Program 2: Command Design Pattern & Javadoc

Java API
KeyedItem, CD, Song Documentation
FileIO Documentation

The Iterator design pattern provides a way for the user to visit all of the items in a data structure, performing an operation on each item. The user performs the loop and operates on the data, so this technique is called **external iteration**.

As useful as this is, it is possible for the user to add or remove elements while the iteration is being performed. Taking a snapshot of the data when the iterator is requested or throwing an exception when an attempt is made to change the data in the collection are possible ways of handling this potential problem. However, the Expert pattern suggests that the data structure holding the items should be performing the iteration over the data that it contains. This technique is called **internal iteration**, and is commonly implemented using the **Command Design Pattern**.

Objective:

- Interfaces
  - Command Interface
  - Execute Interface

Part I: For-Each and Iterable

The for-each statement requires that the collection being iterated over implement the built-in **Iterable** interface. The only method in this interface is **public Iterator iterator()**. In order to use the for-each statement with your AVLTree, your SearchTreeInterface must extend the Iterable interface. However, you will need to comment out the iterator() method currently in SearchTreeInterface.

Modify the old iterator method in BinaryTreeBasis to conform to the new method signature. Use setInorder as your method to traverse the tree. Now, when working with the Command design pattern, you can use the convenience of the for-each statement.

Part II: Command Interface, Execute Interface

The Command interface contains one method that accepts a single Object.

- public void execute(Object item)

The Execute interface also contains one method, but it accepts Objects that implement the Command interface.

- public void execute(Command command)

Part III: Command Design Pattern

Objects implementing the Execute interface are objects that store a collection of other objects. When the method in the Execute interface is invoked, the collection object will iterate over the items in the collection, calling the method in the Command interface on each object in the collection. Thus, it is not possible for the user to manipulate the collection while the internal iteration is performed, and the collection maintains control over the data that it is responsible for.

Modify AVLTree to implement the Execute interface, performing an inorder traversal over the items in the tree. Call the Command interface method on each item visited.

Part IV: WorstCDs

- Song.java
- KeyedItem.java
- CD.java
- FileIO.java
- FileIOException.java

Implement the Command interface using a class (**WorstCDs**). This class should figure out the lowest rating of the CDs in the AVLTree, collecting all of the CDs with this rating in an ArrayList. Allow a driver class to have access to the resulting ArrayList of worst CDs (**ArrayList<CD> getWorstCDs()**). Display the results, which should be in sorted order by CD title. (This will be similar to BestCDs.java)

Part V: Longest Song

Implement the Command interface with another class (**LongestSong**). Process the CDs to find the overall single longest song. Allow a driver class to have access to the longest song (**Song getLongestSong()**). Display the longest song in your driver.

Part IV: Javadoc

Fully comment **all** (including protected methods) of your newly created methods using preconditions (description of parameters), postconditions (description of return values), and throws (does the method throw an exception?). Note that interface comments are automatically inserted into the class that implements the interface.

Example:

```
/**
 * Searches for the leaf insertion location for item. <br>
 * Precondition: item is not null
 * Postcondition: returns the current node so that it can be linked (or relinked) to its parent
 * depending on whether a left or a right was taken in obtaining the leaf insertion location.
 * Throws: TreeException if an attempt to insert a duplicate is detected.
 * Calls: insertLeft, insertRight, insertDuplicate depending on the item's sk
 */
protected TreeNode insertItem(TreeNode tNode, KeyedItem item) throws TreeException
{
```