# Introduction

Java code is not directly converted to machine instructions. It is compiled into byte code and when running in the JVM (Java Virtual Machine) those byte codes are converted into machine instructions. Java byte code has predefined structure which allows it be decompiled back to the high-level programming language. Therefore, it is possible to perform analysis on the byte code to reconstruct the java code.

The purpose of this application is to decompile a given byte code file (.class file) and construct the method call tree for a selected method. The application will also decompile subsequent class files to which the method calls if the class file is available in the directory. It will also be able to detect and handle special instances like missing class files, recursion etc.

The application was built on top of the ClassFileParser application which was provided.

# Compiling and Running

# Functionality

The program provides the capability to read a java class file and construct a call tree for a chosen method.

The class file is passed as an argument to the program and you will be prompted to choose a method from a list of methods that are available in that class. Once a method has been chosen, the call tree for the method will be shown as a command line output.

If the method being called belongs to another class, then the program will try to parse the corresponding class file to obtain the subsequent method calls.

If the program detects recursive calls then it will terminate the call tree branch and indicate recursion with a [RECURSIVE] tag next to the call tree branch.

# Design

## High Level Design

SmAssignment2.java contains the main method for the application. The main method will get the class file directory from the command line arguments and create a ClassFile object passing in the class file directory.

The implementation in ClassFile.java will take care of creating a DataInputStream and reading the byte code into a ClassFile object. Once the object has been constructed, the method information of the class files is taken using the getMethods() method that is available in the class file. This will return an array of method\_info. Subsequently the user will be prompted to choose a method from the list of methods shown.

Once the user selects the method, the method\_info object will be passed to the DrawMethodTree() method which will recursively draw the call tree starting with this method. The DrawMethodTree() method will obtain CodeAttributes of the method if any exists and create a list of instruction objects. Instructions of the following type are filter from all instructions.

* Invokevirtual
* Invokestatic
* Invokespecial
* Invokedynamic

These instructions are represent a method call in JVM instructions. Once each Instruction object is known to be of the above types, the next two bytes can be considered as a reference to the Constant Pool MethodRef which has the information of the method being called. From the class\_index and the name\_and\_type\_index we can figure out the method name, descriptor and the class it belongs to.

If the method being called has already been called before in this call tree branch, then that indicates recursion. To identify recursion, a list of method called is maintained that is unique to each branch. When a new method is called, it checks if the method already exists in the branch or is the same as the caller. In that case we can indicate the recursion with the rag [RECURSIVE] and return so that we don’t end up in an infinite recursive call to DrawMethodTree().

If the method being called has a different class name than the current class, then find the class in the current directory and try to create a class file object using it and set it as the current class file. If it failed to find that class file, then we can ignore it by displaying the tag [MISSING].

Then try to find out if the MethodInfo exists in the current class file using the MethodRef of constant pool and if exists call DrawMethodTree() recursively. This recursion will take of drawing the call tree. The base condition of the recursion here is that there are no more invoke instructions in the last method.

Each time a method is being printed, add it to the allMethods list if it does not exist before (to avoid duplicates). Do the same to class files with allClasses. When the program finishes execution we can read the size of both the list and figure out the number of unique method/constructors and classes.

## Classes

The following files were provided to be used with the program.

* ClassFile.java - Modified
* ClassFileParser.java - Unused
* ClassFileParserException.java
* ConstantPool.java - Unmodified
* CPEntry.java – Modified
* Instruction.java – Unmodified
* Opcode.java - Unmodified

Extra classes added

* FieldInfo.java
* MethodInfo.java
* AttributeInfo.java
* CodeAttributeInfo.java
* SM Assignment2.java

### Class – ClassFile

Initial ClassFile provided had the implementation to take a class file as a parameter and creates a ClassFile object from it. But the implementation was not complete. The given file could only parse till the constant pool. To construct the call tree method information is needed. The rest of the implementation to parse the ClassFile was added later.

### Class - MethodInfo

MethodInfo class maps directly to the method\_info in the class file. The constructor accepts a DataInputStream which can be used to read the bytes and map it to a Methodnfo object.

### Class – AttributeInfo

AttributeInfo class can be used to map an attribute\_info type in the class file. Implementation of this class is like MethodInfo class. AttributeInfo has a class name parse that reads the attribute type using attribute\_name\_index. If it is a code attribute, then a CodeAttributeInfo object is created and returned. Else an AttributeInfo object is returned.

## Class – CodeAttributeInfo

CodeAttributeInfo is a type of AttributeInfo. Implementation is like AttributeInfo. Instead of the info array in AttributeInfo this class contains additional attributes. The most important of which is the code array. This array is a byte array and contains the JVM instructions which is needed to find out the method calls.

The method GetInstructionList will parse the code array and return a list of Instruction objects.

# Testing

# Quality

RSM (Resource Standard Metrics) was used to perform quality analysis on the software. The option -n was used to generate a quality analysis report.

# Referencing