CSC 413 Project Documentation

Fall 2019

Andrew Hwang

918450486

CSC 413-02

https://github.com/csc413-02-spring2020/csc413-p3-AndrewHwang97

Table of Contents

1 Introduction

1.1 Project Overview

When it comes to writing code for a program, there are many aspects that go into it. One of the main aspects of coding is the language used for programming. A programming language is one consisting of different forms of instructions that carry out functions for a program. There are many languages as of today and they mostly do the same thing, but the way that they are written are structured differently. Some languages are even well suited for certain situations over others. The interpreter program is a program designed to understand the programming language X. the program reads the code line by line, and determines what functions to do based on what that line says.

1.2 Technical Overview

The interpreter program is one that is designed to read code written in language X. Language X is written with ByteCodes to carry out functionality. Taking a look at the program as a whole, it consists of many files that work together to get the interpreter working. The structure the program has consists of RuntimeStack, VirtualMachine, Program, CodeTable, ByteCodeLoader, and ByteCode Classes.

The RuntimeStack class is one that implements a Runtime stack for the program as well as a frame pointer stack, one that is used to mark activation frames. The runtime stack is made from an arraylist, which has functions that help it act like a stack, such as pop(), push(), etc.

The VirtualMachine class is the class that acts like the controller of the program. It is in charge of carrying out functions of the runtime stack and frames. All the bytecodes and other classes reference the virtual machine for functionality so that they are not directly accessing the runtime stack itself. It follows a loop:

> While the virtual machine is still running, we get the ByteCode from the program, and then we execute the bytecode. if dumping is on, then we print the information to the console. The Halt ByteCode changes the state of the virtual machine so that we can finish the program.

The program class is designed to store the program itself, containing all the Bytecodes of the program. It also has the functionality to resolve addresses of the files. This is important for the program as a whole because we have ByteCodes in the program that jump to different lines of the code, and we need the addresses resolved so we can process the jumps. How a resolved address works is that It scans through the file, and looks for specific labels or jump codes. If the line has a label or jump code, it stores it to a Hashmap consisting of that label and the address of the label. The program class is vital for the program as a whole.

The CodeTable class is one that has all the Bytecodes in a hashmap that stores the name of the ByteCode, and the corresponding String of the ByteCode Class. The ByteCode loader class uses this class to reference the ByteCodes. The ByteCodeLoader class is designed to read the bytecodes line by line, tokenize the strings up into arguments, and create ByteCode objects for the program. it follows a loop:

while there are more lines to be read from the program, we tokenize the string,

We get the className based on the first element in the string, and then we get arguments that are all the elements that follow the class name, we then initialize the class, and then push it to the program.

The ByteCode Class is an abstract class that all other ByteCodes reference. It contains an initialize and execute function. We also have a jumpcode abstract class that extends the ByteCode class, but has extra functions that the jump ByteCodes need for implementation.

1.3 Summary of Work Completed

Some starter code was given at the start of this project. The RuntimeStack class was implemented with methods that gave it the arraylist stack-like functionality. The ByteCode abstract class was implemented with the methods init(), and execute(). All ByteCodes were implemented by creating files that extended the ByteCode class. The JumpCode class was implemented to extend the Bytecode class that added some functionality for ByteCodes that involved jumping addresses. After the ByteCode Classes were completed, the ByteCodeLoader class was implemented specifically the loadCodes() function. The program class's resolveAddress() class was implemented along with the virtualMachine's while loop.

2 Development Environment

For this project, Java version 10.0.2 was used. The Intellij IDE was used to complete this project.

3 How to Build/Import your Project

To build/import the project to the Intellij IDE, the project will need to be downloaded from github. Once the project is downloaded from github, open Intellij. Once Intellij is opened, Click on the "Project from existing sources" Under File > New. Find the folder containing the project, highlight the entire project folder, and hit the 'next' button. A series of options will appear. The default options are sufficient to use for the project.

4 How to Run your Project

to run a file, goto the run/debug configurations menu, and set the main file to interpreter.interpreter. and you can change the file used by inputting that into the program arguments line. The project should be able to run after.

5 Assumption Made

Some assumptions that have been made when working on the project included assuming that the code written in the cod files are written correctly without errors. if there are errors within the cod files, the program could possibly crash. Another example of assuming that everything is written correctly is the case where there could possibly be overflow of data, such as using a big number for the factorial program.

6 Implementation Discussion

The goal for this project was to keep consistency and scalability of the project. For example, if someone else wanted to add on to the project, the goal is to not have to add a lot more code to the project. one way this was achieved was by using the Class constructor function. In the ByteCodeLoader class, we take the first argument in the tokenized string and compare that to a hashmap, which then takes the correct ByteCode associated with that first argument, and constructs it based on that value in the hashmap. This achieves scalability for that If we wanted to add a new ByteCode, we would only have to input the ByteCode in the hashmap in order for the program to continue to work. Another example of Scalability is how Jump Codes were implemented. instead of hardcoding all the JumpCodes in the Program class, we instead checked if the ByteCode was of a JumpCode class, which was made as another abstract class that extended the ByteCode class, and if a ByteCode is a JumpCode, It would be implemented so. In the Program class. In the case of anyone wanting to add a new ByteCode that jumped addresses, All they would need to do is add the ByteCode class and extend it to the JumpCode. Another implementation goal for this project was to keep principles of encapsulation in tact. This was achieved by leaving fields private in their respective classes and have the virtual machine access them with built in functions. The UML diagram can be found in the documentation folder of this project.

7 Project Reflection

I feel like the project went fairly well. I finished the project with little time to spare. This project taught me how to work with a timeline and stick to it. As big and daunting as this assignment was, I feel like taking the time to read the assignment more than twice definitely helped out a lot with digesting what needed to be done and what the program actually did/how it works. I feel like understanding how the program should work beforehand eases the process as we know what to expect. The assignment has also taught me alot about breaking a problem down into smaller chunks and then bringing it all

together. In the beginning of the assignment, I was overwhelmed with where to start in terms with the assignment, but as said in class, I had started to work on the Runtime stack first, and only that. finishing that gave me the confidence into starting the ByteCodes, and as I was doing the ByteCodes/attending lectures, I felt like I was really beginning to understand more and felt more confident in tackling the project. Debugging was also a turning point in this assignment. I had broken the code at some point and going line by line and into various functions definitely helped me solve/fix the bugs that had been made.

8 Project Conclusion/Results

for this project, we were given the task of creating an interpreter that could process the language X. The project utilized abstraction as well as encapsulation. I also consisted of some scalability principles for some of the classes that were involved There were a great amount of different files/classes that were worked on in this project that came together and interacted to create the interpreter, all of them going through the Virtual machine that acted as a controller for the program. As a result, assuming that all the code that is coded in language X is correct and without error, the program is working correctly and is intended to do.