CSC 413 Project Documentation

Fall 2018

Jonathon Knack

917629133

413.01

https://github.com/csc413-01-summer2019/csc413-p2-jknack0.git

Table of Contents

1 Introduction

1.1 Project Overview

The interpreter program reads a file containing a program written in the programming language X and executes those instructions. Think of the program written in X as a video game and the interpreter is the console that plays the game.

1.2 Technical Overview

The interpreter reads byte codes from a file, creates an instance of that byte code instruction then executes the instruction. All byte codes are an abstraction of the ByteCode class that has an init method and an execute method. The init function in each byte code handles the initialization of any variables in the byte code and the execute method in each byte code is how it interacts with the runtime stack.

The ByteCodeLoader class is responsible for reading the byte code file and imports them into the program class. The ByteCodeLoader loops through the lines of the file until it reaches the end of the file. For each line, creates a stringTokenizer that breaks the line into separate strings that are the byte code name and possibly it's parameters. Then using reflection we create an instance of the byte code class and use the classes init method to initialize any variables that the code may have. Once we have an instance of a byte code we add it to the program.

The program class contains an array list of ByteCodes that is our program and a method called resolveAddrs. The resolveAddrs method loops through the program array list and stores the labels name and the position of the label in the program in a hash map where the key is the label name and the value is the position in the program. Then the method loops through the program looking for instances of GotoCode, FalseBranchCode and CallCode. The method then uses the symbolic address of the code and initializes the resolved address with the position in the program.

The RunTimeStack class contains an ArrayList called runTimeStack, a Stack called frame pointer stack, a dump method and various methods that byte codes use to perform operations on the both of these data structures. The runTimeStack contains the values of the variables for each method call in the program, the frame pointer stack is used to create a boundary between each of these of these method calls. The dump method is used to output the values of the runtime stack separated by the frames in the framePointersStack.

The VirtualMachine class contains instances of RunTimeStack, Program and a stack the contains the return addresses for all of the method calls in the program. It also contains a method called executeProgram that loops through the program and executes each byte code in the program in order.

1.3 Summary of Work Completed

I implemented all of the byte code abstractions from ByteCode and all of the init and execute methods for each byte code. I implemented the ByteCodeLoader class and it's method loadCodes(). I implemented the Program class and it's resolveAddrs() method. I implemented the RunTimeStack class and all the required methods except dump doesn't print out the correct output. I implemented the VirtualMachine class and the execute() method.

2 Development Environment

The IDE used was the latest version of IntelliJ using java version “12.0.1”.

3 How to Build/Import your Project

To import the project just import using existing files and make csc413-p2-jknack0 as your root.

4 How to Run your Project

To run the project run the interpreter class and use the x.cod file as the arguments.

5 Assumption Made

The assumption made in this project is that all the x.cod files are valid programs.

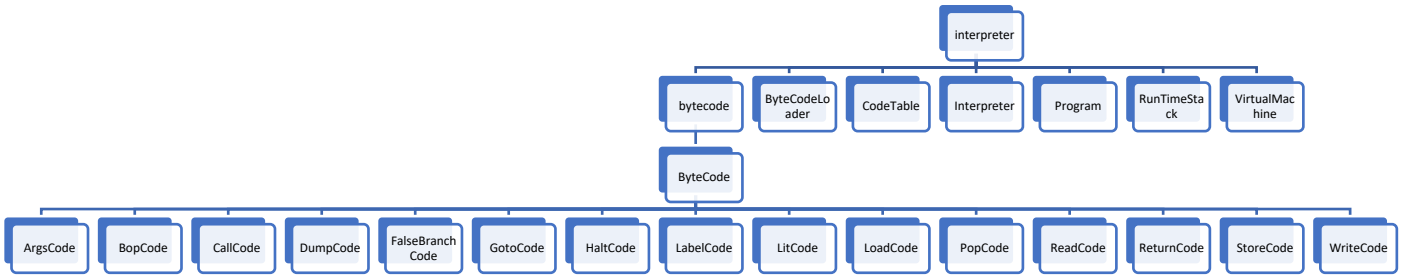
6 Implementation Discussion

In order to not break encapsulation public methods were added to the VirtualMachine class and the Virtual machine was a parameter passed to all execute functions so that each byte code could interact with it. These methods in VirtualMachine also were used to call RunTimeStack methods so that ByteCodes execute in both places without breaking encapsulation.

I chose to use a StringTokenizer in the loadCoades() method instead of the String.split() method just incase there were random spaces in the x.cod file that would have caused problems making certain arguments be stored in different places of the array.

In the resolveAddrs() method in the program class I couldn’t figure out how to do this in one pass but liked using a hashmap to store the string value of the label as the key and the value of the key the position of the label.

6.1 Class Diagram



7 Project Reflection

I really liked this project and I'm glad I started early! I was going home for the 4th of July so I wanted to get it done by that Thursday but I started so early I finished the weekend before. At first I was really scared because of how big the project was and that PDF was a nightmare but as I started working on it the more you started seeing the big picture and how things worked together. I also really liked using reflection to create the instance of an object at runtime which I had only read about until now.

8 Project Conclusion/Results

I successfully implemented all of the byte codes by abstracting ByteCode and completing all of their init() and execute() methods. I implemented the ByteCodeLoader class to read from a x.cod file and create instances of the byte codes using reflection. I impelemented the Program class and it's resolveAddrs() method. I successfully implemented the runtime stack and all of the required methods. I implemented the VirtualMachine class and it's execute method and all other methods needed for the byte codes to interact with the VirtualMachine and not break encapsulation. The only thing that doesn't work is the dump which kind of works but I just couldn't figure out how to parse the strings correctly.