftChildren(t.right);
    return t.data;
  }

  public void remove()
  throws UnsupportedOperationException {
    throw new UnsupportedOperationException();
  }
}
```