**CSC 413 Project Documentation**

**Summer 2019**

**Kevin Venegas-Mendoza**

**915142571**

**Section.02**

**https://github.com/csc413-02-summer2019/csc413-p2-Kevinv234.git**

Table of Contents

[1 Introduction 3](#_Toc522827688)

[1.1 Project Overview 3](#_Toc522827689)

[1.2 Technical Overview 3](#_Toc522827690)

[1.3 Summary of Work Completed 3](#_Toc522827691)

[2 Development Environment 3](#_Toc522827692)

[3 How to Build/Import your Project 3](#_Toc522827693)

[4 How to Run your Project 3](#_Toc522827694)

[5 Assumption Made 3](#_Toc522827695)

[6 Implementation Discussion 3](#_Toc522827696)

[6.1 Class Diagram 3](#_Toc522827697)

[7 Project Reflection 3](#_Toc522827698)

[8 Project Conclusion/Results 3](#_Toc522827699)

*1.Introduction*

For this project we are going to be creating a language interpreter for a language called X. The interpreter is going to have a runtime stack and a frame pointer, it is supposed to imitate the way a compiler works, and this is going to allow us to have a deeper understanding on how a computer reads raw code. The compiler is going to work off of a set of instructions, once it reads the code and it is inserted into the runtime stack, byte code comes into play in order for the code to be executed.

The instructions of the interpreter are going to be within the bytecode folder. These bytecodes are going to consist of 15 different instructions, and they all do something to code that is being read by the runtime stack. A good analogy for the bytecode is almost like the LC3 Architecture for Assembly systems, where there is a limited amount of instructions. All of these instructions contain a label so you can see when they are specifically called to understand what is happening during runtime. The main program, the interpreter, is going to run off of 4 main files. These main files are program, runtime stack, virtual machine, and Bytecode loader. The Code table contains a HashMap of all possible bytecodes that are being read.

*2. Development Environment*

The IDE of choice used for this project is intellij ide. Where the main program that we are going to be running is interpreter.interpreter. This is where the main program starts and stems over to the program file and the bytecode loader.

3. *Building and Running*

In order for you to import this specific project, you are going to want to import the working directory, which is interpreter.interpreter. Once the program is imported you can work from this directory where you see the main files and the bytecode classes that are going to be added within the bytecode folder. It is important to note that you want your bytecode classes to contain the same names as those given within the Code table HashMap, since the bytecode loader is going to be scanning through the map as tokens and comparing it.

*4. Compiling*

Running this project is a bit tricky since the professor gave us multiple files in order for us to test our code. We have files in the main directory that have the .cod extension. These files are the files that contain specific instructions only. The files ending in .x are going to be files containing raw code in which you can test your program as well. If you want to target a specific program where you want to run your code, you can put the name of any of these files within the programming arguments.

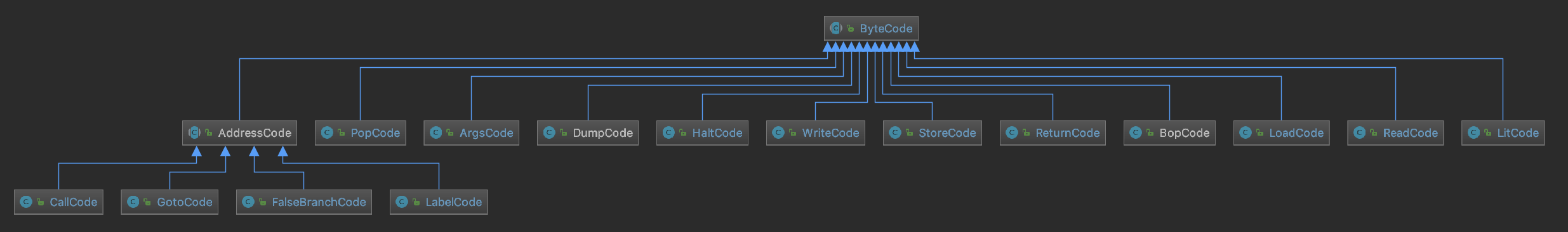
*5. Assumptions*

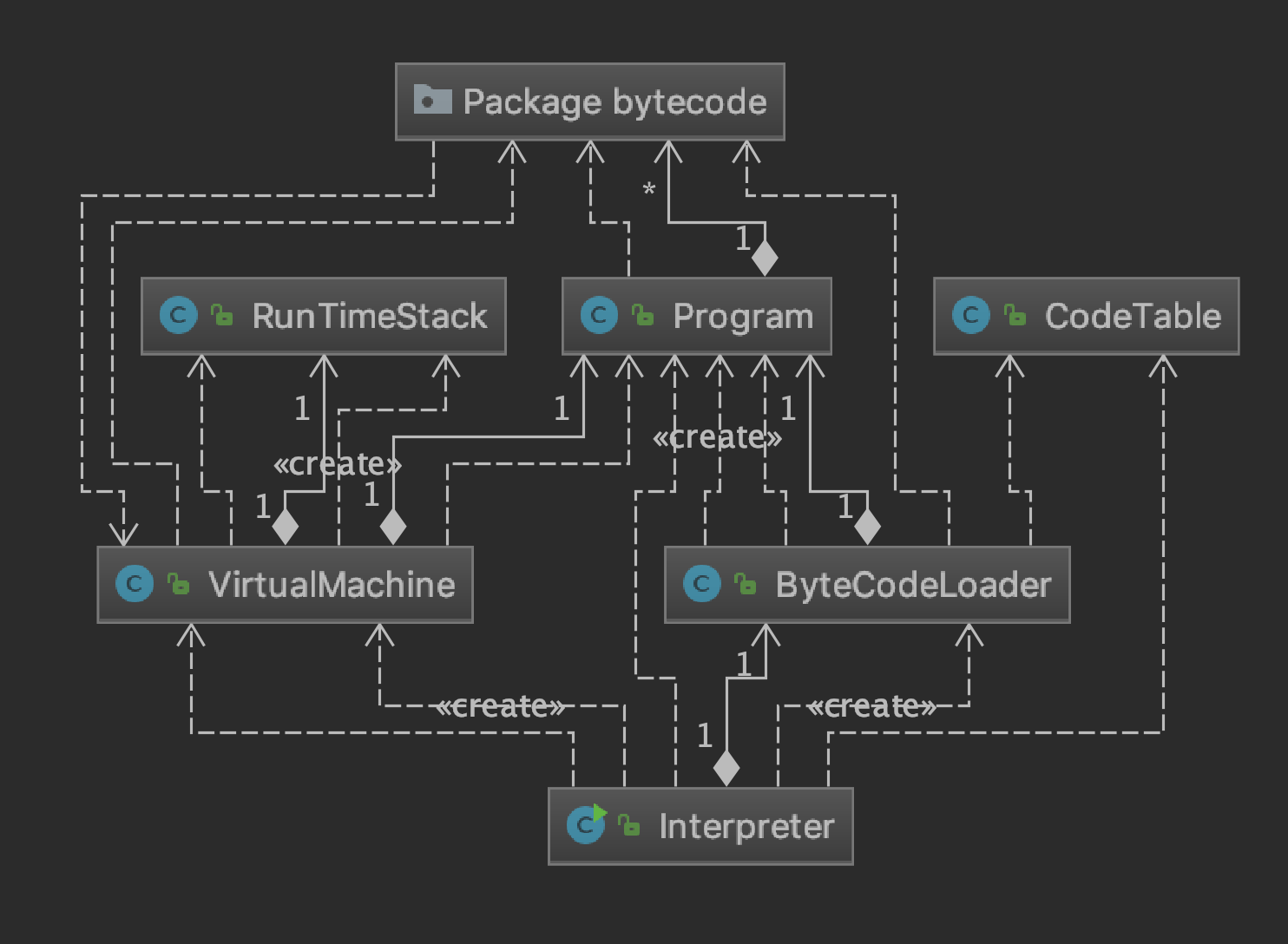
Starting the project, I made the assumption that the Interpreter file and CodeTable were correct since we are specifically not allowed to touch those files. I also made the assumption that if one file was able to be compile, the rest of the files should work as well, So I tested my program on two of the files only.

*6. Implementation*

Before I start discussing the interdependencies of the files and classes I am going to provide UML maps below so it can be easier to picture. The way I started my implementation was working on the runtime stack and the virtual machine. The reason behind this is, if I understand the way these two files work, I should be able to understand the anatomy of a compiler, in this case the interpreter for language X. Once I understood these two files, I was able to understand the direction of the code, the runtime stack, the purpose of address, and the purpose of the frame pointers. The runtime stack is going to contain the current code and instructions until a instruction is performed. Once we have a value, we pop it and keep it on top of the stack. The frame pointer is going to contain sections that separate the runtime stack per function being ran. With the frame pointer, we can jump to specific parts of the runtime for whenever we call GOTO. Once I implemented these files, I started working on all the bytecode classes and implemented them solely off of the directions given within the PDF documentation for the interpreter. The final function I implemented was the loadsCode function in the ByteCode loader.

Discression\* My program breaks in loads code, the function catches the ClassNotFound Exception. I tried debugging my program for days and could not figure out what was going on. At first, I verified the names of the ByteCode, and that was correct. I traced the stack and I was getting Null in return when my program was tokenizing the classes. I am going to partner with the professor to understand where I went wrong.

**

**

ß