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. I started by creating all of the **ByteCode** operator classes. They did not have implementations for methods, but I wanted to get the skeleton of the classes finished to help me design the rest of the program. After that, I worked on **ByteCodeLoader** and **Program**. Before rest of the project could be tested, there needed to be a translated set of **ByteCodes**.

Next, I worked on the **VirtualMachine** and the **RunTimeStack**. I implemented the methods that would be necessary for the **ByteCodes** to perform their operations.

Lastly, I went back to the method stubs in each **ByteCode** class and implemented them.

After the program was running, I worked on less vital features. These include the dumping feature, error checking, and the dynamic implementation of the **BopCode** class.

2 Development Environment

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

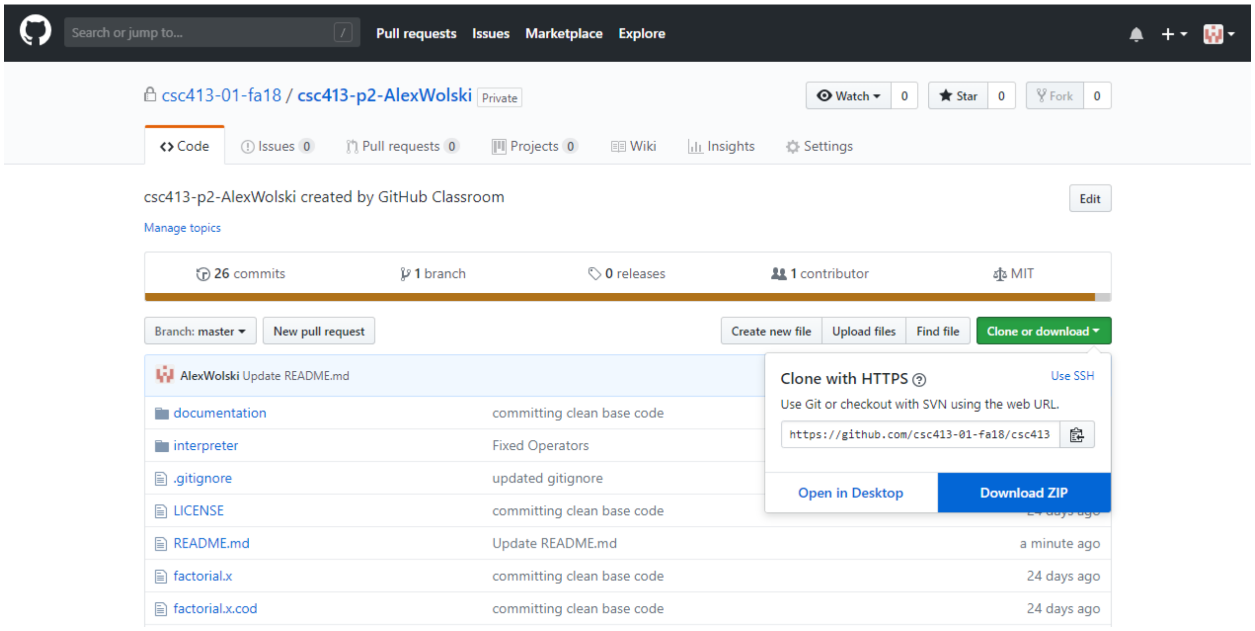
3 How to Build/Import your Project

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:



csc413-p2-AlexWolski created by GitHub Classroom

Use Git or checkout with SVN using the web URL.

`https://github.com/csc413-01-fa18/csc413`

AlexWolski Update README.md

documentation	committing clean base code	
interpreter	Fixed Operators	
.gitignore	updated gitignore	
LICENSE	committing clean base code	
README.md	Update README.md	a minute ago
factorial.x	committing clean base code	24 days ago
factorial.x.cod	committing clean base code	24 days ago

Configuring:

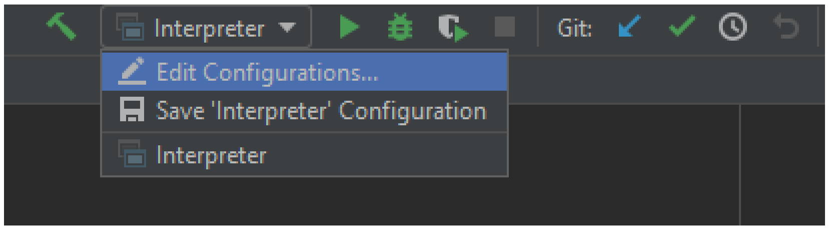
The entire IntelliJ project in included in the folder. So if you are using IntelliJ, you can simply open the “Interpreter” folder as an existing project.

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

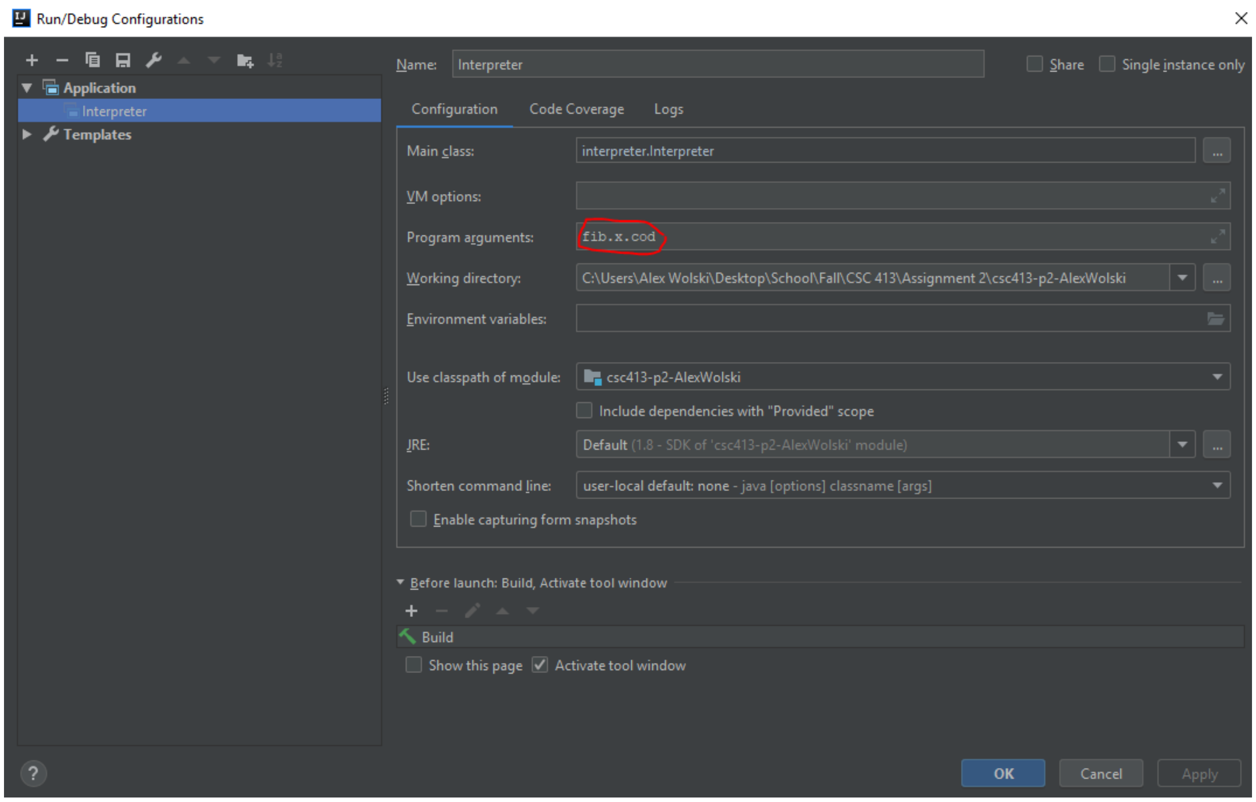
4 How to Run your Project

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.



Name: Interpreter
[ ] Share [ ] Single instance only

Main class: interpreter.Interpreter
VM options:
Program arguments: fib.x.cod
Working directory: C:\Users\Alex Wolski\Desktop\School\Fall\CSC 413\Assignment 2\csc413-p2-AlexWolski
Environment variables:
Use classpath of module: csc413-p2-AlexWolski
[ ] Include dependencies with "Provided" scope
JRE: Default (1.8 - SDK of 'csc413-p2-AlexWolski' module)
Shorten command line: user-local default: none - java [options] classname [args]
[ ] Enable capturing form snapshots
[ ] Show this page [✓] Activate tool window

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

ByteCode:
Each byte code has a subclass that inherits from **ByteCode**. This is an abstract class populated with empty methods that the subclasses can default to. However, there is an additional **JumpCode** class that inherits from **ByteCode**. This is designed to categorize the **ByteCode** objects that need to jump to other places in code (CallCode, GotoCode, and falseBranchCode.) These three byte code classes inherit from **JumpCode** so that the **Program** class has an easier time identifying them.

BopCode:
The BopCode class relies on numerous **Operator** classes to perform its operations. Each **BopCode** object contains a specific **Operator** object that computes the operation needed by that **BopCode**. This **Operator** object is created in the initialization phase. The symbol for the operator is passed as an argument and is looked up in the **OperatorTable** class. This class contains a HashTable relating symbols to their respective **Operator** object.
Each operator subclass inherits from an abstract **Operator** class in a similar fashion as each byte code subclass inherits from **ByteCode**.

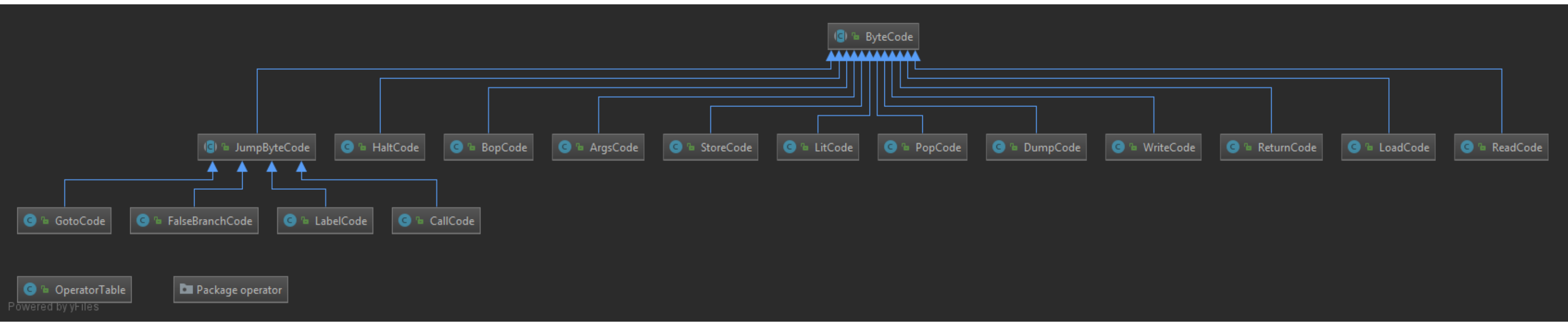
ByteCodeLoader:
To parse the arguments in **ByteCodeLoader**, I use both a BufferedReader and the *String.split()* method. Using the BufferedReader, I read the file line by line. Then I cut up the string on every space using the *.split()* method and store it in an array. The first element is the **ByteCode** itself. I pass this to the **CodeTable**, which return the name of the Java Class associated with that **ByteCode**. With this, I use Java Reflections to create a new **ByteCode** object, and pass it the remainder of the arguments to initialize the object.

Program:
In order to resolve the addresses of the **ByteCodes**, I create a HashMap while adding items to the array. Every time a **ByteCode** is added, it is checked if it is also of type **LabelCode**. If it is, then its

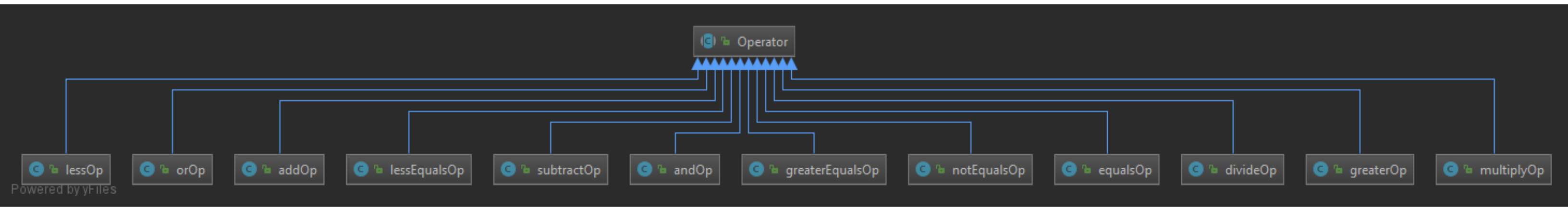
label is accesses and used a the key, while its index in the ArrayList is used as the value. So once all of the **ByteCodes** are added and the *resolveAddresses* method is called, the HashTable is already created. Then it is only a matter of searching the label name in the HashTable to get the target address for any CallCode, GotoCode, or falseBranchCode objects.

VirtualMachine & RunTimeStack:
These methods were quite simple. The **RunTimeStack** did the hard work in maintaining the memory of the program. **VirtualMachine** simply executed the bytecodes and forwarded requests for memory augmentation on to the **RunTimeStack** class.

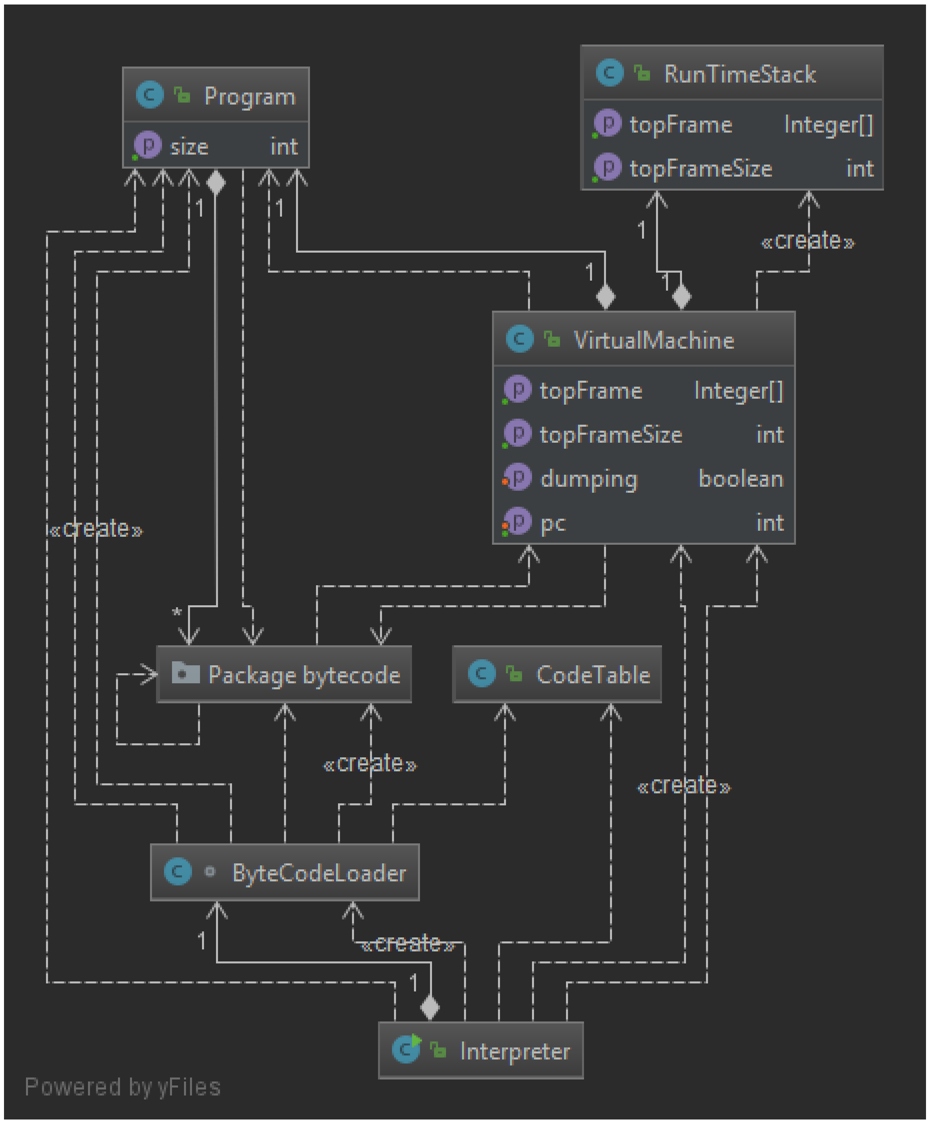
6.1 Class Diagram



This diagram shows the relationship between the **ByteCode** abstract class and its subclasses. All of the regular byte code classes and the **JumpByteCoe** class inherits from **ByteCode**. And the four classes involved in jumping (GotoCode, falseBranchCode, LabelCode, and CallCode) inherit from the **JumpByteCode** class. **OperatorTable** and the operator package are in the same package but are not related to this hierarchy.



This diagram shows each of the operator subclasses inheriting from the **Operator** parent class. Each of these subclasses are identical in general function, so there was no need for an organizational abstract class like **JumpCode**.



This diagram shows the ownership and relationships in the program. The program enters from **Interpreter**, so it makes sense that it directly or indirectly creates all of the classes in the program. For example, the **Interpreter** creates the **VirtualMachine**, which then creates the **RunTimeStack**. This diagram also depicts which classes call on the methods of the other classes. For example, we can see that the **ByteCode** package and the **VirtualMachine** class both call on each other.

7 Project Reflection

This program was a challenge. I had a particularly hard time debugging it because it required stepping through the emulated program to find the result. At times, I even had to step through the .cod files by and write down the expected behavior. However, I enjoyed overcoming these hurdles. I am satisfied with the solutions I implemented and the additional features I added. Most notable is the error checking. The program can give informative errors on syntax errors and even runtime errors. Although this was not required, I think it makes the program much more usable and professional.

8 Project Conclusion/Results

This project was a great learning experience for me. I now understand how virtual machines work and saw a glimpse of the process behind converting a high-level programming language into machine language. I had fun solving the problems I encountered, and I am particularly happy with the error checking. This project was a success. The end result was an efficient virtual machine that can run complex programs given simple instructions.