# *CSC 413 Project Documentation*

# *Summer 2021*


***Vicente Pericone***

***918693144***

***CSC413.01***


***https://github.com/csc413-su21/csc413-p2-ViP-Cente***

# Table of Contents

# 1 Introduction

## 1.1 Project Overview

So, this project is an interpreter for the mock coding language X. This interpreter project has a virtual machine where it will execute functions written on files that are in the mock language X.

## 1.2 Technical Overview

So we have the classes runTimeStack, VirtualMachine, Interpreter, ByteCodeLoader, and all the ByteCode classes. The runTimeStack is essentially the core of this project. The runtimestack class has an ArrayList that acts as the runtimestack for the mock language we'll be computing. Then there is a stack that stores pointers to different frames inside the runtimestack. These frames act as boundaries for methods that we'll call using the mock language. This runtimestack class also has a dump() function that displays the current state of the runtimestack. There are a lot of dump() functions within this project but each of them prints something different based on who is calling that function.

The VirtualMachine class acts as a controller that interacts with the runtimestack. The VirtualMachine has a variety of functions that the ByteCode classes use to do their designed functions. The VirtualMachine will take a ByteCode and call the execute() method to do its function. The VirtualMachine will also run the Bytecodes dump() function if the dumpFlag is "ON".

There is a abstract class called ByteCode that has the functions init(), dump(), execute(). The init function for each class will initialize the data fields for the ByteCode class depending on the type of ByteCode. Dump() will print something based on the ByteCode. Execute() will do a function based on the ByteCode.

The Program class is essentially the entire program the user imports using the mock language. This class has an ArrayList of ByteCodes that is the program we'll be interpreting. There are functions addByteCode() and getByteCode() that adds bytecodes and gets the bytecode at the current counter for the program respectfully. There is also a function resolveAddress() that matches all the Call, FalseBranch, Goto to their respective LabelCode address before the program is actually run. This class work with the ByteCodeLoader which is responsible for reading a .cod files and adding the ByteCodes to the Program ArrayList. It uses the CodeTable class to match the ByteCode names to their respective class names.

## 1.3 Summary of Work Completed

I worked on the VirtualMachine, RunTimeStack, Program, ByteCodeLoader, and all the ByteCode classes. I also added and AddressLabel interface for the Bytecodes that jump. I also wanted to add a BinaryOperation interface for the BOP ByteCode so that it's a little cleaner but I ran out of time to implement it.

# 2 Development Environment

I am on Java 14 and using IntelliJ.

# 3 How to Build/Import your Project

To import this project, import the csc413-p2-ViP-Cente.

# 4 How to Run your Project

To run this project, you will need .x.cod files. You the need to edit you configurations so that it runs your .cod file when running the main() in Interpreter.

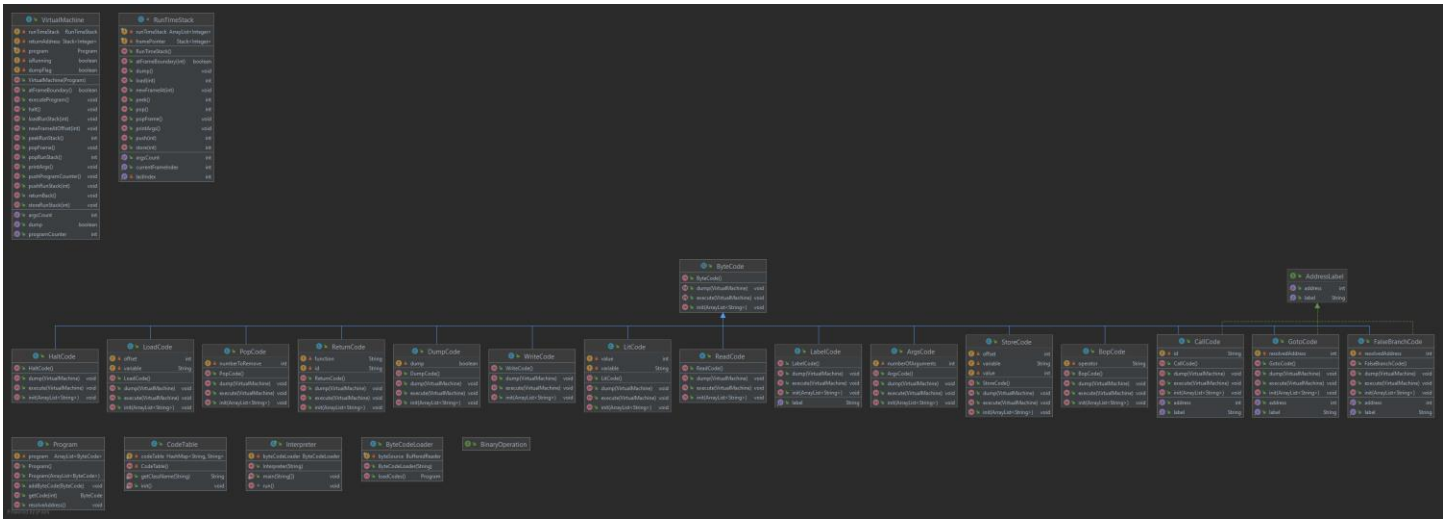
# 5 Assumption Made

For this project, I assumed that all .x.cod files are all correct.

# 6 Implementation Discussion

So design wise, the VirtualMachine interacts with runtimestack based on the execute() function that it is calling at the current ProgramCounter. This programCounter is a pointer to where the current part of the program(Mock Language X) is at. The ByteCodes extends to the abstract class ByteCode that each have their own execute functions that the VirtualMachine will use. The CallCode, GotoCode, FalseBranchCode also implements the interface AddressLabel which is used find where they will jump to before the program is ran. There is a BinaryOperation interface that I wanted to use for BopCode to make that class a little cleaner looking but I ran out of time for now.

## 6.1 Class Diagram



# 7 Project Reflection

This project really taught me a lot about designing a program in an efficient way. For example, using the Interface AddressLabel saved me a lot of time because I would've have to use conditional statements to see if a ByteCode was a Call, Goto, FalseBranch ByteCode. It also taught me how important encapsulation and to make sure that each class was doing their own job and not doing the job that other classes should be doing. Using encapsulation helped me find bugs in my code much faster because all I had to do was go to the class that was causing an exception and find the area within that class. I also saw the importance of breakpoints because my program wasn't running and it wasn't until I used breakpoints that I found the errors in the program.  I used the breakpoints to see if the ByteCodeLoader

class was creating the program correctly and I found out it wasn't when looking at all the elements at the pause.

# 8 Project Conclusion/Results

This program correctly imports and executes the program given by the user. All the Bytecodes initialize and execute in the correct way. There might be some formatting issues when dumping because I was a little confused on the exact output we are going for. There are some parts I can improve on this program to make it more efficient, specifically for the BopCode class where I used cases for the operation where I could use a hashmap or interface to call the correct binary operation.