CSC 413 Project Documentation

Fall 2018

Alex Wolski

918276364

413.02

https://github.com/csc413-01-fa18/csc413-p2-AlexWolski

Table of Contents

1 Introduction

The aim of this project is to make a simple programming language. This program takes basic instructions from a file, processes them, and runs them. I worked on parts of this project with Amari Bolmer, Andrew Sarmiento, Kevin Reyes, Mubarak Akinbola, and Tim Wells.

1.1 Project Overview

This project consists of a virtual machine that executes byte codes, a runtime stack that manages the program's memory, and an Interpreter that processes .cod file. These three components work together to read, compile, and run a program.

1.2 Technical Overview

The **Interpreter** reads instructions from a .cod file one by one and translates it. It accomplishes this by relying on two other classes: **ByteCodeLoader** and **Program**. **ByteCodeLoader** reads the .cod file line by line and converts the statements into **ByteCode** objects. This is done with the help of **CodeTable**, which takes a ByteCode and returns the Java class associated with it. There is a **ByteCode** subclass for each of the instructions, such as POP or GOTO. The **ByteCodeLoader** stores these objects in **Program**, which contains an ArrayList of type **ByteCode**. This will be used later by the **VirtualMachine** to run the program. Once all of the instructions in the file have been translated, the **ByteCodeLoader** calls on *resolveAddress* in **Program**. This method searches for references to labels in the byte codes and replaces it with an address in the ArrayList. Once ByteCodeLoader is finished, the ArrayList of byte codes is passed on to the **VirtalMachine.**

The **VirtualMachine** loops through the ArrayList of byte codes and executes each of them. This is done by calling the *execute* method in each **ByteCode** object, which performs its own unique operations. These operations involve manipulating memory, so the *execute* method will request the **VirtualMachine** to perform these tasks. The **VirtualMachine** then forwards the request to the **RunTimeStack**.

The **RunTimeStack** has numerous methods to access, add, remove, or organize data in the program's memory. This is where the majority of the logic for the program itself is done. The **RunTimeStack** organizes the memory with an ArrayList that holds Integers. There is also a stack to hold the indices for each "frame." These two data structures work together to make a cohesive run time stack.

1.3 Summary of Work Completed

For this project, the majority of the program was made from scratch. Only the Interpreter and CodeTable were provided.

2 Development Environment

To create this program, I used the **IntelliJ IDEA 2018.2 IDE** along with **JDK 1.8.0.**

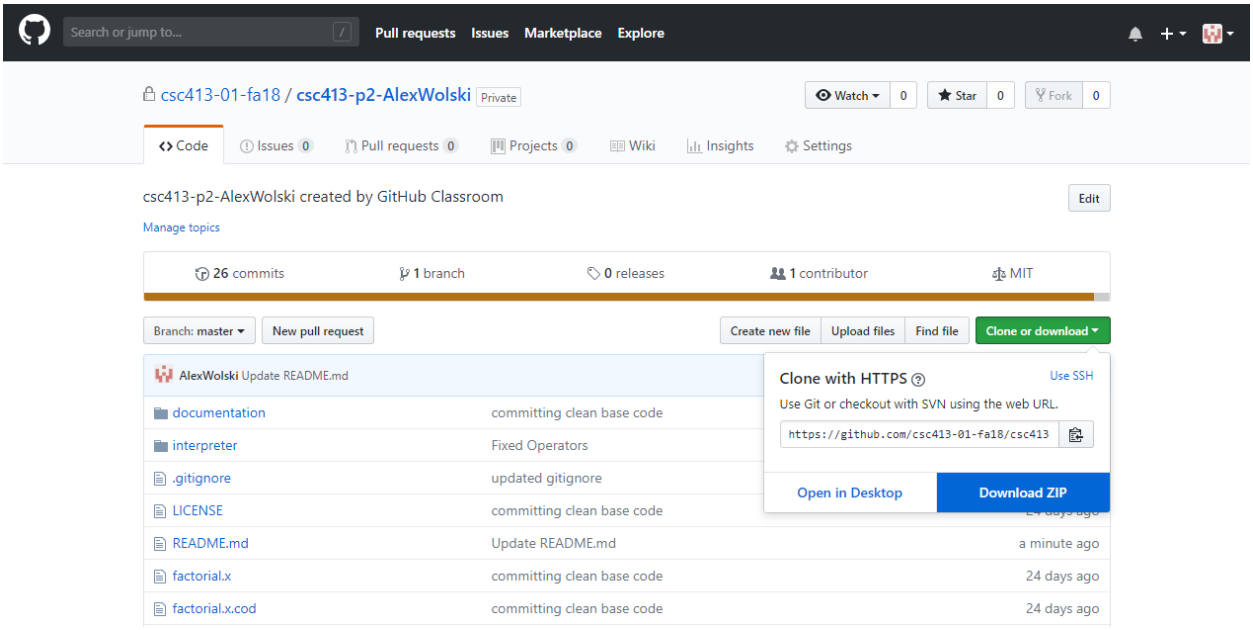
3 How to Build/Import your Project

Downloading:

To import this project, first download the files. This can done by running command in the command line:

git clone https://github.com/csc413-01-fa18/csc413-p2-AlexWolski

Or you can download the project in a zip file from the same URL:



Configuring:

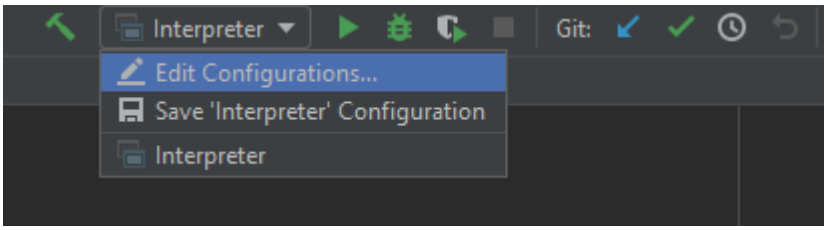
The entire IntelliJ project in included in the folder. So if you are using IntelliJ, you can simply open the “Interpreter” folder as an existing project.

If you are using a different IDE, you can import the project using “existing resources.”

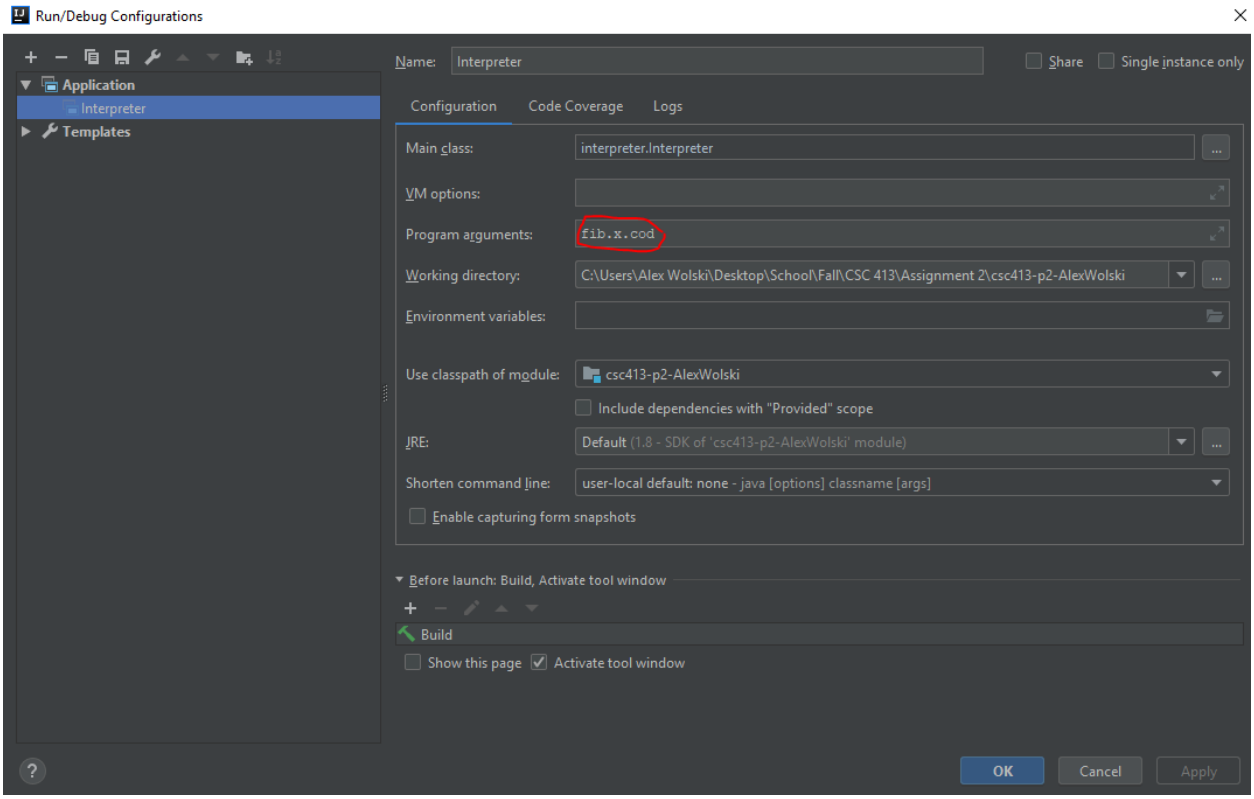
4 How to Run your Project

To run the project, you need a .cod file. There are two included in the GitHub repository: fib.x.cod and factorial.x.cod. Move this file inside of the csc413-p2-AlexWolski folder.

To pass this file to the program, you need to specify the file name in the Project Configurations. The easiest way to do this is to attempt to build the project by pressing the green start button. After an error is displayed, click on the box next to the start button. A drop-down menu will appear. Click on “Edit Configurations.”



A new window will pop up. Under the “Program Arguments” section, type in the name of the .cod file.



Lastly, click apply. Now you can use the green start button to run the program. For the fib.x.cod and factorial.x.cod files, simply enter the a position in the number sequence and the program will return the value in that sequence.

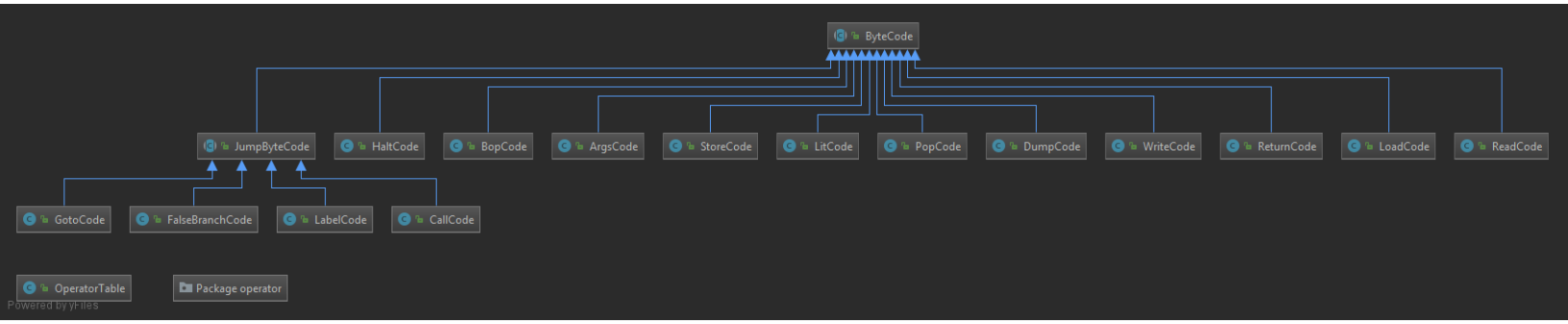
5 Assumption Made

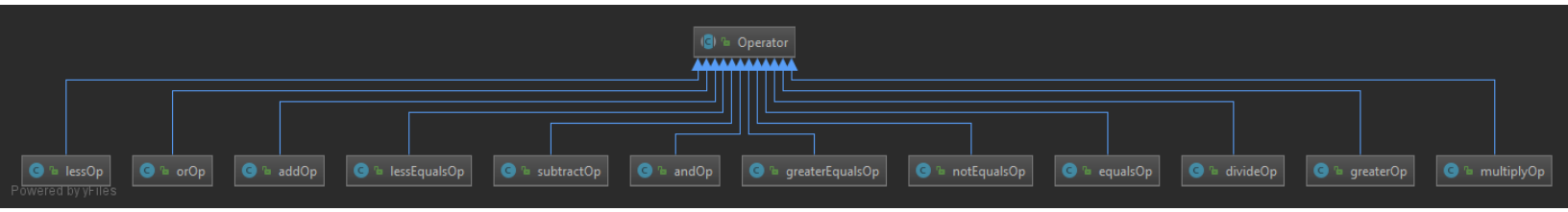
For this project, our clients allowed us to assume that the .cod files were written perfectly. This means that we didn't have to worry about syntax or logical errors. However, I decided to take the extra precaution and defend against these errors. This program will catch mistakes in syntax and terminate the program if there is a runtime error.

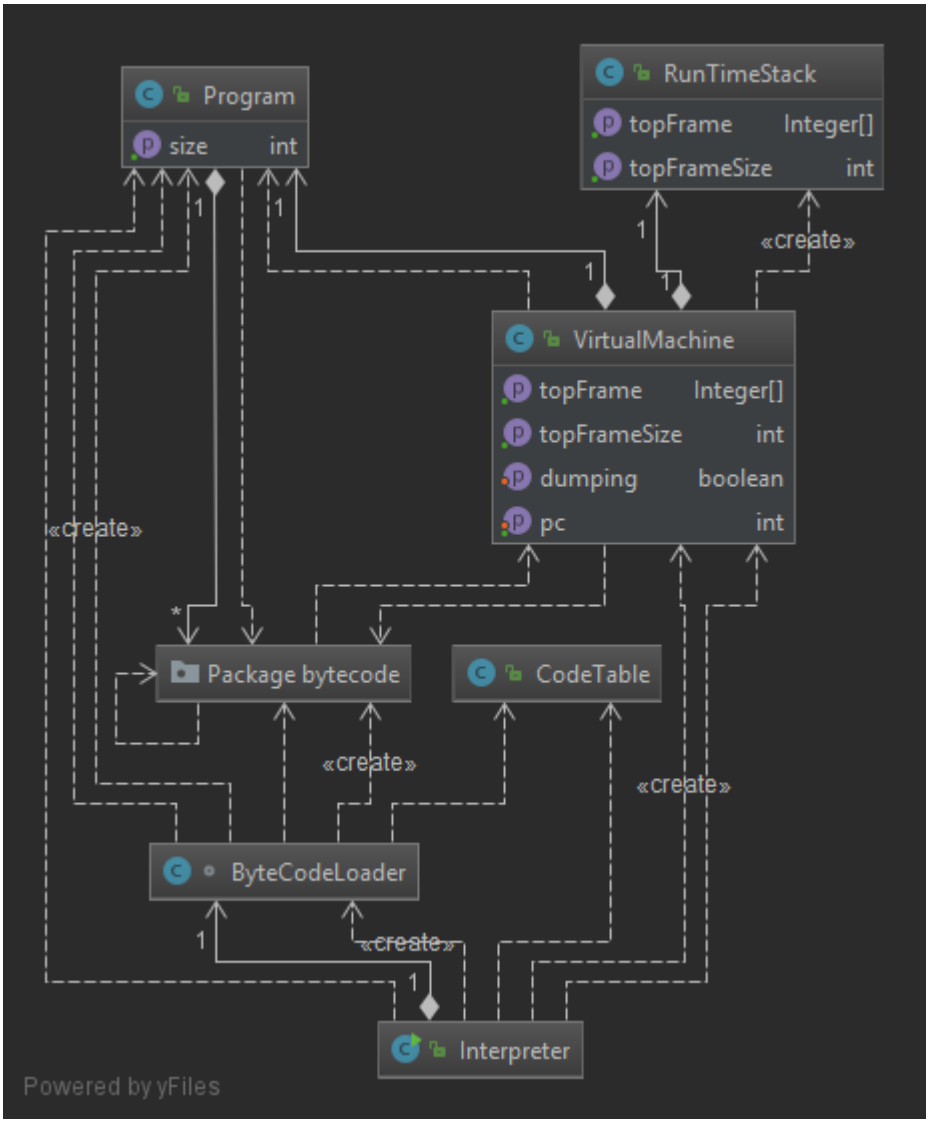
Although, I do assume that potential users of this program will understand how to use it. In its current state, it needs to be launched through an IDE to link it to a .cod file. I am making the assumption that any users will be able to compile the program into a .jar file and integrate it with their compiler.

6 Implementation Discussion

6.1 Class Diagram







7 Project Reflection

8 Project Conclusion/Results