CSC 413-02

The Interpreter

Joshua Agulto

915304694

https://github.com/csc413-02-spring2019/csc413-p2-joshagulto

Table of Contents

Introduction

Project Overview

This program is designed to implement a mock language and execute commands from a file. The interpreter will process various byte codes created from source code files with the extensions ".x.cod".

Technical Overview

The purpose of this project was to utilize many types of data structures to implement a mock command language. The project was designed to execute various commands at the input of a user. Our job was to implement several commands, classes extended from a class ByteCode, implement the ByteCodeLoader, implement a RunTimeStack, to track and execute the ByteCodes, implement Program class, to run the RunTimeStack, and implement a VirtualMachine, to run the mock program. The Interpreter and the CodeTable have already been implemented. CodeTable uses a HashMap to recognize the string input from the file and interpret it as a command. Interpreter is used to interpret and determine which commands are being done. RunTimeStack, VirtualMachine, and all of the ByteCode classes all needed to be implemented while the others had several functions. Our job was to harmonize all of the functions to run cohesively.

Completion

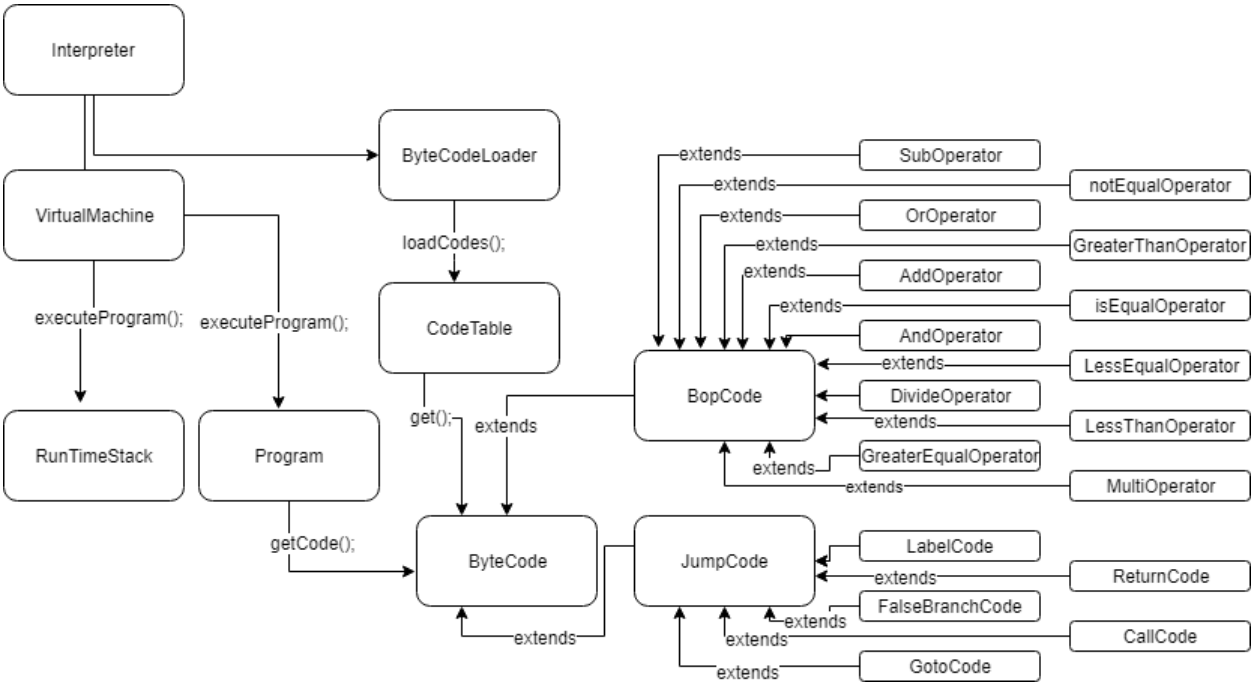
The project is completed. All classes, RunTimeStack, VirtualMachine, Program, ByteCodeLoader, RunTimeStack, and all of the ByteCodes have been fully implemented and should be working properly. The program will only operate assuming there are no errors within the x.cod files.

Build Information

This program was written using IntelliJ IDEA 2018.3.4 from JetBrains. The Java Development Kit that was used was jdk11.0.2.

The program can be downloaded from the specified link above. With the link, there is an option to download the repository to a zip file and an option to clone the file via Git Bash. When the file is downloaded and unarchived, it can be imported to any IDE of choice. Because this program was implemented on JetBrains' IntelliJ IDEA, it is recommended that the project be ran and executed on IntelliJ IDEA. To import on IntelliJ IDEA, you can go to File > Import > Open Project and import the folder. Once the project is imported, you can run the program. To run the program, navigate to the Interpreter class and run Interpreter.java.

Implementation



Interpreter uses VirtualMachine to process all the information. Interpreter is essentially the heart of the cell and the gateway to using the program. The interpreter class calls for VirtualMachine to execute Program and RunTimeStack. Program uses several functions, primarily getCode(), to access the ByteCodes. Program uses ByteCodeLoader to access the ByteCode functions. ByteCodeLoader uses a HashMap within CodeTable to understand a string and iterate it as a ByteCode. Each ByteCode is abstract and extends to several classes. Given the amounts of ByteCodes, there were not included in the diagram but extend directly from ByteCode. BopCode and JumpCode also extend ByteCode but they are their own abstract classes that extend to their own sub classes as well. BopCode is an abstract class for all the binary operations that can be executed. All BopCodes have an execute(), toString(), and getOperator() methods. All BopCodes all have the same init() method. JumpCode is an abstract class for all of the ByteCodes that require labels or "jumping". In this case, Jumping means referring to another address. All JumpCodes share the same init(), setJump(), getLabel(), and getJump() methods. Each JumpCode has their own execute(). ByteCode has an abstract init() and execute() and are implemented differently per class.

ByteCode loader uses several methods to understand the ByteCodes given by the user. The function ByteCodeLoader uses a BufferedReader, byteSource, to understand the codes given by a file. These codes are then passed to loadCodes() and to interpret each of the codes. The loadCodes() method uses several for loops and conditions to determine whether the code is a stand ByteCode, a JumpCode, or a BopCode. ByteCodeLoder also has an operation hasLabel to make determining whether the code is a JumpCode a lot easier.

Program is used to retrieve the ByteCodes. ByteCodes are stored into an ArrayList so the VirtualMachine can keep track of all the addresses of the ByteCodes. The addByteCode() functions adds a ByteCode to the ArrayList. The ArrayList, program, is a private list that stores byteCodes and can only be accessed by using the resolveAddrs() method. This method is used to recall the ByteCodes at particular addresses. Other functions, getSize(), isLabel(), isJump(), are self-explanatory and are single line methods.

RunTimeStack is one of the classes that needed to be fully implemented. This class did not have any methods filled out so it was up to the student to decide what functions the RunTimeStack will execute and what variables and classes it was going to have access to. The purpose of RunTimeStack is to keep track of active frames. The RunTimeStack has several functions, similar to a standard Stack in Java.

VirtualMachine is the other class that needed to be fully implemented. The purpose of VirtualMachine is to act as the powerhouse of the cell. VirtualMachine has access to most of the classes and practically implements the entire project. VirtualMachine implements a standard stack, returnAddrs, and a RunTimeStack, runStack. These are used to keep track of the different addresses of ByteCodes acquired from the executeProgram(). Most functions in VirtualMachine are self-explanatory as they are single line methods. The purpose of executeProgram() is to create a RunTimeStack and an address Stack. The method then calls its private program object to execute the ByteCodes acquired from a given file.

Reflection

Assumptions

My initial assumptions when designing this project was that it was going to be time consuming and require a lot of tedious effort. A lot of the projects I've done in my coding classes prior would mostly be skeleton codes and require us to do the implementation. This project was a little different because the implementation, although there were steps, it seems that the design was completely open ended. Because of this, it required a lot of attention and understanding of what the concept of the interpreter was essentially asking. At first, I didn't really understand the interpreter at all. I didn't really understand what the interpreter was asking me to do or understand what we were supposed to do in general. In the end, the project did require a lot of time and effort. The concepts to execute in the project were not that hard to do but the overall concept to understand and how the interpreter works was really complicated. This project served as a good building block for upcoming projects.

Reflection

This project was absolutely the hardest project I have ever had to do in coding. When starting this project, I had absolutely no idea what to do. Unfortunately, I missed the first day of class that the professor talked about the project, so I pretty much went in to the project completely blind. This project really set in my mind how hard coding projects can actually be. The hardest part of the project was definitely implementing VirtualMachine. When I was doing the code, I attempted to do the project according to the recommended sequence in the instructions but I found that particular method a little bit difficult. The suggested order the instructions offered were to create, but not implement, all of the ByteCode classes. The next was to implement ByteCodeLoader. Everything was okay for me up until the next step. The next suggestion was to implement Program. Particularly, it was suggested to implement resolveAddress method. The idea behind resolveAddress was to acquire the address of a code. Unfortunately, because there are cases that the ByteCode could be a label with a jump index, I was not sure how to completely implement it. Next was to implement VirtualMachine and RunTimeStack. The hardest part of figuring out how to implement VirtualMachine and RunTimeStack was understanding what needed to be done. Since there were a couple functions given in the instructions, I began with those. The reason I found implementing each class hard was because I was not really understanding what exactly I was implementing. I was simply trying to put concepts into methods but I couldn't figure out what to do.

The way I figured out how to go about this project was to do it method by method. Rather than working on a particular class at one time, I found it substantially easier to implement a particular idea first. For example, rather than implementing the several classes asked in the instructions, I decided to implement the methods without the program counter or frames. I first began to focus on how each of the ByteCodes will be retrieved and instantiated into the VirtualMachine. When I figured that out, I then started worrying about how I was going to retrieve the addresses of each of the ByteCodes. Once I got that down, I could

start worrying about frames and the use of the program counter. I was essentially coding each class little by little every time rather than implementing an entire class.

One idea that I thought was really good that I would not have thought of myself was to make an abstract class for the BopCode and JumpCode. Originally I tried to do all of the binary operations within the BopCode. I was going to setup a bunch of if and switch statements. I took a suggestion from another student to implement the ideas from the first project. I ended up making a bunch of operator classes, i.e. AddOperator and SubtractOperator, within their own execute() functions. This made it substantially easier since the concept was already executed in a previous project. I decided to implement the abstract class idea with the JumpCodes as well since they all required a "jump index".

Conclusion

This project overall was extremely tough. I found the hardest part of doing this project was simply understanding how to even start it. I'm not used to doing projects that are partially implemented. A lot of the projects I did before were either skeletons codes with all the methods but no implementation or a project completely empty and required us to design it from scratch, although all those projects were in the lower level programming classes. I had a fun, although extremely stressful, time doing this project and it does, in fact, serve a great purpose although I'm not 100% that my project is ok. There does not seem to be any errors within the project but I wasn't able to execute the cod files. I haven't joined Slack yet because I keep procrastinating it and forgetting to do it.