CSC 413 Project Documentation

Summer 2022

David Ye Luo

Student ID: 917051959

Class Section 2

GitHub Repository Link

Table of Content

Introduction

Project Overview

This project is about building an interpreter program for a specific mock programming language. An interpreter program isn't too different than an interpreter in person. The difference is who or what the interpreter is working for. An interpreter person works from person to person(s) whereas a program interpreter works for a programmer to the computer. When a programmer writes code for a software application like a calculator, the programming language they use is often human-readable. It then goes through a process to convert the code into machine instruction to finally execute the code. However, the interpreter part of this project is the last part where it takes the machine's instruction to execute it. It is like teaching someone how to cook a recipe but the person doesn't understand it. The scope of my program assumes that the person understands the recipe language and teaches what to do for each step. Like, use a measuring spoon when the recipe mentions measurements. What motion to do when it says whisking something. Or how to use turn on and set the right settings for a stove. These are basics but are necessary to try to recreate a recipe. An interpreter is similar in that it takes an intermediate language that the computer understands and does those instructions.

Technical Overview

The interpreter in this assignment contains two important classes. The ByteCodeLoader takes in the machine instructions that the user will provide in a file and the ByteCodeLoader will load them in memory in the form of ByteCodes. Then we have the Virtual Machine which takes

these series of bytecodes and executes those instructions. The interpreter is simply a collection of these two classes where its purpose is to take in the path of the bytecode files and let the two classes do the heavy lifting.

Summary of Work Completed

Overall it might sound simple in theory but making it isn't as straightforward. I've worked from taking a file path to load bytecodes to running the program. Worked from the design and the process of the creation of bytecodes to how they will execute. Then lastly, implementing the virtual machine to provide the execution environment for each bytecode.

Development Environment

The Operating System that I'm using is Ubuntu 20.04.4 LTS but actually, I'm using Ubuntu 18.04.6 LTS on a virtual machine to work on this project.
The IDE that I'm using is IntelliJ Ultimate version 2022.1.2 with Java JDK 18 to do my coding and testing.

I also took notes, kept track of tasks, and some documentation in a markdown document outside of the project using the Linux terminal (Not in the virtual machine).

Lastly for preference, I used IdeaVim to work on the IDE with vim layout.

How to Build/Import your Project

1. Download IntelliJ and set up Java in it. (If you don't have IntelliJ)
2. Import project to IntelliJ
    a. Clone Repository through my GitHub.
    b. Import the repository to your IntelliJ
        i. **Import it from the folder**

How to Run your Project

1. Change the directory to the evaluator package.
2. Run the interpreter.java file and pass in a file path of the bytecode file you want to run on the command line argument.
    a. There are some example files such as factorial.x.cod that you can try.

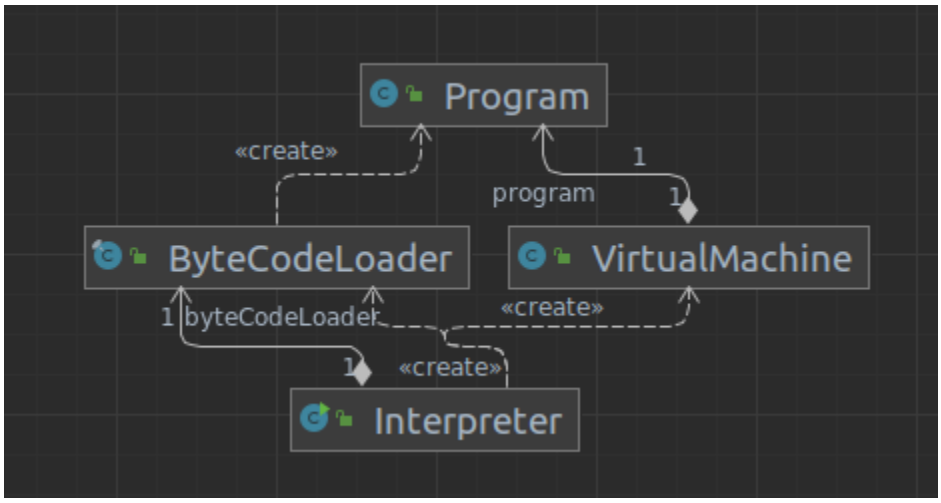
For Junit tests, there are jars included in the resources folder if you want to add more tests.

Assumption Made

The interpreter assume that the bytecodes provided in the files are correct. There are cases that the interpreter will catch error for certain bytecodes. However, not all ByteCodes does this. Even if it catches, there isn't a guarantee for a bug free program.

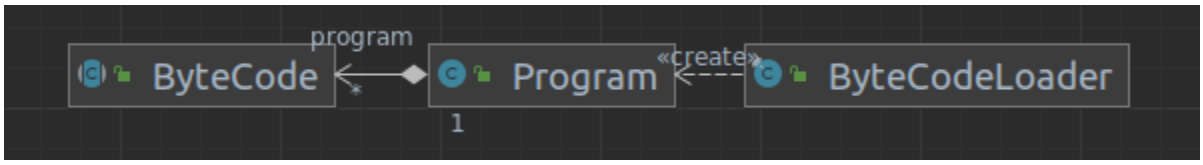
Implementation Discussion

Interpreter



The interpreter class simply takes in console argument which should be the file path of the bytecodes. The secondary part is creates and use ByteCodeLoader and the VirtualMachine class. It's purpose is to take the ByteCodeLoader and create a Program class with the bytecodes. Once a program is created then it is passed to the virtual machine to execute the code. The interpreter class is provided and there wasn't any changes to it.

ByteCodeLoader/Program

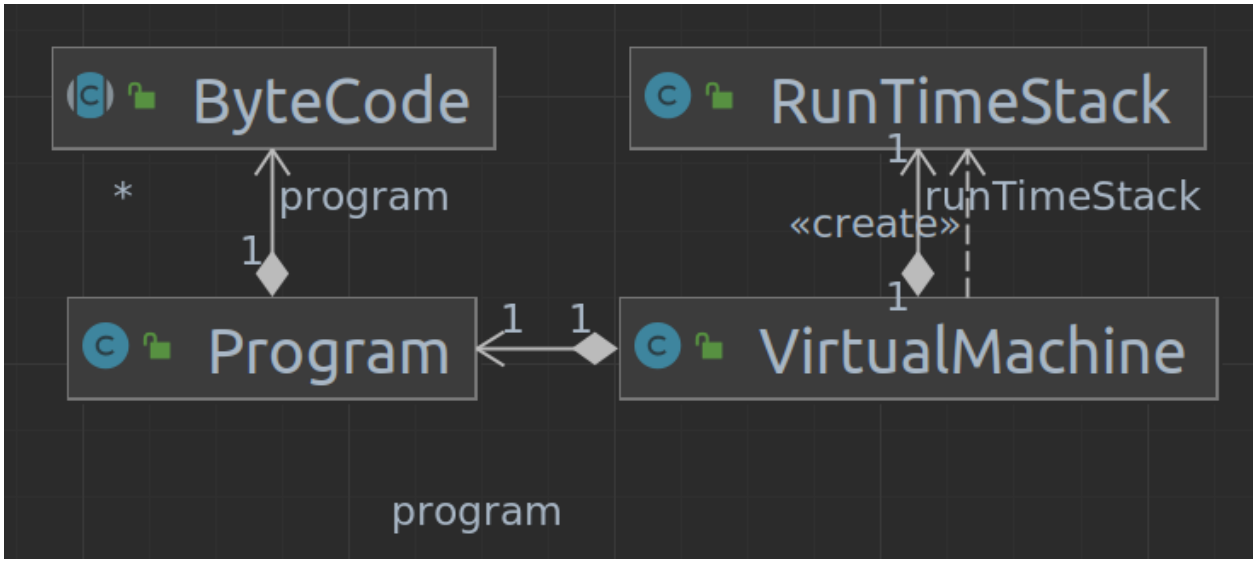


The ByteCodeLoader class will read and tokenize each line in the provided file. It will gather all the bytecodes to create a program object. A program object is simply the whole

collection of the bytecode. In the process, the loader class creates bytecode objects and sets up their arguments.

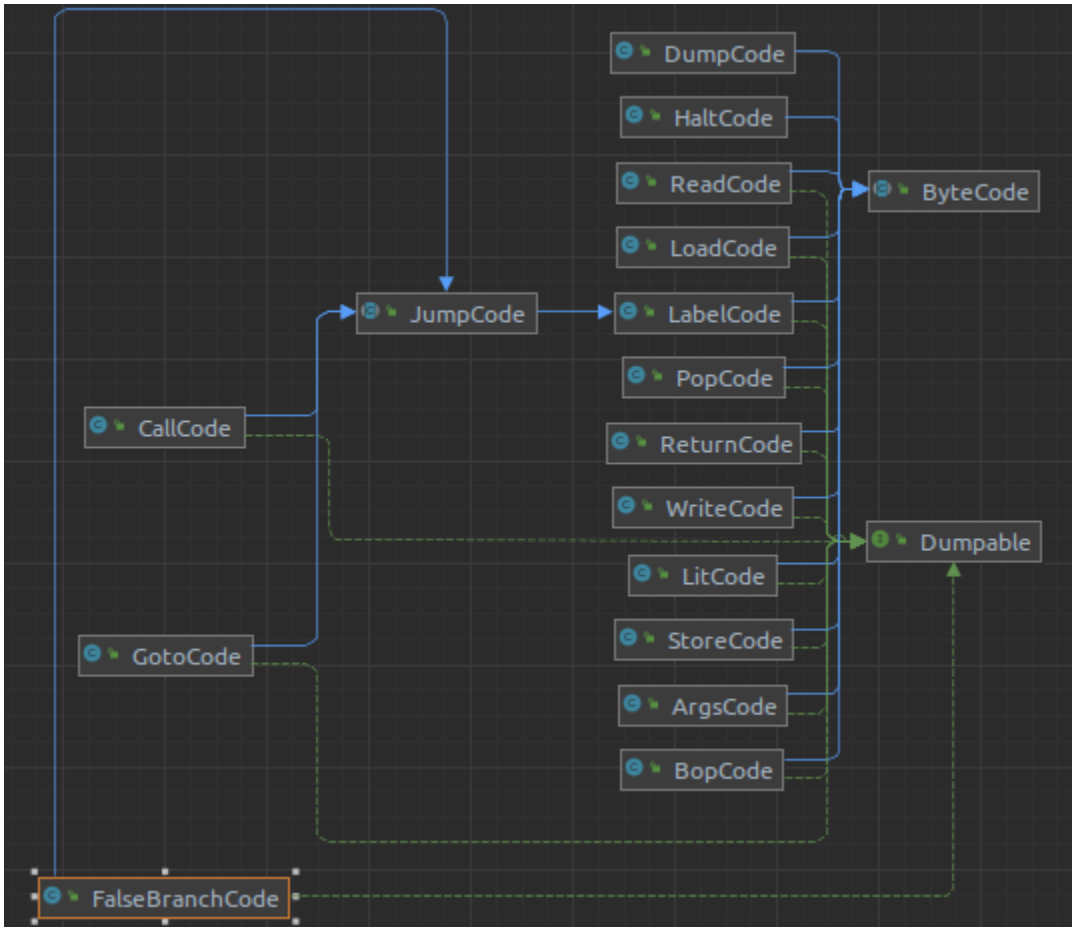
The Program class has one simple but crucial job. That is to resolve address for bytecodes. A branch bytecodes has instruction to jump to a label. However, a label isn't a known location. Resolving the address will solve this issue. My implementation of this is using a hashmap to gather all store labels bytecodes and its location. Then, it will find all branch bytecodes that I named them JumpCode and give them the location.

ByteCode/VirtualMachine/RunTimeStack



The virtual machine executes the programs from the first bytecode until it reaches the HALT bytecode. The virtual machine uses a stack data structure known as the RunTimeStack as a program memory. The runtime stack is a stack but puts a boundaries of where the program can operate. The bytecodes have an indirect access to the runtime stack through calls to the virtual machine. This is so that the virtual

machine protects the runtime stack from bytecodes trying to access data where it shouldn't. Such methods are pop() load() and store().



Each bytecodes mostly falls into two main categories which are JumpCode and Dumpable. Anything that extends from JumpCode will have their address resolution solved. Lastly, anything that implements Dumpable allows the bytecode to dump information to the screen.

For the BopCode, I used part of the calculator project from assignment 1 so that this bytecode can do basic math operations. Logical operators were new addition to do logical operation.

Project Reflection

The amount of work of effort that I put to this project was overwhelming. For the amount of time I had, I was worried that I felt like I wouldn't get pass compiling successfully. Let alone completing the assignment. I've started with writing tests using JUnit for the runtime stack and the virtual machine. I wanted to also test the bytecodes and have an integration test which was very useful. However, I dropped making tests and decided to use a debugger instead when time was an issue.

The biggest time sink for me was getting the newFrameAt() working. I was getting confused on which number to use when calculating the offset from. I had an idea that it was the runtime stack size and not the last index. Even though my interpreter works now, we can still see in the failed tests in my JUnit class that indicates the confusion with which to use. However, I had unnecessary code that was supposed to protect ByteCodes that want to set a frame beyond the current frame ended up causing trouble. Thankfully, professor helped me solve this issue.

Project Conclusion/Result

In this project, I further developed my programming skills by using programming design patterns to structure my code and followed the principle of Object-Oriented Design. Also, JUnit tests are very useful and powerful but they are time consuming to make. Unit test becomes a challenge when there are dependencies. However, I find that using a debugger was just as important skill to have and it was the strategy that I switched to when dependencies were annoying to deal with. It was also a great tool to trace through what your code is doing and look

at what the values are at a particular time. This taught me to persevere through a big project even when I think it is impossible. It also encouraged me to create unit tests and using a debugger. Lastly, it helps me appreciate the use of abstraction to solve complex problem.