CSC 413 Project Documentation

Fall 2019

Mohamed Farah

918378258

CSC 413-03

https://github.com/csc413-03-fall2019/csc413-p2-FarahMo24

Table of Contents

1 Introduction

1.1 Project Overview

For this assignment, we are tasked to implement an interpreter for a “Language X ”. Language X is a simplified version of Java, and the interpreter sole job is to process the byte codes. The byte codes here are created from the source files with the extension x. The two source codes that we’ll look at for this assignment are a Fibonacci program and a Factorial program, both of which are recursive

In this assignment, the interpreter and the Virtual Machine will work together to process and run the “Language X”. And the way we can think of this Virtual Machine is that it’s the main controller for this program, all operations will go through the Virtual Machine. The number of byte codes we have to handle are about 15 types and thus all of which will be handled by the Virtual Machine indirectly.

1.2 Technical Overview

In this assignment our main focus starts at the Interpreter. In the Interpreter, we have a class called ByteCodeLoader, which loads the bytecodes line by line using the BufferedReader. During this process we have another class helping us with the naming conventions and this class is called CodeTable. CodeTable class, which is heavily used in ByteCodeLoader class, stores a HashMap that maps between our bytecode, that are loaded in, and the bytecodes respected classes. Once the ByteCodeLoader class is near its end, all the objects are stored in a Program object. The Program class main idea is to store all the bytecodes in an ArrayList in its custom ByteCode type. Program class also resolves any and all symbolic addresses. We then move on to our Virtual Machine object, which processes our Program object.

The Virtual Machine, or VM, is the controller of this program as stated above. All operations need to go through this program and the VM keeps track of the file’s index as it executes the bytecodes. The VM uses encapsulation in a way that it uses functions that are called by individual bytecode subclasses. Each function that is called upon by such subclass must go through the VM in order to make changes to the runtime stack. This is a indirect approach in accessing the RunTimeStack class.

1.3 Summary of Work Completed

In this projected I started out by working on ByteCodeLoader class. As I started to understand what was being loaded in, I used the class CodeTable to help me create an abstract class called ByteCode. During the rest of the assignment, as gaps started to fill in, I slowly worked on and implemented each of the subclasses needed for ByteCode class.

The next phase of the project was the Program class. I first worked on implementing the ArrayList and adding a helper function to add to the ArrayList from the previous ByteCodeLoader class. Once that was completed, I worked on the resolveAddrs function which was able to resolve target addresses such as a GOTO jump.

Once the first half of the project was completed, I then started both RunTimeStack and VirtualMachine class. RunTimeStack had a lot of functions to implement, these functions were then used by VM to make changes to the class. VM was more difficult because at the same time I was working on the Bytecode subclasses, all of which interact with the VM.

2 Development Environment

IntelliJ IDEA 2019.1.3 (Community Edition)

Build #IC-191.7479.19, built on May 27, 2019

JRE: 1.8.0_202-release-1483-b58 x86_64

JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o

macOS 10.15

3 How to Build/Import your Project

1. Navigate to GitHub with the link provided on Page 1 of this document.
2. Clone the project to a folder in your desktop that is easily accessible.
3. Make sure that your IDE is updated to the current Java version
4. Open IntelliJ or your preferable IDE and navigate to your cloned folder and import

5. Follow your IDE instructions after importing.

4 How to Run your Project

1. Navigate to your Interpreter file
2. Click on Run on the top menu bar
3. Navigate down to “Edit Configurations”
4. Under the Program argument type in either factorial.x.cod or fib.x.cod
5. Click on Run

5 Assumption Made

The first assumption is that the user will always input a valid integer when asked for an integer. The user here could enter a big number that we can’t handle, and the user can enter a non-integer.

Another assumption to be made is the file being read in has a valid string bytecode. Having a valid bytecode being fed in will work correctly in the program otherwise it would not be suitable for this particular program.

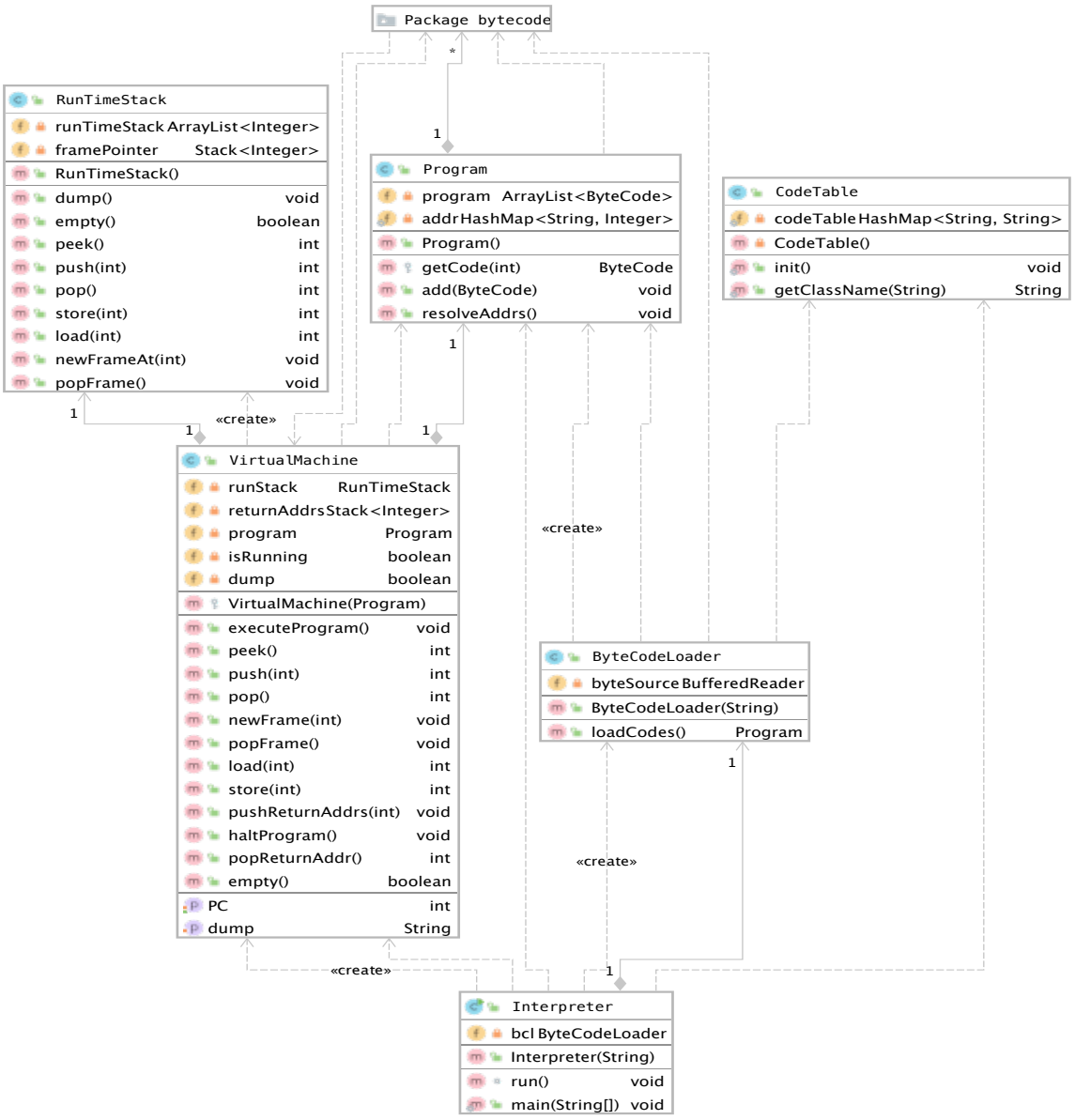
Last assumption is that the operations are not extended beyond the operations we already handle here. If we receive an operation beyond our scope for this assignment, the program will not function.

6 Implementation Discussion

6.1 Class Diagram

For this assignment I’ve had a difficult time in creating a UML Diagram. Though in the previous project I had drawn it out by hand, for this project I could not create a better UML than the one I have created using the IntelliJ diagram program. From the below UML diagram, I had trouble opening up the bytecode folder hence it being closed in the diagram

Overall the diagram shows the outlined explanation from the previous section of this discussion. The interpreter class is the starting point for this program, once started the interpreter class loads the source codes into a program object, which is taken care of by ByteCodeLoader class and CodeTable class.



7 Project Reflection

This project by far was the most difficult project I've ever done in class and outside of class. The hardest part of this project was starting from zero, and really learning from the ground up. First lesson in this project was learning how a compiler worked and how interpreter works. Once

understanding the fundamentals and how bytcodes played a role, I began going over the project in details.

As time went by with this project, I followed many of the class hints, one of which was creating a Bytecode superclass and creating their subclasses by using CodeTable as reference. Eventually I had class help working out the details for this project and how I could better improve the errors I was receiving in my ByteCodeLoader class, especially how I implemented my loadcodes function from the beginning.

After completing loadCodes, I had no problem with completing Program class, because of how we went over it in class. RunTimeStack though, was a different problem that I struggled with immensely. From working on my VM, errors kept bringing me back to RunTimeStack. I had several functions that were not implemented correctly and through the days I seemed to understand how they worked properly. I did guess a few times and debugging the code helped tremendously.

8 Project Conclusion/Results

This project I have learned a lot, one of the biggest lessons I have learned going through this project was learning how to use the debugger. The debugger was extremely helpful in not only understanding how this program worked but also helping me catch mistakes in my OOP set up. This project did eventually compile and come to a correct output, however the design in output did take a hit because of how much time I spent with the encapsulation design. I had saved the output design for last and that's why my output seems off. I will be updating this project as time goes on and until the deadline. Overall, I had learned a lot in this small-time frame, and I plan on going back to this project with a different set of bytecodes and source code.