Peter Freeman Mutch
https://github.com/CSC-413-SFSU-02/csc413-p3-interpreter-peterpanning

CSC 413 HW 3: X Language Interpreter

For this assignment, I implemented an interpreter for the mock language X. Some code--the Interpreter and CodeTable class--were already implemented for me. The Interpreter class is the entry point to this project. Requirements included:

1. Implementing the ByteCodes classes listed in the table on page 5/129 of the assignment, making sure to abstract similarities between the classes and create the correct parent class.
2. Completing the implementation of a. ByteCodeLoader b. Program c. RuntimeStack d. Virtual Machine
3. Making sure that all variables have the correct modifiers.
4. Making sure not to break encapsulation.

Although my code does not break encapsulation, compiles, and executes without throwing exceptions, it does not produce the correct output. This turned out to be due to two off-by-one errors in runTimeStack.newFrameAt() and runTimeStack.load(), as discovered by Prof. Souza shortly before the submission deadline. I'm disappointed that I was unable to find this error myself, and plan to revise my submission after the deadline for -25% penalties has passed.

Command line instructions to compile and execute

To compile and execute this program, I created a new IntelliJ project with the git repository as the project root directory. I edited the build configuration to point at the local fib.x.cod or factorial.x.cod file as desired, built the project, and ran it. When testing the ouput, I set interpreter.VirtualMachine.dumpOn to True. It is set to False by default, and does not dump the runtime stack until that is changed, either manually or by a DumpCode.

Assumptions

I was given a codebase and told that I could assume it was correct. We were also told not to expect zero division, so that will break the interpreter. It also only works with integer operations, no floating points. Because of the input validation I wrote, it should be able to interpret Xcode files when PopCodes are called with arguments larger than the size of the frame or runtime stack.

Implementation

Starting at the project root and proceeding in the order which files and folders appear in the GitHub repository (that is, alphabetically):

ByteCode: This folder contains the ByteCode Abstract class and all of its inherited classes, from ArgsCode to WriteCode. The common features between every ByteCode were that they would all require a method of initialization, a method of execution, and a toString() method. The abstract ByteCode class therefore declared all of those and made them abstract, expecting every single subclass to implement each of those functions.

The toString() functions for each subclass were made by IntelliJ. I don't think I changed any of them at all except for the ReadCode class, which requires a Scanner. Instead of printing out the Scanner, which is fairly uninformative and quite lengthy, I just had that class print out the input it had read in.

The init() function of every ByteCode subclass takes in an ArrayList of Strings, because they are parsed from a file, which is text. This often meant converting those Strings to Ints, which was pretty painless.

The execute() function of every ByteCode typically just does what it says it does. The functions are short and sweet, and the main goal was to write them in such a way that did not break encapsulation, simply calling functions of the Virtual Machine rather than interacting directly with the RunTimeStack.

There are, of course, some special cases. CallCode, FalseBranchCode, and GoToCode all required special treatment. They would initially hold labels read in from program files, which needed to be converted to line numbers. This was the responsibility of the Program class, using the destInt field to represent the Destination of that ByteCode as an Integer.

For one heart-stopping moment just now, I thought that I had forgotten to preserve encapsulation at this step. Just checked, we're all good.

Another heart-stopping moment later, my Sublime crashed as I was wrote that sentence, and I thought that I had lost all the changes I made to this documentation. It reopened just fine, and I still had all my changes. We're all good.

What a roller-coaster of emotions.

There is one other special ByteCode, which is DumpCode. It simply flips a switch in the VM, which, if the switch is flipped, causes the VM to print last executed ByteCode and the contents of the runtime stack after the execution of every ByteCode. Extremely useful for

debugging and grading, not so useful for an actual interpreter. The flag in the VM is set to false by default for this reason.

That's about all there is to say about ByteCodes in this project, from what I can tell.

The **ByteCodeLoader** class reads in the file which the interpreter will operate on, and tells the Program class to resolve all the address labels within to line numbers.

The **CodeTable** class holds a HashMap of all the ByteCodes as they appear in Xcode files and their corresponding class names, as well as a method to access that table.

The **Interpreter** was already given to us, but for good measure: it tells the ByteCodeLoader to load in ByteCodes from the appropriate file and then executes that Program in a Virtual Machine.

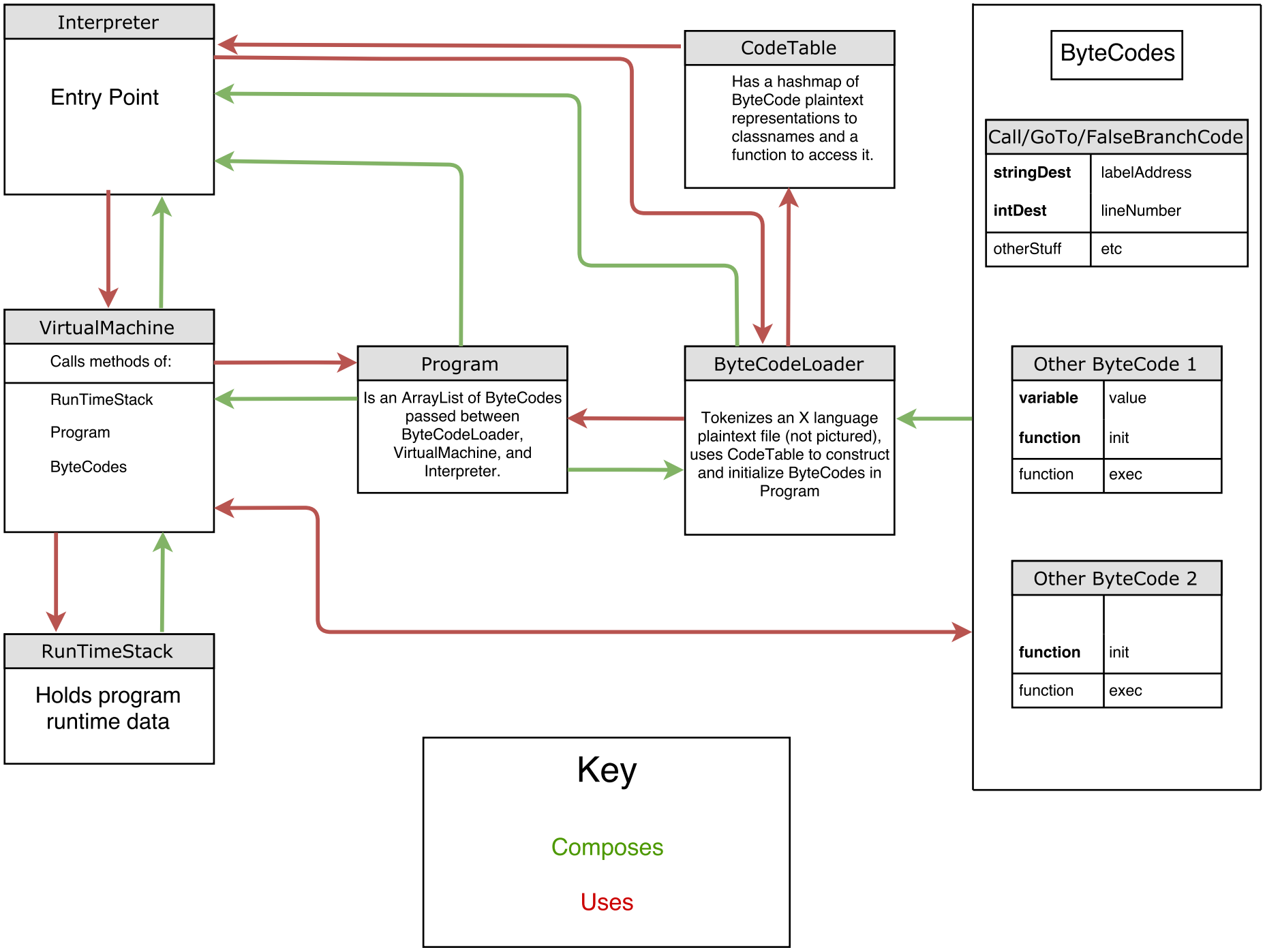
The **Program** class holds the ByteCodes, and resolves their address labels to line numbers the first time it loads a new program.

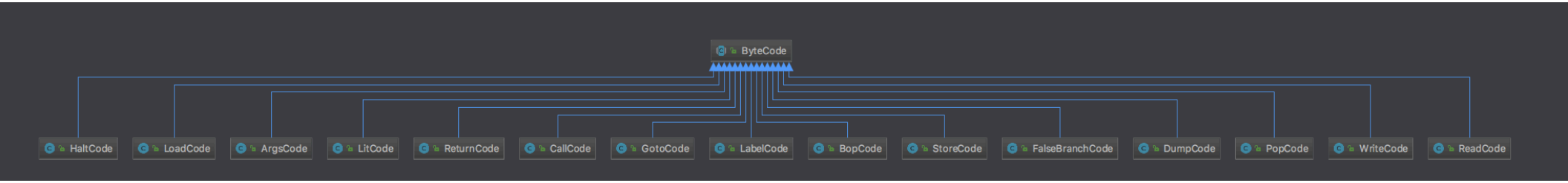
The **RunTimeStack** class is where most of the magic happens. It has pretty good commenting throughout so I'm not going to review every function in depth. It functions as a runtime stack for our load/store based mock language X. It has a couple of methods that I'm pretty sure are redundant, but they were in the assignment spec. It also does not work properly, having off-by-one errors in newFrameAt() and load().

VirtualMachine is where the rest of the magic happens. The Virtual Machine provides the interface through which ByteCodes interact with the runtime stack. It does this through the magic of ***encapsulation***. ByteCodes take in the VirtualMachine as an argument so that they can call its various methods, which in turn call methods of the runtime stack. It's all pretty majestic, really. The VirtualMachine has the option of printing the last executed ByteCode and the entire runtime stack if the dumpOn variable is set to true, either manually or by a DumpCode.

Results and Conclusions

Again, although my code does not break encapsulation, compiles, and executes without throwing exceptions, it does not produce the correct output due to two off-by-one errors in runTimeStack.newFrameAt() and runTimeStack.load(). The conclusion which I draw from this result is that I need to learn how to write JUnit tests to test individual functions as I write them, even if they seem simple at first glance. It also might have been helpful for me to write some small XCode files of my own, just simple "Hello World" and "1 + 1 = 2" files, atomic tests of each operation as I added them, but that seems excessive. This project was a decent exercise in encapsulation. I'm very glad to be done with it.


Interpreter
Entry Point
CodeTable
Has a hashmap of ByteCode plaintext representations to classnames and a function to access it.
ByteCodes
Call/GoTo/FalseBranchCode
stringDest
labelAddress
intDest
lineNumber
otherStuff
etc
VirtualMachine
Calls methods of:
RunTimeStack
Program
ByteCodes
Program
Is an ArrayList of ByteCodes passed between ByteCodeLoader, VirtualMachine, and Interpreter.
ByteCodeLoader
Tokenizes an X language plaintext file (not pictured), uses CodeTable to construct and initialize ByteCodes in Program
Other ByteCode 1
variable
value
function
init
function
exec
Other ByteCode 2
function
init
function
exec
RunTimeStack
Holds program runtime data
Key
Composes
Uses


ByteCode
HaltCode
LoadCode
ArgsCode
LitCode
ReturnCode
CallCode
GotoCode
LabelCode
BopCode
StoreCode
FalseBranchCode
DumpCode
PopCode
WriteCode
ReadCode