SOFTWARE PROJECT - INTERPRETER

I. Title Page

Assignment Title: The Interpreter

Name: Jianfei Zhao

Link to the repository: https://github.com/jzhao11/interpreter

II. Introduction

a. Project Overview

This project is mainly about implementing an interpreter for the mock programming language X, similar to but simpler than Java. The interpreter aims to cope with the given bytecode and compute corresponding results. The completion of project will familiarize us with the principles of OOP design.

b. Technical Overview

From the technical perspective, much endeavor is made on the design of bytecode classes and the implementation of functional classes for interpreter. Compared with assignment 1, this project requires more logical analysis, since the interpretation process involves collaboration of multiple functions. With the utilization of OOP principles, the implementation could be divided into modular classes. Specifically, encapsulation rules could not only decouple the complicated relations between modules, but reduce code redundancy. The abstraction and inheritance ensures the flexibility of bytecode classes.

c. Summary of Work Completed

The bytecodes are read from file and parsed by interpreter’s code loader to construct ByteCode objects, during which the labels are resolved into program counter (pc) addresses. The interpreter then launches its virtual machine to run the loaded program, with RunTimeStack and concrete ByteCode classes maintaining implementation details and VirtualMachine responsible for execution. Technically, VirtualMachine works as a bridge that links other functional classes together while keeping them apart (encapsulated).

Based on the interpretation process, bytecode classes are created to provide method interfaces and modified by following OOP inheritance. To be more precise, the ByteCode is treated as an abstract superclass, and all other bytecodes derive from it as concrete ones. Among the subclasses, PcJumpCode also serves as a superclass, which is extended by CallCode, FalseBranchCode, and GotoCode. Detailed discussion for class design is available in section **VII**.

When working on the above development, the project is debugged and rebuilt back and forth, with the running outcomes compared with theoretical values of factorial and Fibonacci sequences. During bytecode tests, the dumping method for run-time stack is also implemented. It could be regarded as a powerful tool for error detection.

III. Development Environment

a. Version of Java Used

The version of Java used for development is 1.8.0_101.

b. Integrated Development Environment (IDE) Used

The IDE for Java is NetBeans 8.2 on Windows System.

IV. How to Build or Import the Project in the IDE

i). Clone the repository from the link in title page (page 1), by using the command: git clone https://github.com/jzhao11/interpreter.

ii). Open the NetBeans IDE. Import the Project by clicking “File->New Project”.

iii). Choose “Java Project with Existing Sources” in the “Projects” choice box and click “Next”. Specify a name (e.g. csc413_hw2_jzhao11) and location for the new project and click “Next”.

iv). Click “Add Folder” next to the “Source Package Folder” choice box. In the pop-up window, find the local “csc413-p2-jzhao11” repo, and choose “csc413-p2-jzhao11->interpreter”. Click “Open” and the file path will appear in the upper box. Then click “Next”.

v). All the *.java files in the “Included Files” box are needed. Click “Finish” to finish importing the project.

vi). To build the project, right click the imported project in the top-left “Project” box, and choose “Clean and Build”. This can also be done by clicking the “Clean and Build Project” button (Shift+F11) on the top navigation bar.

V. How to Run the Project

i). Before running, a bytecode file has to be set as the argument of main function. In the top-left “Project” box, right click the project and choose “Set

Configuration->Customize". In the pop-up, choose "Run" from "Categories" and set the "Working Directory" as the path of local repo, for example, "C:\Users\......\csc413-p2-jzhao11". Fill in the "Arguments" box with the file name used for testing, e.g., "factorial.x.cod", "fib.x.cod", or "fibA3.x.cod".

ii). To check the correctness of this interpreter, the given bytecode file has to be logically and syntactically correct. Choose the imported project and click the "Run Project" button (Ctrl+F11) on the top navigation bar.

VI. Assumptions in Design and Implementation

In this project, the interpreter only considers some basic instructions of bytecode. Therefore, the following bytecodes are permitted in X-machine: "HALT", "POP", "FALSEBRANCH", "GOTO", "STORE", "LOAD", "LIT", "ARGS", "CALL", "RETURN", "BOP", "READ", "WRITE", "LABEL", and "DUMP". Besides, the main method requires the input file name as argument. This project will interpret and execute the bytecode instructions included in this file, among which labels should look like "id<<num>>" and bytecodes should be bug-free both in logic and in syntax for the interpreter to achieve right results.

Additionally, the interpreter only accepts integers as valid operands for computation. This means that all decimal numbers together with non-numeric types of operands are excluded from the bytecodes.

VII. Implementation Discussion

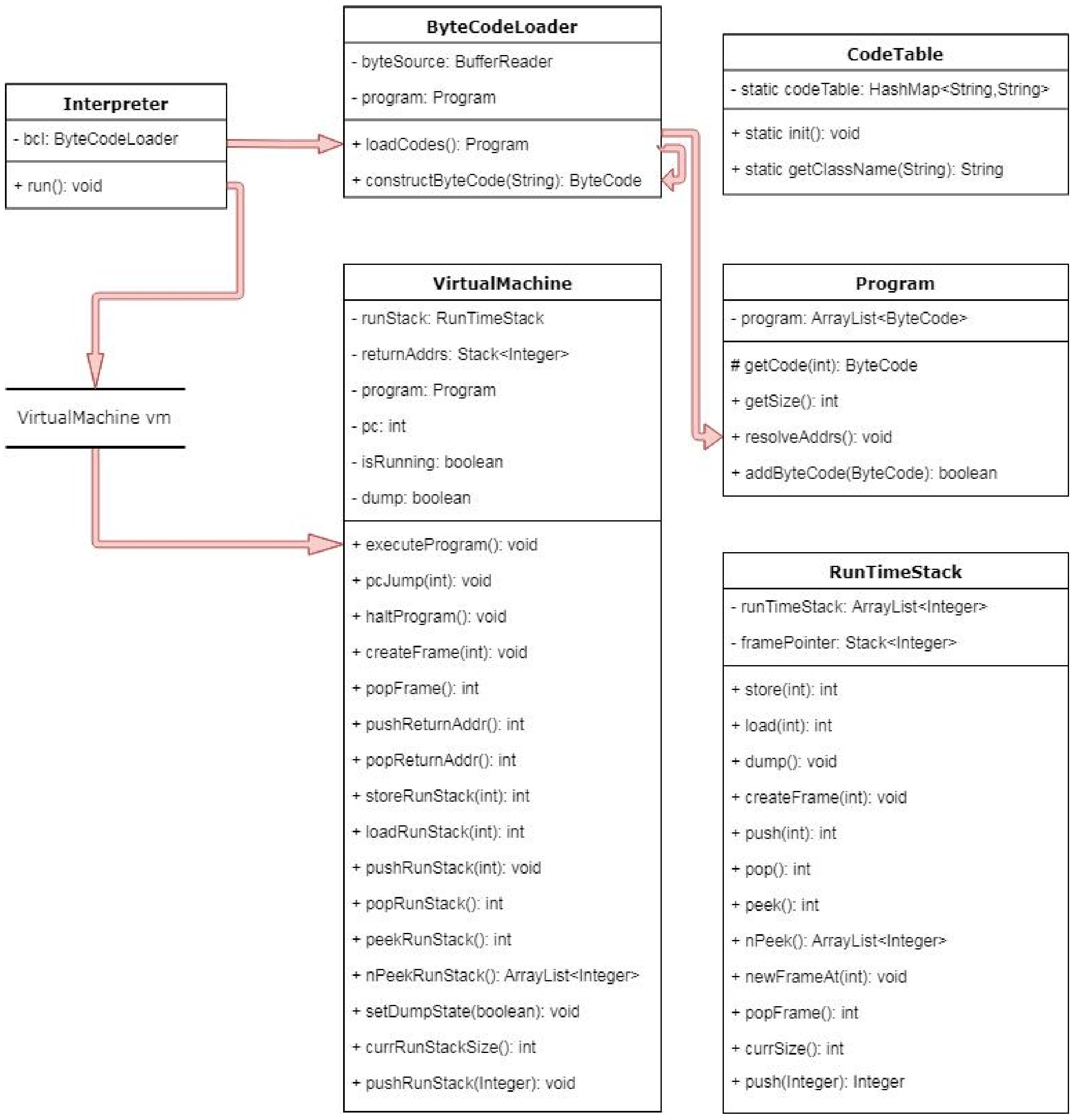

Figure 1. Implementation of functional classes of the interpreter

Figure 1 lists the functional classes in this interpreter. Each member field is written as “name: type” while each function is in the form of “name(parameter list): return type”, where “+”, “-”, “#” denotes the keyword “public”, “private”, and “protected”, respectively. The interpreter’s working process (highlighted as red arrows in Figure 1) can be summarized as: **i)** loading code; **ii)** constructing bytecode objects; **iii)** resolving pc addresses; **iv)** carrying out the interpreted program by virtual machine.

The development begins with the ByteCodeLoader, which contains bytecode loader and constructor. Basically, the loader will read input file and tokenize each line by using white space as delimiter. With the “bytecode - class name” pairs stored in a static hash map in CodeTable, concrete bytecode objects can be easily constructed with those parsed token strings. Afterwards, all the labels are translated into pc addresses through the resolveAddrs() method in class Program. At last, an instance of VirtualMachine will run the program. Furthermore, RunTimeStack provides dump() method to check whether the run-time stack is exactly as expected, especially useful when encountering an error.

To avoid producing much duplicate code, class VirtualMachine does not make an executing method for each bytecode. Instead, it merely focuses on how to instruct other functional classes for specific operations, without worrying about the inner routine within each of them. From another point of view, ByteCode’s subclasses and RunTimeStack can manipulate their own logic, according to what is requested by VirtualMachine. The bytecode classes do not have any direct access to the run-time stack, ensuring that the encapsulation will not be broken. In this way, the design pattern partitions the modularity clearly and enhances the compactness and readability of code.

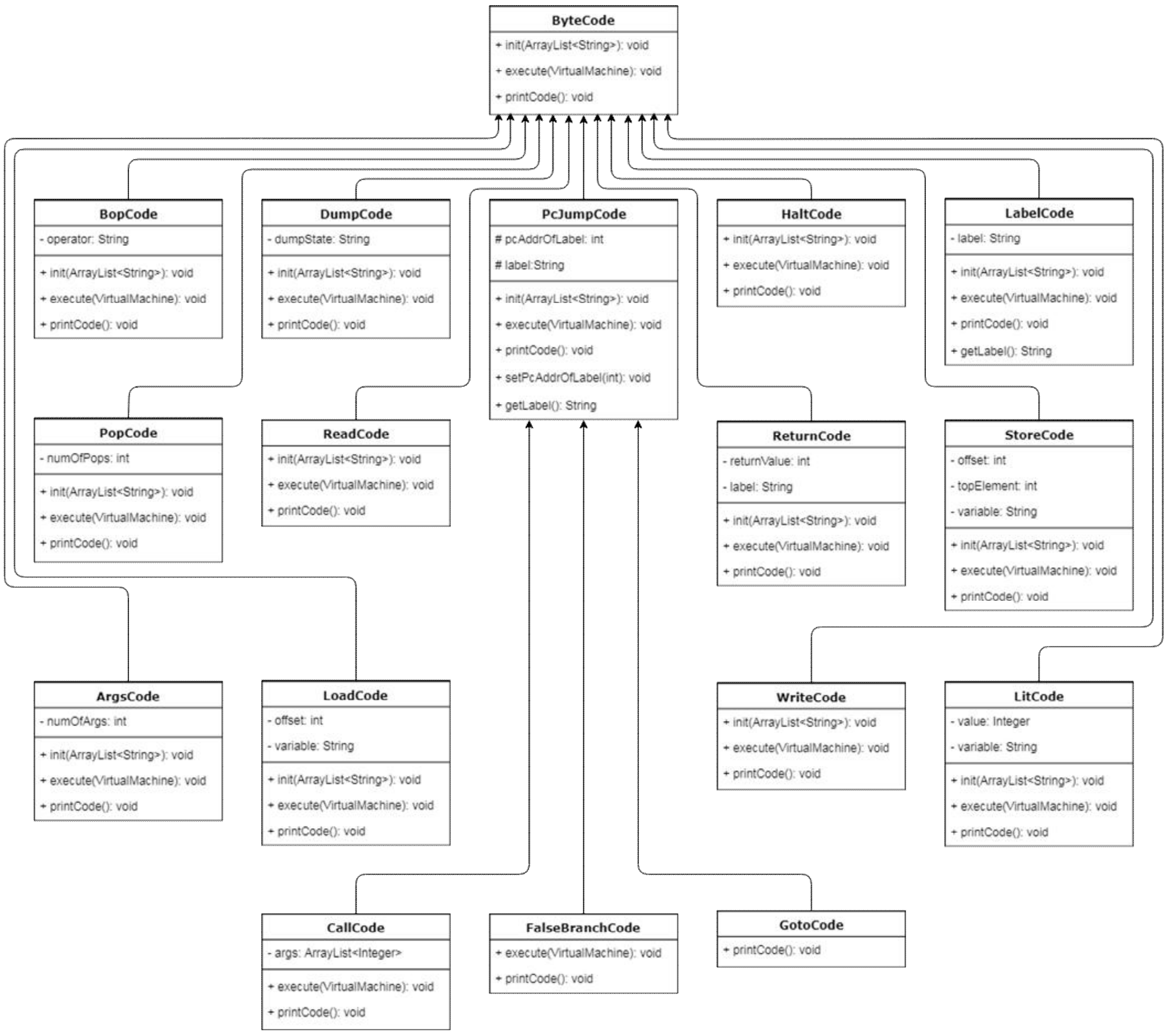

Figure 2. Hierarchy and inheritance of bytecode classes

As drawn in Figure 2 (could be zoomed in if necessary), the graph depicts the hierarchy of bytecode classes, with each arrow referring to the inheritance from a subclass to a superclass. ByteCode is defined as an abstract class, since it extracts the common functions that all of its descendants should have, such as initializing the member fields by using strings parsed from the input file, arranging their own logic of execution, as well as printing bytecode information to check whether it has been correctly constructed.

When implementing the interpretation method, the ByteCode reference needs to be instantiated as an object of each concrete subclass listed in Figure 2. Due to the different parameters they may request, member fields and methods are added individually in each derived bytecode classes. Apart from ByteCode, its subclass PcJumpCode is the special one, since it seems to declare a “pseudo” bytecode. The reason for such design can be explained as follows. Unique as they are, the “CALL”, “FALSEBRANCH”, and “GOTO” share the same feature: they all trigger the jumping of pc. Hence, PcJumpCode is extracted as another layer between ByteCode and them, providing a general version of init(), as well as fields and method interfaces related to retrieving label and setting pc address. Call-, FalseBranch-, and GotoCode can extend PcJumpCode and override execute() or printCode() as needed, without the re-definition of other things.

VIII. Project Reflection

Having passed the recursive factorial and Fibonacci program tests and self-defined simple cases, this project satisfies the requirements listed in the assignment document. However, there is still drawback within this interpreter. For example, the method for resolving pc addresses is a double loop, whose time complexity can be optimized by using a hash map, with label names as the keys. And the run-time stack can only accept integers as operands for computation.

In spite of the defects it may have, this project succeeds to interpret bytecodes under the assumptions mentioned in section **VI**. By obeying OOP design principles, the modules are clearly partitioned into functional classes (VirtualMachine, RunTimeStack, ByteCodeLoader, etc.). The inheritance hierarchy of bytecode classes makes it extensible for involving other types of bytecode. And the encapsulation rules help me remove a lot of repetitive code.

IX. Project Conclusion and Results

The “Interpreter.java” has the main function, serving as the entry point of the project. Before checking the interpretation method, the name of the testing bytecode file needs to be passed as an argument.

After successfully building, users can run this project and the result of bytecode will appear in the Netbeans console. As long as the logic and syntax in the given file is valid (like factorial-, fib-, fibA3.x.cod), the interpreter will generate correct output. Factorial and Fibonacci are good examples to verify an interpreter, since they contain direct and branch flow, and their recursive calls can validate the correctness of the run-time stack frame. For dumping tests, the flag can be turned on or off simply by adding “DUMP ON” or “DUMP OFF” anywhere in the file. When turned on, the interpreter can dump the bytecode that has just been executed, and also the state of run-time stack, with “[” and “]” embracing the integers (each two separated by “,”), or “[]” as an empty stack.