Laith Hussin
921659619
Apr 17-2022

Assignment 4 Documentation

Github Repository

https://github.com/sfsu-csc-413-fall-2022-roberts/assignment-4---interpreter-laith57th.git

Project Introduction and Overview

Using a code skeleton provided for the interpreter class and other corresponding classes, I was required to create 15 new byteCode classes that extend an abstract super class, implement the correct behavior in each subclass for correct functionality, and expand on the ByteCodeLoader, CodeTable, Program, RunTimeStack, and Virtual Machine classes to utilize the behavior of each bytecode dynamically to correctly read and interpret a bytecode x file.

Scope of Work

Task		**Completed**
Create new subclasses that extend the abstract ByteCode class		**X**
	Create 15 subclasses for each bytecode that extend the ByteCode class inside the subpackage bytecode, inside the package interpreter.	**X**
	Create a BranchCode super abstract class that extends the ByteCode class inside the subpackage bytecode, inside the package interpreter.	**X**
	Provide the necessary behavior to each bytecode subclass.	**X**
Implement CodeTable class		**X**
	Create a HashMap that takes the name of the bytecode as values keyed by the bytecode as scanned from the bytecode file.	**X**
	Create getClass method that takes the bytecode as the string read from the file and returns the value inside the hashMap.	
Implement the ByteCodeLoader class		**X**
	Write a loadCodes method that reads the token stream from the bytecode file, creates instances of each bytecode, then resolves the addresses by calling the helper function from the program class.	**X**
Implement Program class		**X**
	Create a HashMap that stores the symbolic addresses of each read bytecode.	**X**
	Create a ByteCode Vector that keeps track of the program counter.	**X**
	Implement the addCode method that adds a bytecode to the program object.	**X**
	Implement the resolveAddresses method by iterating through the program vector and setting the new target pc for the jump bytecodes such as CALL, FALSEBRANCH, and GOTO.	**X**

Implement RunTimeStack class		**X**
	Write push, pop, peek, length, newFrameAt, popFrame, store, setFrameIndex, and load methods.	**X**
	Implement the dump method using the frameIndex instance variable to help generate the correct output.	**X**
Test the program by generating bytecode files from the compiler class in our last assignment.		**X**
	Tested the functionality of the interpreter by setting DUMP on throughout the bytecode file tokenization.	**X**
	Test using the following scopes.x.cod file: DUMP ON GOTO start<<1>> LABEL Read READ RETURN LABEL Write LOAD 0 dummyFormal WRITE RETURN LABEL start<<1>> LIT 0 i LIT 0 j GOTO continue<<3>> LABEL f<<2>> LIT 0 j LIT 0 k LOAD 0 i LOAD 1 j BOP + LOAD 2 k BOP + LIT 2 BOP + RETURN f<<2>> POP 2 LIT 0 GRATIS-RETURN-VALUE RETURN f<<2>> LABEL continue<<3>> LIT 0 m LIT 3 ARGS 1 CALL f<<2>> STORE 2 m LOAD 1 j LOAD 2 m BOP + ARGS 1 CALL Write	**X**

	STORE 0 i POP 3 HALT **The Result matched the desired output for the interpreter.**	

Execution and Development Environment

To complete this project, I used Visual Studio Code on my Macbook M1 pro.

openjdk version "17.0.2" 2022-01-18

OpenJDK Runtime Environment Temurin-17.0.2+8 (build 17.0.2+8)

OpenJDK 64-Bit Server VM Temurin-17.0.2+8 (build 17.0.2+8, mixed mode)

Compilation Result

Using the terminal, I used the following commands on the project files:

```
> javac interpreter/bytecode/*.java
> javac interpreter/Interpreter.java
> java interpreter.Interpreter <bytecode file>
```

The program ran as expected and no error messages were displayed

Assumptions

Assumed a correct bytecode file will be passed to the interpreter class during execution.

Summary of technical work

To complete this project I implemented a strategy design pattern to dynamically choose the behavior executed for each scanned bytecode from the token stream. Using object oriented programming, I passed the information of the bytecode classes to the interpreter while keeping all instance variables private to each respective class. I also used single function methods in each bytecode class and the corresponding classes, especially the RunTimeStack class.

Implementation

ByteCode subclasses

1. ByteCode
    a. This is the abstract superclass that is parent to all the bytecode subclasses. This class initializes the init method that takes an arraylist of string parameters and an execute method that takes a VirtualMachine object.
2. ArgsCode
    a. This bytecode class prepares for a function call by creating a new frame and passing the instance variable derived from the tokenstream. The init method simply sets the instance variable string equal to the bytecode name.

3. BopCode
    a. This class is the longest class out of all the bytecode files; however, this is only because there are multiple operators that require different behavior. The way I went about doing this was a switch statement that takes the variable operator as a parameter and performs the correct operation then simply pushes the result on top of the runTime stack.
4. BranchCode
    a. This class is simply a helper byteCode instance for the resolveAddresses method inside the program class. This class extends ByteCode and is a parent class to CALL, FALSEBRANCH, and GOTO. Essentially all the classes that require a new target program counter. This class initializes new abstract base classes, getTarget, setTarget, and getLabel that need to be implemented in the child classes.
5. CallCode
    a. This bytecode extends BranchCode. The init method simply sets the functionName instance variable to the string read from the bytecode file. The execute method sets a new target pc and transfers control to the function call. This class implements all the BranchCode abstract methods.
6. DumpCode
    a. This class simply sets the dump on or off depending on the passed value from the bytecode file.
7. FalseBranchCode
    a. This bytecode extends BranchCode. The init method simply sets the label instance variable to the string read from the bytecode file. The execute method sets a new target pc if the value on top of the runTimeStack is true. This class implements all the BranchCode abstract methods.
8. GotoCode
    a. This bytecode extends BranchCode. The init method simply sets the label instance variable to the string read from the bytecode file. The execute method sets a new target pc and transfers control to the new label. This class implements all the BranchCode abstract methods.
9. HaltCode
    a. This class sets isRunning to false in the VirtualMachine and prints “HALT” if dump is on.
10. LabelCode
    a. This class’s init method sets the label instance variable to the value read from the bytecode file and has a getLabel method that returns the label.
11. LitCode
    a. This class’s init method checks if there’s an id connected to the literal value and sets the id instance variable to that value; otherwise, sets id to null. The execute method for this class simply prints the dump output if dump is on.
12. LoadCode
    a. This class’s init method sets the id and value instance variables to the values read from the bytecode file. The execute method calls the load helper method from the runTimeStack class and prints the dump output if dump is on.
13. PopCode
    a. This class’s init method sets the level instance variable to the value read from the bytecode file. The execute method pops all the values up to the specified level in the runTime stack and prints the output if dump is on.
14. ReadCode
    a. This class reads a new value from the user and pushes it on top of the runTime stack.
15. ReturnCode
    a. This class’s init method checks if there’s functionName present after the return

keyword and sets functionName to null otherwise. The execute method for this class simply prints the dump output if dump is on.

16. StoreCode
    a. This class simply stores the value read into the offset from the start of the frame by calling the store method in the RunTimeStack class. The execute method also prints the output if dump is on.
17. WriteCode
    a. This bytecode simply writes the value on top of the stack to the output if dump is on.

CodeTable class

This class was simple to implement. I started by creating a HashMap to store the bytecodes, then I created a getClass method to retrieve the bytecode instance from the hashmap using the passed string parameter.

ByteCodeLoader class

This class was similar to the lexer class I implemented in an earlier assignment. To implement the loadCodes method, I created a new program object and iterated through the bytecode file adding each bytecode to the program object. One thing I struggled with was using the forName method to create a new instance of each bytecode. My IDE kept suggesting that the newInstance method had been deprecated. After some research, I found that I need to use the getDeclaredConstructor() to retrieve the respective constructor of each bytecode.

Program class

I started implementing this class by creating a Vector of bytecode objects and HashMap to hold the symbolic addresses used in the resolveAddresses method. The program class contains three methods: addCode, getCode, and resolveAddresses. The addCode method takes a byteCode object as a parameter, checks if the bytecode is a label and adds it to the addresses HashMap, then adds the bytecode to the program vector. The getCode method simply retrieves the bytecode from the program vector. Finally resolveAddresses creates a temporary integer jumpAddress, iterates through the program vector, checks if the bytecode is either a CALL, FALSEBRANCH, or GOTO, and sets the new target address by using a BranchCode object. Overall this class wasn't too difficult to implement, the trickiest part was probably the resolveAddresses function.

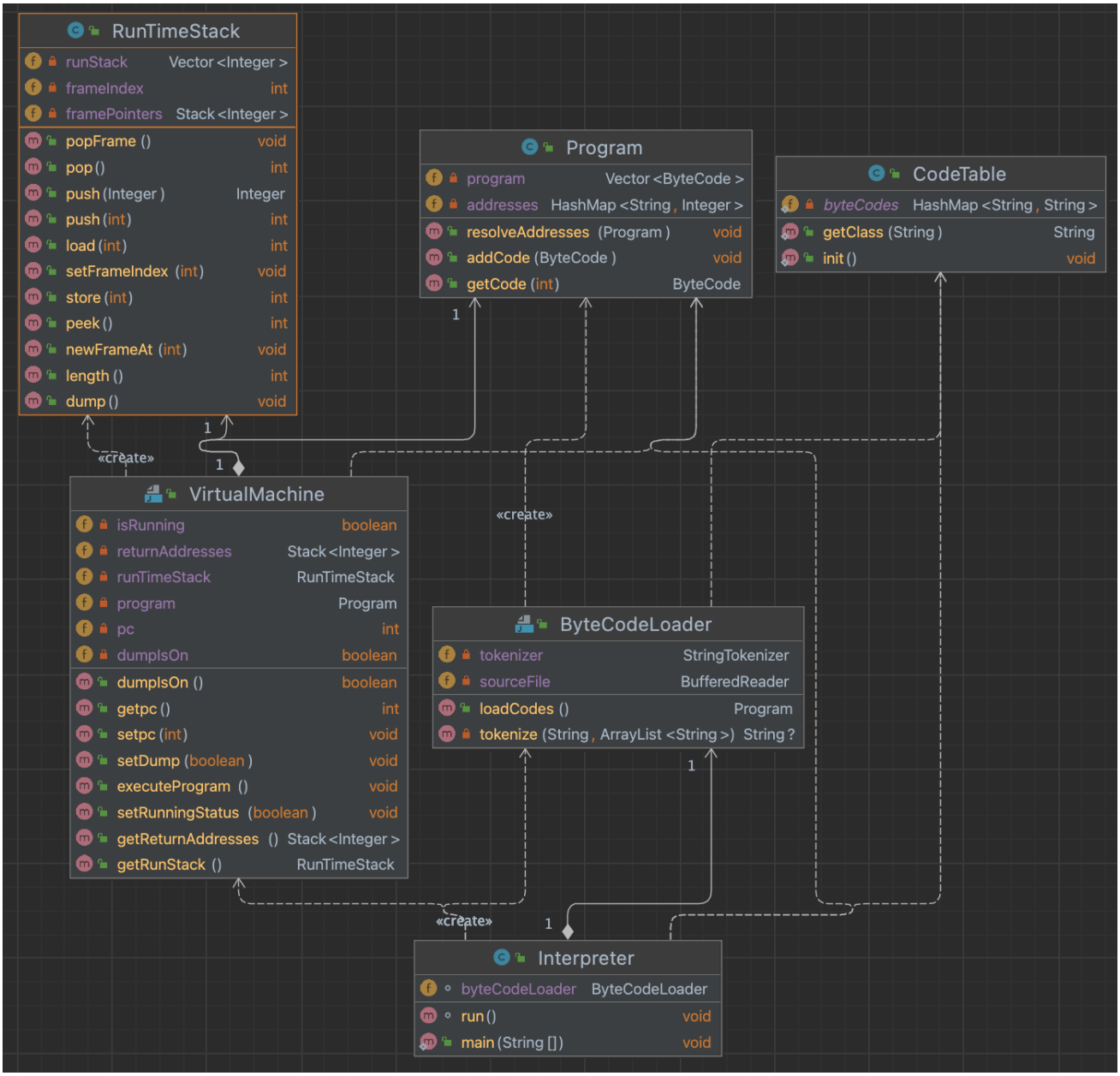
RunTimeStack class

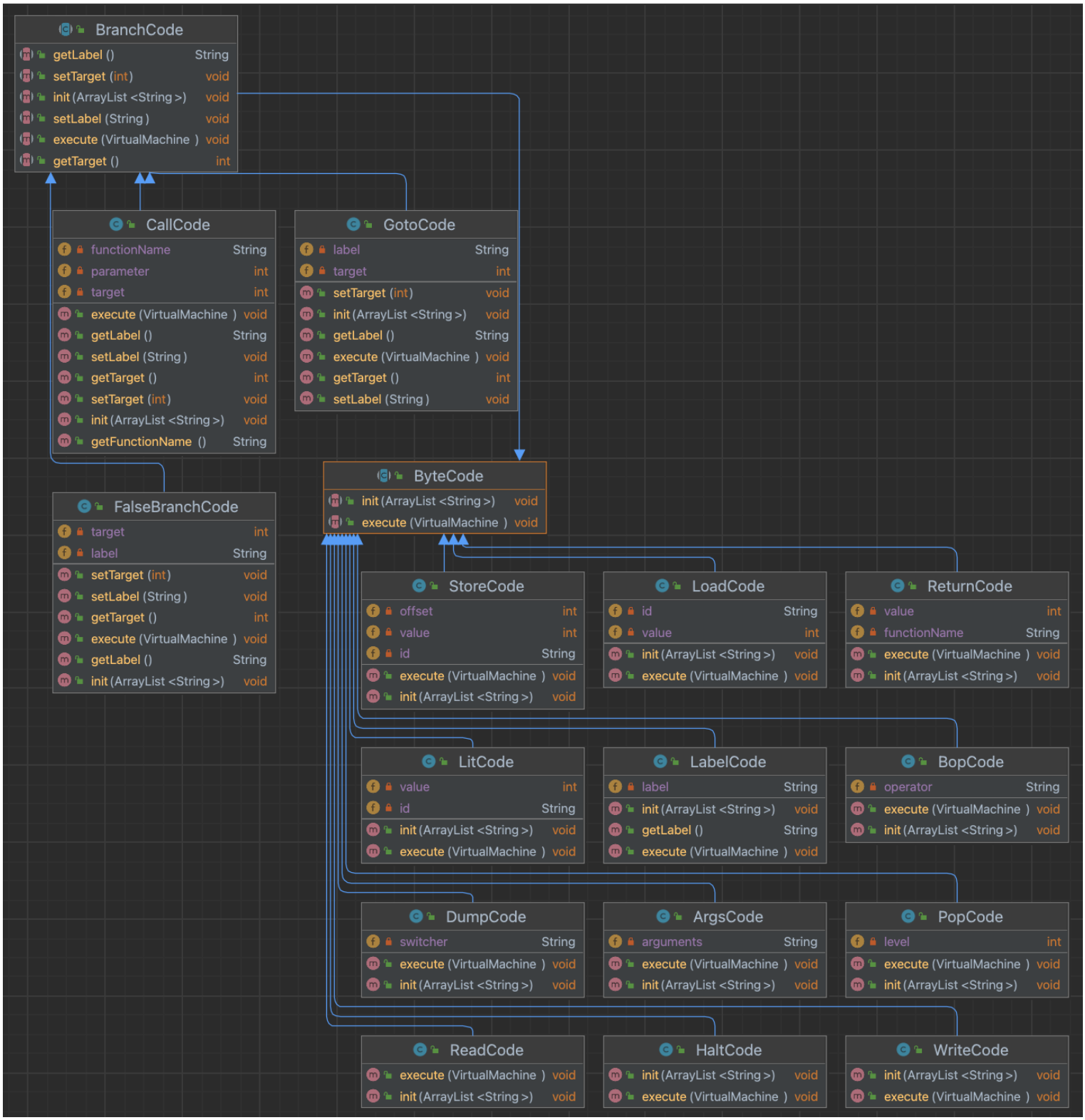
This was probably the longest class to implement out of all the classes in the project. However, most of the functions were straightforward (getters and setters). The most complex method to implement for me was the dump method. To implement this method, I decided to create a new instance variable "frameIndex" that simply tracks the index of the vector that a new frame was created. I used this to find the end of the runTime stack and the beginning of the temporary framePointer stack if applicable. In order not to modify the runTime stack, I used a temporary arrayList to hold the frame stack and print the contents after the for loop.

Code Organization

For better organization, all bytecode class files are inside the subpackage bytecode, inside the package interpreter. Also, all instance variables are private to each class and can be accessed through getter methods.

Class Diagram




BranchCode
getLabel () String
setTarget (int) void
init (ArrayList <String >) void
setLabel (String) void
execute (VirtualMachine) void
getTarget () int
CallCode
functionName String
parameter int
target int
execute (VirtualMachine) void
getLabel () String
setLabel (String) void
getTarget () int
setTarget (int) void
init (ArrayList <String >) void
getFunctionName () String
GotoCode
label String
target int
setTarget (int) void
init (ArrayList <String >) void
getLabel () String
execute (VirtualMachine) void
getTarget () int
setLabel (String) void
FalseBranchCode
target int
label String
setTarget (int) void
setLabel (String) void
getTarget () int
execute (VirtualMachine) void
getLabel () String
init (ArrayList <String >) void
ByteCode
init (ArrayList <String >) void
execute (VirtualMachine) void
StoreCode
offset int
value int
id String
execute (VirtualMachine) void
init (ArrayList <String >) void
LoadCode
id String
value int
init (ArrayList <String >) void
execute (VirtualMachine) void
ReturnCode
value int
functionName String
execute (VirtualMachine) void
init (ArrayList <String >) void
LitCode
value int
id String
init (ArrayList <String >) void
execute (VirtualMachine) void
LabelCode
label String
init (ArrayList <String >) void
getLabel () String
execute (VirtualMachine) void
BopCode
operator String
execute (VirtualMachine) void
init (ArrayList <String >) void
DumpCode
switcher String
execute (VirtualMachine) void
init (ArrayList <String >) void
ArgsCode
arguments String
execute (VirtualMachine) void
init (ArrayList <String >) void
PopCode
level int
execute (VirtualMachine) void
init (ArrayList <String >) void
ReadCode
execute (VirtualMachine) void
init (ArrayList <String >) void
HaltCode
init (ArrayList <String >) void
execute (VirtualMachine) void
WriteCode
init (ArrayList <String >) void
execute (VirtualMachine) void

Results and Conclusion

This project was a good learning experience about bytecodes and how an overall program is interpreted behind the scenes. While it was challenging and frustrating at times dealing with a very large project, it was rewarding to finally be able to understand the overall premise of this assignment. All required classes and methods were implemented and tested successfully on four different bytecode files.

Challenges

With this being the fourth assignment, I expected the contents to be very challenging so I wasn't surprised while examining the code skeleton for the first time. However, while seeming scary at face value, after working on it for some time, I realized that it's less complicated than the parser assignment. Overall, RunTimeStack and program classes were the most challenging to implement and test.

Future Work

In the future, we can implement interpreting behavior to more data types such as strings or floats. We can also create a debugging interface using the dump bytecode we created in this assignment.