Assignment 3 Interpreter

CSC 413

Peter Le

https://github.com/csc413-02-sp18/csc413-p3-Peter408

Instruction to compile:

Option 1 IntelliJ:

-Open IntelliJ and select “file”, “New”, and select “Project from Existing Sources...”
-Select the folder where the source files are held named “csc413-p3-Peter408”
-In the top right, select “Edit Configurations” and put the file you wish to test in the “Program Arguments”
-Hit “Run” which will compile then run

Assumptions:

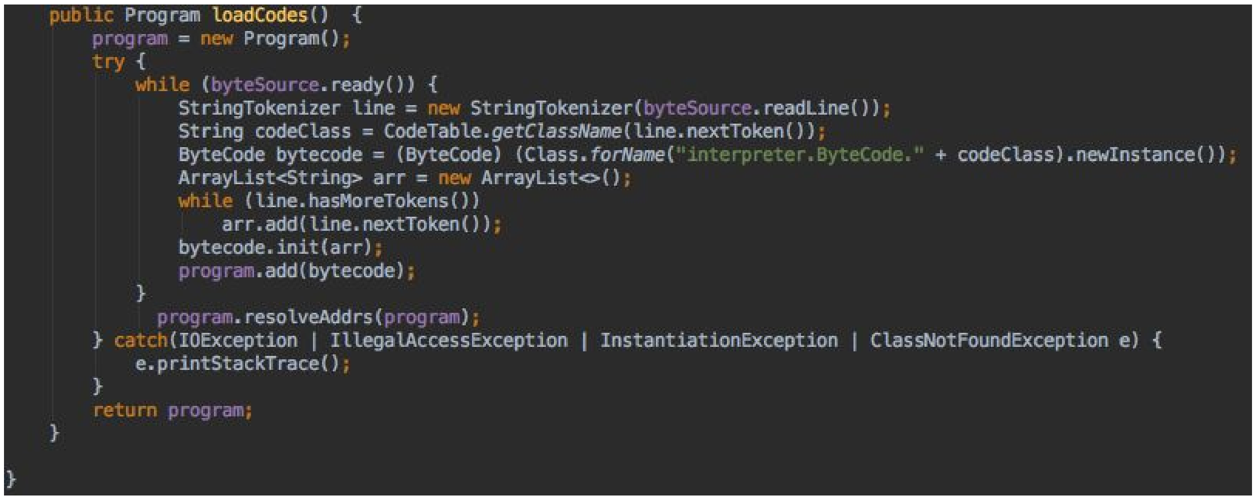
We are assuming that the .x.cod files work with all valid inputs as we did not check bounds for invalid .x.cod. The program will take in a x.cod file and ask the user to enter in a value to evaluate. The program also has a dump function to display frames for debugging purposes.

Project Introduction:

For this assignment, we were given a mock language for us to write an interpreter for. A skeleton code was given for the structure of the program, however many of the main functions still needed to be implemented. The Program starts with the “Interpreter” class where it uses the “ByteCodeLoader” to read from the file of .x.cod into ByteCodes using “CodeTables”. It then will put these ByteCodes into a “Program” which is an arrayList to hold them. From there the “VirtualMachine” is called which uses the “RunTimeStack” to manage itself and execute the mock language with the ByteCodes.

Implementations:

-LoadCodes()

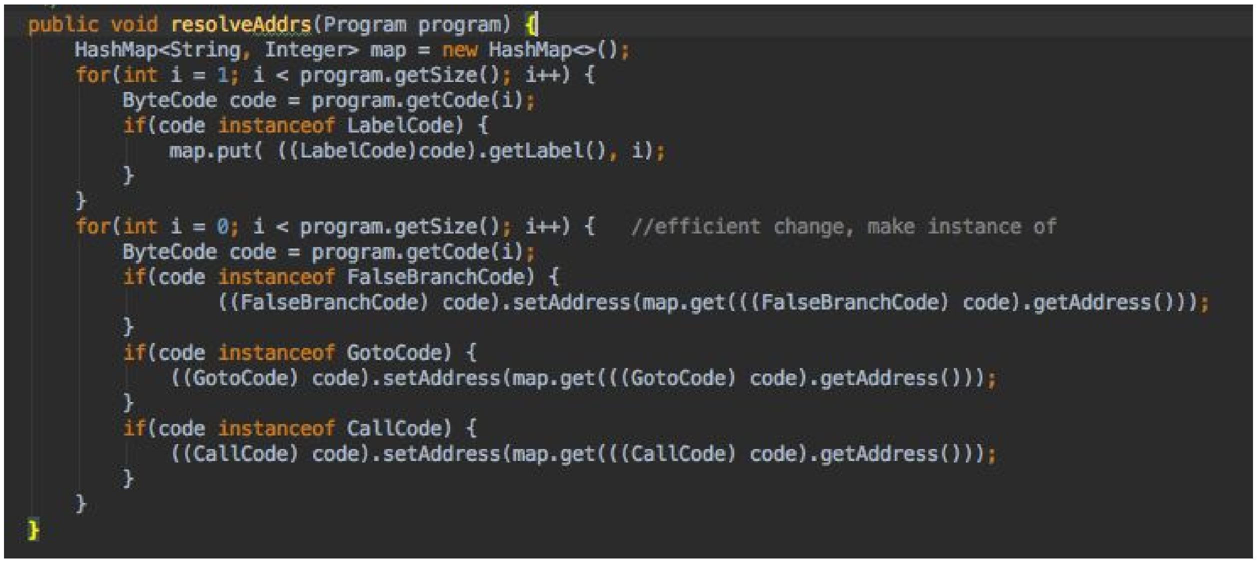


```
    public Program loadCodes()  {
        program = new Program();
        try {
            while (byteSource.ready()) {
                StringTokenizer line = new StringTokenizer(byteSource.readLine());
                String codeClass = CodeTable.getClassName(line.nextToken());
                ByteCode bytecode = (ByteCode) (Class.forName("interpreter.ByteCode." + codeClass).newInstance());
                ArrayList<String> arr = new ArrayList<>();
                while (line.hasMoreTokens())
                    arr.add(line.nextToken());
                bytecode.init(arr);
                program.add(bytecode);
            }
              program.resolveAddrs(program);
        } catch(IOException | IllegalAccessException | InstantiationException | ClassNotFoundException e) {
            e.printStackTrace();
        }
        return program;
    }

}
```

The LoadCode() function using the class variable ByteSource which holds the x.cod string taken in from the BufferedReader. It reads it line by line breaking up the line using the StringTokenizer into ByteCodes and its additional arguments into an arrayList. Each ByteCode is then created with init and added to the "program" which will hold all the ByteCodes. At the end it will call the resolveAddrs function which will replace each address / jump with the corresponding location in the program.

-resolveAddrs(Program)

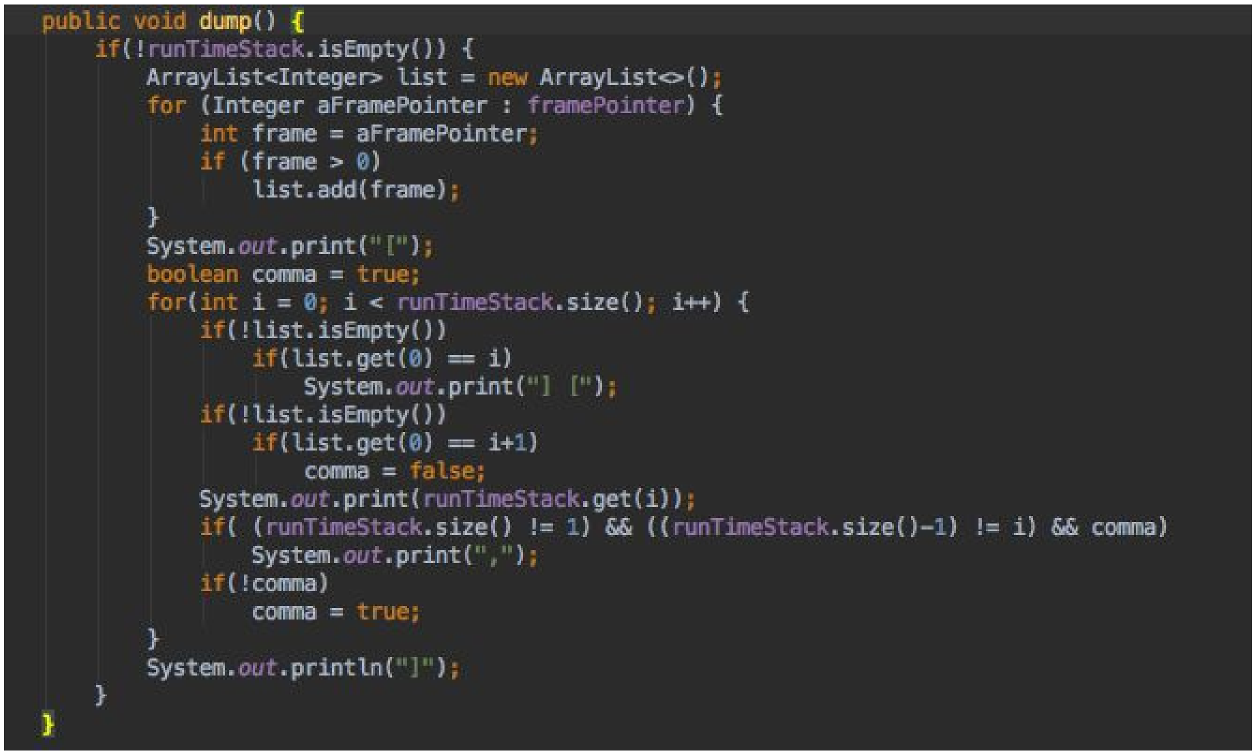


```
public void resolveAddrs(Program program) {
    HashMap<String, Integer> map = new HashMap<>();
    for(int i = 1; i < program.getSize(); i++) {
        ByteCode code = program.getCode(i);
        if(code instanceof LabelCode) {
            map.put( ((LabelCode)code).getLabel(), i);
        }
    }
    for(int i = 0; i < program.getSize(); i++) {   //efficient change, make instance of
        ByteCode code = program.getCode(i);
        if(code instanceof FalseBranchCode) {
                ((FalseBranchCode) code).setAddress(map.get(((FalseBranchCode) code).getAddress()));
        }
        if(code instanceof GotoCode) {
            ((GotoCode) code).setAddress(map.get(((GotoCode) code).getAddress()));
        }
        if(code instanceof CallCode) {
            ((CallCode) code).setAddress(map.get(((CallCode) code).getAddress()));
        }
    }
}
```

The resolveAddrs function will replace the address jumps with its corresponding location. It does this by first looking through the program and adding to a hashMap if it is an instance of

LabelCode with its corresponding line number. The reason for this is because LabelCode is the ByteCode where the program will jump to that we need to resolve. We will then do a second loop of the program that will check if it is contained within the hashMap, as this will allow us to know which line it was on. Using three if checks, we check if the found code is the three that can jump, which we will return its line number that we put in the hashMap to resolve the address.

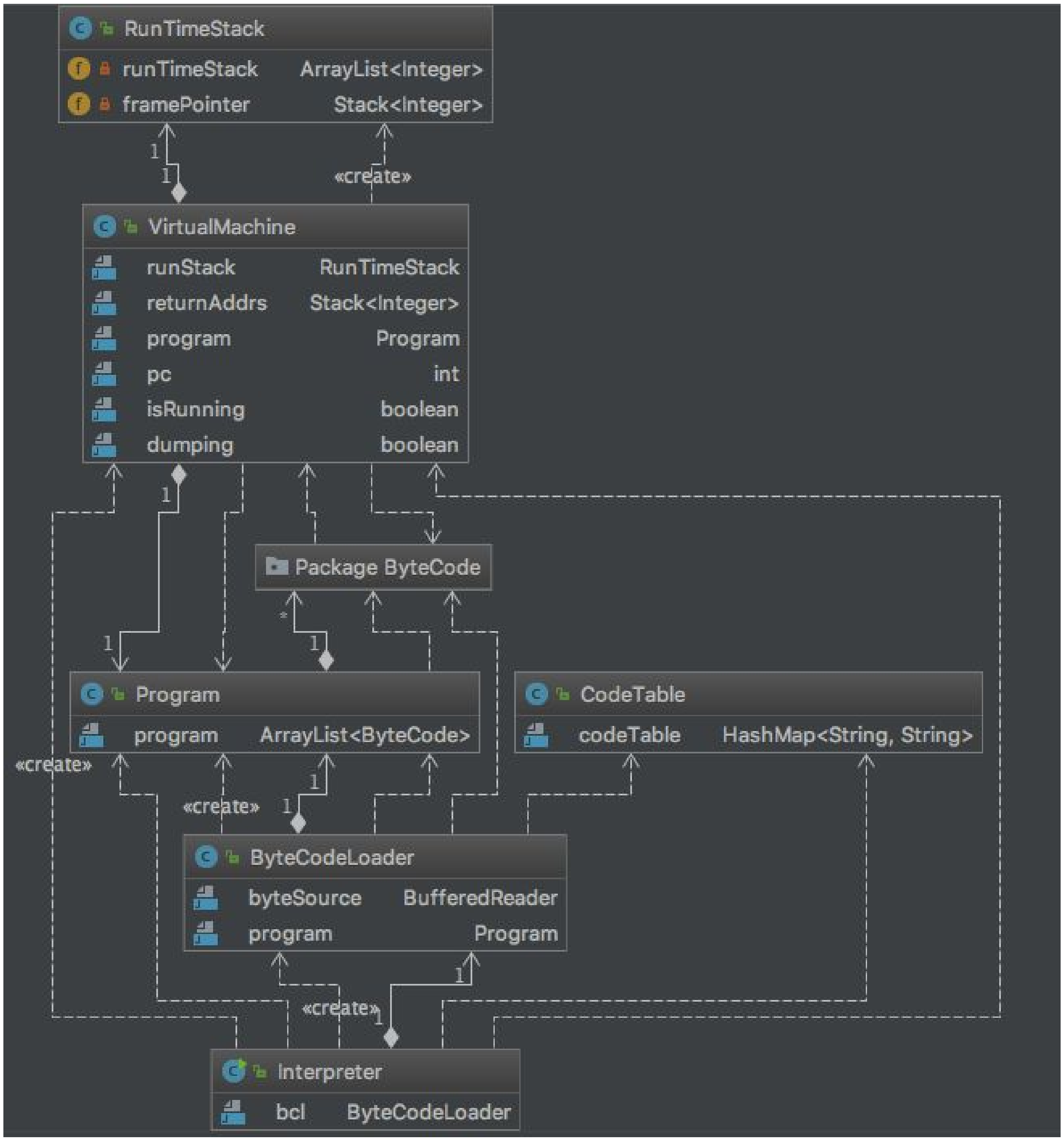
-dump()



```java
public void dump() {
    if(!runTimeStack.isEmpty()) {
        ArrayList<Integer> list = new ArrayList<>();
        for (Integer aFramePointer : framePointer) {
            int frame = aFramePointer;
            if (frame > 0)
                list.add(frame);
        }
        System.out.print("[");
        boolean comma = true;
        for(int i = 0; i < runTimeStack.size(); i++) {
            if(!list.isEmpty())
                if(list.get(0) == i)
                    System.out.print("] [");
            if(!list.isEmpty())
                if(list.get(0) == i+1)
                    comma = false;
            System.out.print(runTimeStack.get(i));
            if( (runTimeStack.size() != 1) && ((runTimeStack.size()-1) != i) && comma)
                System.out.print(",");
            if(!comma)
                comma = true;
        }
        System.out.println("]");
    }
}
```

The dump method will is used for debugging purposes as it will display the byteCode along with the stack. It first checks to see if the runTimeStack is empty, so that we know if anything needs to be done. We loop through the fromPointer and push the values to an arrayList if the frame is valid. A boolean is created to manage the display of the stack for the coma as we only want it in the middle. We run another loop of the runTimeStack size and check if the list is empty, if not empty then we will display the stack till the end.

Implementation UML:



Results and conclusion:

The program works with all valid inputs as we did not check for out of bounds or incorrect x.cod files. What I learned from this assignment how was to manage multiple files working together for a program to run. It had a lot of parts to it which made it extremely hard to manage as many of the classes uses one another to function correctly. Being able to understand how an interpreter works is interesting because shows what happens behind the scene of the program. Knowing this will help me with future works because of the way I overcame some of the challenges I faced on this assignment as mentioned above.