Homework #7
due Monday, October 31, 10:00 PM

In this assignment, you will implements stacks and queues of "endogenous" coin objects. You will also get some experience in using inheritance to save code duplication. It is due *after* the midterm, so that it doesn't interfere with your studying, but you are recommended to do as much as practical before the midterm, because you may be tested on the concepts in this assignment.

Use this link to accept the assignment:

`https://classroom.github.com/a/GHYgpFUI`

1 Concerning the new `Coin` class

In lecture, we have used a normal `Coin` class as an element type of Bags. The ADTs didn't enforce that a coin was only in one bag at a time despite the fact that having the same coin in two different bags doesn't make sense. (Although it would be very useful to be able to stick a coin in the parking meter and still hold onto it for other purchases.) For this homework, we redefine the `Coin` class so that each coin can only be in one contaner at a time. This uses the concept of "endogenous lists" which was explained in the "linked list variations" handout and the pre-recorded lectures in Module 5. You are recommended to take another look at these resources.

An endogenous list uses links within the values themselves, rather than external "node" objects. Endogenous lists are usually hand-written (not library classes) and specialized for low-level situations where we don't want to incur memoy allocation or for security situations where we want to contol how objects are used. In this homework, our endogenous lists are singly-linked lists (SLLs) of coins and so every Coin has a "next" pointer in it that is used to link up the lists.

The links in the data values (coins, for us) must be accessible to the list manipulation code. A number of different techniques can be used for this purpose. For this assignment, we use a special package `edu.uwm.cs351.money` which has all the classes that need to access the internals. Client classes are declared in different packages. The fields are not private, but "default" (package-level) access. The whole package co-operates to encapsulate the data structure properties.

Each coin has a "type" (such as `QUARTER`) recognizing that each coin is its own object, but it is very possible to have multiple quarters. The type has associated its value (e.g., 25¢), and a size (relevant when we want to stack coins).

Each coin remembers which "mint" created it, and also a serial number (which isn't used in our homework assignment). It also keeps track of which container (endogenous list of coins) that it is within. The owner can be null while the coin is outside any container.

The creation of coins is strictly controlled. Other than the "Spy" (which creates apart from a mint, as explained later), they are only created by a "mint." Furthermore, mints are controlled too; a client cannot get access to a mint (we don't want a client to be able to create their own cash). There is a `Bank` class, which for simplicitly has a single account.

The client cannot create a Bank, but it can go to the (single) bank and ask to withdraw cash from the (single) account. The account is initialized with $2.00 which won't get the client very far, but is good enough for our assignment. In a larger program, we'd have multiple accounts and more ways for money to enter circulation in a controlled way. But at some level, this simple system models how the Federal Reserve can create money by just making its account balance bigger.

2 The Container classes

In this assignment, you will implement three classes that satisfy the Container interface (ADT). You have already seen interfaces such as Iterator which specify the methods and what they do, but not how they do it. The `Container` interface specifies the methods that all containers provide. Please read the `Container.java` file which documents all five methods.

For this assignment, you implement three container classes:

DefaultContainer This class implements a container something like the Bag class discussed in lecture. It uses a singly-linked list (all the containers are singly-linked) with a "head" pointer. The data structure is not private, but rather "default" access. Other classes in the package are trusted to work together to maintain the integrity of the money system.

It is effectively LIFO in than coins are added and removed at the head. This class should not declare a "node" class, because for this assignment, all the containers are endogenous.

Stack This class implements a stack of coins, where we require that we can never place a larger coin on a smaller coin (e.g., a quarter cannot be placed on top of a dime). The "head" is the top of the stack (the place where coins are added or removed.) Otherwise, a stack behaves just like a default container (and indeed should "inherit" the code, as explained below).

Pipeline This class implement a FIFO queue of coins. As well as a head, it also has a "tail" pointer. Coins are added at the tail, and (as with all the containers) removed from the head.

2.1 Concerning Inheritance

You will use *inheritance* to implement `Stack` and `Pipeline`; each should be declared "`extends DefaultContainer`" and then re-implement/override methods only as needed.

So far, you have been told to indicate whether an override is done because it is "required" by Java, a brand-new "implementation" or needed just for "efficiency." To these three, we add a new reason:

decorate This means that we call the overridden method using "`super.`*methodName*`(...)`" at some point in the code, perhaps doing extra work, before, after or instead of the

overridden behavior. We distinguish this reason from "implementation" which should not make a "super" call.

We have chosen a design for `DefaultContainer` that makes use of two helper methods, declared "protected" (needed for inheriting classes only). You are required to split out some of the work of "`add`" to use "`takeOwnership`" and to split out some of the work of "`remove`" to use "`relinquish`." This will enable the extending classes to get the required behavior while overriding/re-implementing as little as possible.

None of the classes should add fields (other than "tail" for `Pipeline`). In particular, there should be no "manyItems" field or the like.

2.2 The Invariants

Each container class has an invariant but some of the items are shared:

1. (shared) The linked list starting at "head" must not contain a cycle. You should use Floyd's Tortoise & Hare algorithm to check this. The code was given in Homework #4.
2. (shared) All the coins in the linked list must be "owned" by the container being checked.
3. (specific to Stack) No larger coin may be placed on top of a smaller coin.
4. (specific to `Pipeline`) The "tail" must be the last Coin (if any) in the list starting at the "head."

You should use inheritance to avoid duplicating code.

3 Moving Stacks

If you have a tall stack of coins and wish to move it to another stack, it can be tricky. You cannot just move one coin at a time to the new stack, because otherwise the new stack would have the coins in reverse order which may cause problems with a bigger coin placed on top of a smaller one. If you have an unbounded amount of space available, you can place each coin in its own singular stack, but supposing one has only one additional stack (beyond the source and the destination of the move), it's possible to do the move, with a lot of intermediate steps.

The basic idea is that one can move n coins from one stack to another by first moving $n-1$ coins to a temporary (helper) stack, Then one moves the bottom coin to the destination stack. Then one moves the $n-1$ coins from the helper stack to destination stack. Of course, if at some point we need to move zero coins, we can declare the operation done without any work.

When one is moving coins from the source stack to the helper stack, we use the destination stack as a helper stack. Similarly, when moving coins from the helper stack to the destination stack, the source stack can be used a temporary stack for coins. The whole process can be expressed as a very simple recursive algorithm. You should implement it using a *single* recursive helper method in `MoveStack.java`. This will also give you practice writing client code, which uses the ADT methods, without having access to the internals.

You may search "Towers of Hanoi" on the web, which is the classic name for the "move stack" algorithm. If you get any code or logic, make sure to credit the web page in a comment in your code, as always. Of course you have to change things to fit our ADTs, so it may be easier and cleaner to not read anyone else's code, and just write it all yourself.

There is also a main method where you set up a situation where you test the algorithm on a stack of many coins withdrawn from the bank account.

4 Concerning Spies

In previous assignments, we have used an invariant tester class nested into the ADT's class implementation so that it could access the internals, and test your invariant checker. In this assignment, we use a different approach. Since the whole package reveals its internals within the package, we use a "spy" to provide information needed for testing. The spy is used only for testing (development), and wouldn't be provided in the library code made available to the client.

The spy class uses inheritance and the fact that most of the fields are default access to give tests limited access to the internals. For this reason, we can do the internal tests as part of the regular tests. We can also check that functions are called the correct number of times. Some languages (such as JavaScript) make it much easier to define spies (and "mocks"), but many spy-like functions can even be done in Java. Indeed Java provides "reflection" which gives many spy-like abilities, but it works outside the language (unlike the `Spy` class for this assignment). Perhaps a future assignment might use reflection to implement internal tests.

5 Files

The repository includes the following files:

src/TestContainer.java An abstract test for testing containers.

src/Test{DefaultContainer,Stack,Pipeline}.java Tests for the the three container ADTs.

src/TestEfficiency.java Efficiency tests; each test should take no more than a second at most.

TestMoveStack.java Tests for the "move stack" algorithm.

src/UnlockTest.java Code to unlock all tests.

src/edu/uwm/cs351/MoveStack.java Skeleton file for the "move stack" algorithm.

src/edu/uwm/cs351/money/{Type,Coin,Mint,Bank}.java The basic coin implementation files.

src/edu/uwm/cs351/money/Container.java The container ADT interface.

src/edu/uwm/cs351/money/{DefaultContainer,Stack,Pipeline}.java Skeleton files for the three container classes.

src/edu/uwm/cs351/money/Spy.java The "spy" class described above.

There are regular tests, locked tests and efficiency tests but no random tests for this assignment.

6 What You Need to Do

You need to unlock the tests, and before so, to read the documentation on all the methods in `Container.java` in the repository.

You need to complete the following ADTs:

1. DefaultContainer
2. Stack
3. Pipeline

Make sure to follow our usual conventions:

- Every public method/constructor not declared `@Override` must have a complete documentation comment.
- Evry method that is declared `@Override` must have a single word comment: one of "required," "implementation," "efficiency" or (new to this assignment) "decorate."
- Every public method must assert the invariant before it does anything else, and again at the end, just before returning, unless it doesn't change anything.

You need to complete the following algorithms:

1. MoveStack

Finally you need to write a client program (`main` method) that creates a stack of coins withdrawn from the bank (do not use the spy) and moves it to another stack. (There's no capability to deposit the coins back in the bank, so you don't need to do that.)