Eötvös Loránd University

Faculty of Informatics

Department of Programming Languages and Compilers

Information retrieval from Java archive format

Supervisor:

Dr. Zoltán Porkoláb

Associate Professor

Author:

Bálint Kiss

Computer Science MSc

Budapest, 2017

Abstract

During the course of my work, I contributed to CodeCompass, an open source code comprehension tool made for making codebase of software projects written in C, C++ and Java more understandable through navigation and visualization. I was tasked with the development of a module for recovering code information of Java classes in JAR files. This document details background concepts required for reverse-engineering Java bytecode, creating a prototype JAR file reader and how this solution could be integrated to CodeCompass.

First, I studied the structure of JAR format and how class files are stored in it. I looked into the Java Class file structure and how bytecode contained in class gets interpreted by the Java Virtual Machine. I also looked at existing decompilers and what bytecode libraries are.

I created a proof-of-concept prototype that reads compiled classes from JAR file and extracts code information. I first showcased the use of Java Reflection API, then the use of Apache Commons Byte Code Engineering Library, a third-party bytecode library used for extracting and representing parts of Java class file as Java objects. Finally, I examined how CodeCompass works, how part of the prototype could be integrated into it and demonstrated the integration through parsing of a simple JAR file.

Acknowledgements

I would like to thank Dr. Zoltán Porkoláb, Associate Professor of the Department of Programming Languages and Compilers at the Faculty of Informatics for admitting me to the CodeCompass project, supplying the thesis topic and helping with the documentation. I would also like to say thanks to Tibor Brunner for providing information about the project's architecture and the rest of the developers of CodeCompass and CodeChecker for their technical help through the mailing list.

CodeCompass is an open-source software project associated with Ericsson company and is part of *Model C++*, a joint research and development project between Eötvös Loránd University and Ericsson Hungary Ltd.

Table of contents

List of figures

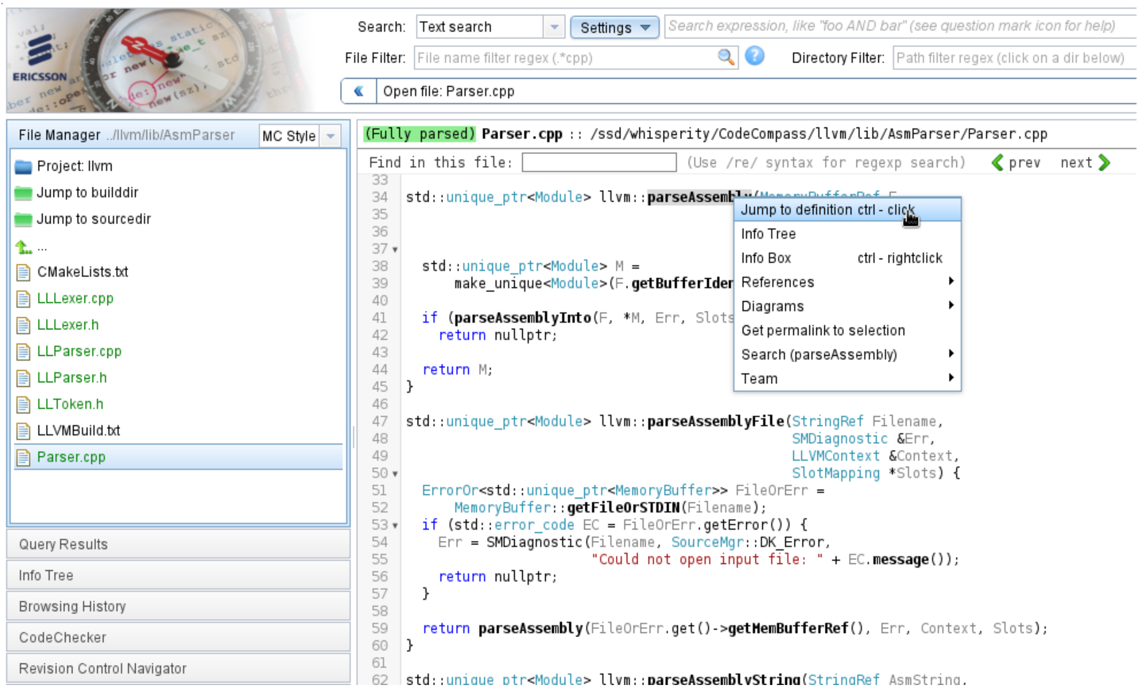
Code listings

Chapter 1

Introduction

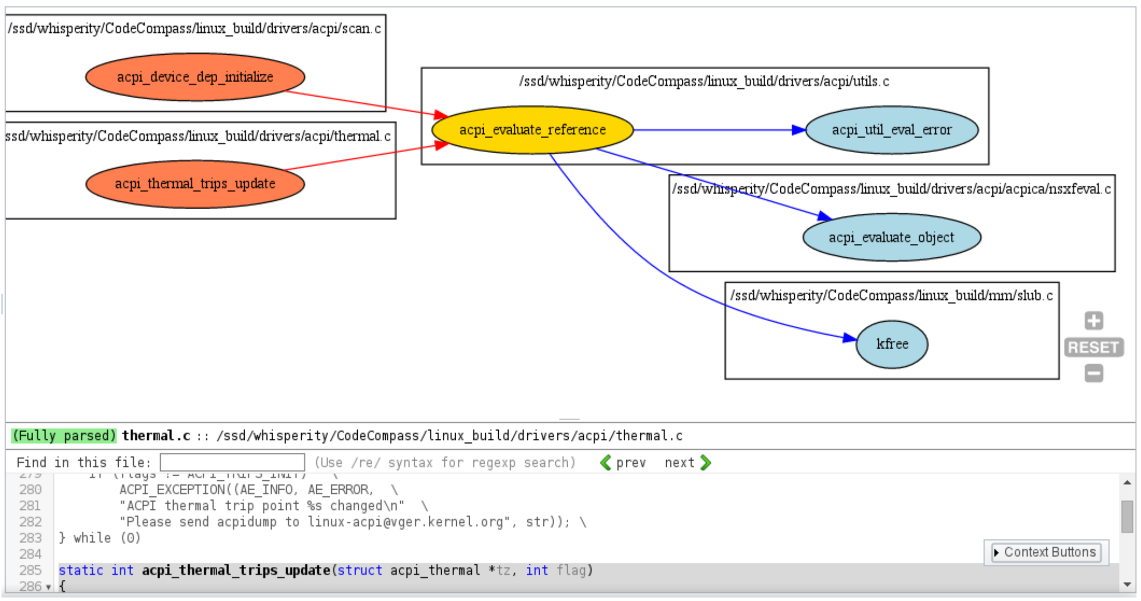
Over time, codebase of a software project can grow to very large proportions, especially with many developers contributing to it. Large codebases tend to get incomprehensible, making them harder for new members of a project to get involved into development or to generally implement new features, create bugfixes and maintain technical debt. CodeCompass is an open-source code comprehension tool, written in C++, Java and JavaScript with the motivation to make large codebases more scalable by providing navigation and visualization to the user. It's capable of parsing source files of software projects, storing the Abstract Syntax Trees (AST) of source files in database, and displaying information about the project's code on a web front-end. CodeCompass runs a webserver, so the parsed project can be accessed remotely.

This project is associated with Ericsson company and is hosted on GitHub at `https://github.com/Ericsson/CodeCompass`. It also supports integration with another Ericsson project, CodeChecker, a static analyzer toolchain. The web user interface provides class inheritance diagrams, function callpath visualization, and a source browser, where the user is capable of navigation by clicking and jumping through source AST node elements. Below are two screenshot examples, Figure 1.1 shows the source browser with parsed LLVM project, while Figure 1.2 shows a two-way function callgraph display within the Linux kernel codebase.


Search: Text search
Settings
Search expression, like "foo AND bar" (see question mark icon for help)
File Filter: File name filter regex (.*cpp)
Directory Filter: Path filter regex (click on a dir below)
Open file: Parser.cpp
File Manager ../llvm/lib/AsmParser
MC Style
Project: llvm
Jump to builddir
Jump to sourcedir
...
CMakeLists.txt
LLLexer.cpp
LLLexer.h
LLParser.cpp
LLParser.h
LLToken.h
LLVMBuild.txt
Parser.cpp
Query Results
Info Tree
Browsing History
CodeChecker
Revision Control Navigator
Jump to definition ctrl - click
Info Tree
Info Box ctrl - rightclick
References
Diagrams
Get permalink to selection
Search (parseAssembly)
Team

```
(Fully parsed) Parser.cpp :: /ssd/whisperity/CodeCompass/llvm/lib/AsmParser/Parser.cpp
Find in this file:            (Use /re/ syntax for regexp search)   prev   next

std::unique_ptr<Module> llvm::parseAssembl

  std::unique_ptr<Module> M =
      make_unique<Module>(F.getBufferIder

  if (parseAssemblyInto(F, *M, Err, Slots
    return nullptr;

  return M;
}

std::unique_ptr<Module> llvm::parseAssemblyFile(StringRef Filename,
                                                SMDiagnostic &Err,
                                                LLVMContext &Context,
                                                SlotMapping *Slots) {
  ErrorOr<std::unique_ptr<MemoryBuffer>> FileOrErr =
      MemoryBuffer::getFileOrSTDIN(Filename);
  if (std::error_code EC = FileOrErr.getError()) {
    Err = SMDiagnostic(Filename, SourceMgr::DK_Error,
                       "Could not open input file: " + EC.message());
    return nullptr;
  }

  return parseAssembly(FileOrErr.get()->getMemBufferRef(), Err, Context, Slots);
}

std::unique_ptr<Module> llvm::parseAssemblyString(StringRef AsmString,
```

Figure 1.1: C++ source file of LLVM opened in the source browser. User can jump to function definition by right clicking on the function name and selecting "Jump to definition".


/ssd/whisperity/CodeCompass/linux_build/drivers/acpi/scan.c
acpi_device_dep_initialize
ssd/whisperity/CodeCompass/linux_build/drivers/acpi/thermal.c
acpi_thermal_trips_update
/ssd/whisperity/CodeCompass/linux_build/drivers/acpi/utils.c
acpi_evaluate_reference
acpi_util_eval_error
/ssd/whisperity/CodeCompass/linux_build/drivers/acpi/acpica/nsxfeval.c
acpi_evaluate_object
/ssd/whisperity/CodeCompass/linux_build/mm/slub.c
kfree
RESET
Context Buttons

```
(Fully parsed) thermal.c :: /ssd/whisperity/CodeCompass/linux_build/drivers/acpi/thermal.c
Find in this file:            (Use /re/ syntax for regexp search)   prev   next
        ACPI_EXCEPTION((AE_INFO, AE_ERROR,  \
        "ACPI thermal trip point %s changed\n"  \
        "Please send acpidump to linux-acpi@vger.kernel.org", str)); \
} while (0)

static int acpi_thermal_trips_update(struct acpi_thermal *tz, int flag)
{
```

Figure 1.2: Callgraph of a function defined in Linux kernel source. The graph lists callers (red ellipses) and callees (blue ellipses) of the selected function (yellow ellipse).

CodeCompass currently supports deep parsing of sources written in C, C++, Java and shallow parsing features for certain script languages like Python, Bash, Perl and even Makefiles. Parsing of Java sources was made possible with Eclipse

Java Development Tools (Eclipse JDT) as third-party library, which contains a utility for traversing textual Java source code.

Java Archive, also known as JAR format is a convenient archive format[1] to hold compiled Java classes, the main executable class and miscellaneous resource files used by program like images and configuration files. Nowadays, IDEs like Eclipse or JetBrains IntelliJ IDEA are capable of displaying classes in decompiled form from JAR libraries. This helps developers while debugging their running Java programs with debuggers, as they can step through lines of decompiled sources in libraries that they don't own the original sources to. It is much easier to decompile and reverse engineer programs and libraries compiled in Java bytecode because of its portability across all platforms, while binary machine code compiled from C or C++ languages are specific to certain processor architectures, operating systems, ABIs and also contain lower language level optimizations. Because of this, even things like building parse trees out of compiled bytecode programs and statically or dynamically analyzing them is possible. FindBugs (`http://findbugs.sourceforge.net/`), a Java bug finder static analysis tool is using a similar approach to search for bugs even in compiled objects stored in JAR libraries[2].

A need for JAR file parsing was proposed for CodeCompass. This thesis documents research and development of such JAR reader module for the project. At the end, the very least what had to be done is to list out information about Java classes, like member variables, inheritance and methods. Chapter 2 lays down technical background information for reversing compiled Java bytecode from JAR files while Chapter 3 gives an overview about existing Java reverse engineering tools and solutions. From the technical background knowledge and with a third-party library, a prototype application was made as a basis for implementing a JAR reader which is described in Chapter 4. Finally, Chapter 5 deals with integration of said prototype into the existing Java parsing process of CodeCompass.

Full recreation of Java source code from JAR content is out of the scope for this thesis. Instead, prototype and CodeCompass integration translate bytecode into compilable Java class structure and commented out JVM instructions as method bodies, while Section 2.4 discusses decompilation techniques to use for the future. Hopefully, this reading will be useful for future contributors to the JAR parsing module of CodeCompass or anyone who is interested in the field of reverse engineering Java applications and libraries.

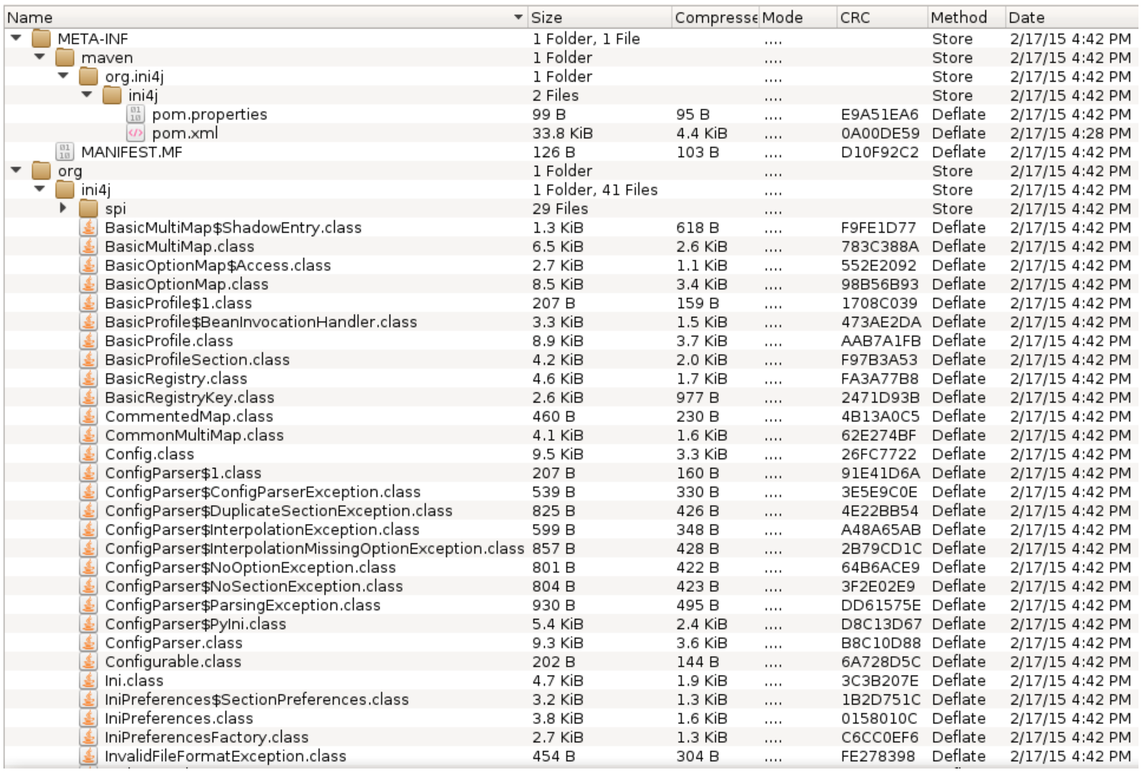
Chapter 2

Technical background for extracting code information

2.1 JAR format and PKZIP format it's based on

JAR files contain Java class files in compiled bytecode form, resources, and an optional manifest file describing such information like classpath, version, and the main entry point of execution. These archive files can either be used as executables if class with `main()` method was declared or just as simple program libraries like DLLs and shared object files.

JAR is a storage format based on the ZIP binary file format, more specifically on PKZIP. They share the same characteristics and can be treated as normal ZIP files. Figure 2.1 shows the contents of ini4j's JAR file, an INI file parsing Java library opened in ark, a graphical archiver utility under Linux.



Name	Size	Compresse	Mode	CRC	Method	Date
META-INF	1 Folder, 1 File		....		Store	2/17/15 4:42 PM
maven	1 Folder		....		Store	2/17/15 4:42 PM
org.ini4j	1 Folder		....		Store	2/17/15 4:42 PM
ini4j	2 Files		....		Store	2/17/15 4:42 PM
pom.properties	99 B	95 B	....	E9A51EA6	Deflate	2/17/15 4:42 PM
pom.xml	33.8 KiB	4.4 KiB	....	0A00DE59	Deflate	2/17/15 4:28 PM
MANIFEST.MF	126 B	103 B	....	D10F92C2	Deflate	2/17/15 4:42 PM
org	1 Folder		....		Store	2/17/15 4:42 PM
ini4j	1 Folder, 41 Files		....		Store	2/17/15 4:42 PM
spi	29 Files		....		Store	2/17/15 4:42 PM
BasicMultiMap$ShadowEntry.class	1.3 KiB	618 B	....	F9FE1D77	Deflate	2/17/15 4:42 PM
BasicMultiMap.class	6.5 KiB	2.6 KiB	....	783C388A	Deflate	2/17/15 4:42 PM
BasicOptionMap$Access.class	2.7 KiB	1.1 KiB	....	552E2092	Deflate	2/17/15 4:42 PM
BasicOptionMap.class	8.5 KiB	3.4 KiB	....	98B56B93	Deflate	2/17/15 4:42 PM
BasicProfile$1.class	207 B	159 B	....	1708C039	Deflate	2/17/15 4:42 PM
BasicProfile$BeanInvocationHandler.class	3.3 KiB	1.5 KiB	....	473AE2DA	Deflate	2/17/15 4:42 PM
BasicProfile.class	8.9 KiB	3.7 KiB	....	AAB7A1FB	Deflate	2/17/15 4:42 PM
BasicProfileSection.class	4.2 KiB	2.0 KiB	....	F97B3A53	Deflate	2/17/15 4:42 PM
BasicRegistry.class	4.6 KiB	1.7 KiB	....	FA3A77B8	Deflate	2/17/15 4:42 PM
BasicRegistryKey.class	2.6 KiB	977 B	....	2471D93B	Deflate	2/17/15 4:42 PM
CommentedMap.class	460 B	230 B	....	4B13A0C5	Deflate	2/17/15 4:42 PM
CommonMultiMap.class	4.1 KiB	1.6 KiB	....	62E274BF	Deflate	2/17/15 4:42 PM
Config.class	9.5 KiB	3.3 KiB	....	26FC7722	Deflate	2/17/15 4:42 PM
ConfigParser$1.class	207 B	160 B	....	91E41D6A	Deflate	2/17/15 4:42 PM
ConfigParser$ConfigParserException.class	539 B	330 B	....	3E5E9C0E	Deflate	2/17/15 4:42 PM
ConfigParser$DuplicateSectionException.class	825 B	426 B	....	4E22BB54	Deflate	2/17/15 4:42 PM
ConfigParser$InterpolationException.class	599 B	348 B	....	A48A65AB	Deflate	2/17/15 4:42 PM
ConfigParser$InterpolationMissingOptionException.class	857 B	428 B	....	2B79CD1C	Deflate	2/17/15 4:42 PM
ConfigParser$NoOptionException.class	801 B	422 B	....	64B6ACE9	Deflate	2/17/15 4:42 PM
ConfigParser$NoSectionException.class	804 B	423 B	....	3F2E02E9	Deflate	2/17/15 4:42 PM
ConfigParser$ParsingException.class	930 B	495 B	....	DD61575E	Deflate	2/17/15 4:42 PM
ConfigParser$PyIni.class	5.4 KiB	2.4 KiB	....	D8C13D67	Deflate	2/17/15 4:42 PM
ConfigParser.class	9.3 KiB	3.6 KiB	....	B8C10D88	Deflate	2/17/15 4:42 PM
Configurable.class	202 B	144 B	....	6A728D5C	Deflate	2/17/15 4:42 PM
Ini.class	4.7 KiB	1.9 KiB	....	3C3B207E	Deflate	2/17/15 4:42 PM
IniPreferences$SectionPreferences.class	3.2 KiB	1.3 KiB	....	1B2D751C	Deflate	2/17/15 4:42 PM
IniPreferences.class	3.8 KiB	1.6 KiB	....	0158010C	Deflate	2/17/15 4:42 PM
IniPreferencesFactory.class	2.7 KiB	1.3 KiB	....	C6CC0EF6	Deflate	2/17/15 4:42 PM
InvalidFileFormatException.class	454 B	304 B	....	FE278398	Deflate	2/17/15 4:42 PM

Figure 2.1: Inside of ini4j JAR file opened in an archive program. Looking at the header bar, hints about inner ZIP structure can be seen like CRC and compression method. This is described down below.

The format of PKZIP is specified by the *.ZIP File Format Specification*[3], published by PKWARE Inc. ZIP is only a file storage and archive format like TAR, as it doesn't specify what compression method or algorithm to use. It can even store files in their original uncompressed state too, such storage method is called `STORE` in the PKZIP terminology. The most used compression method however is `DEFLATE`, a lossless compression algorithm used by most archiving utilities, which utilizes the combination of LZ77 compression algorithm and Huffman coding[4]. Figure 2.2 below summarizes the structure of a ZIP file.

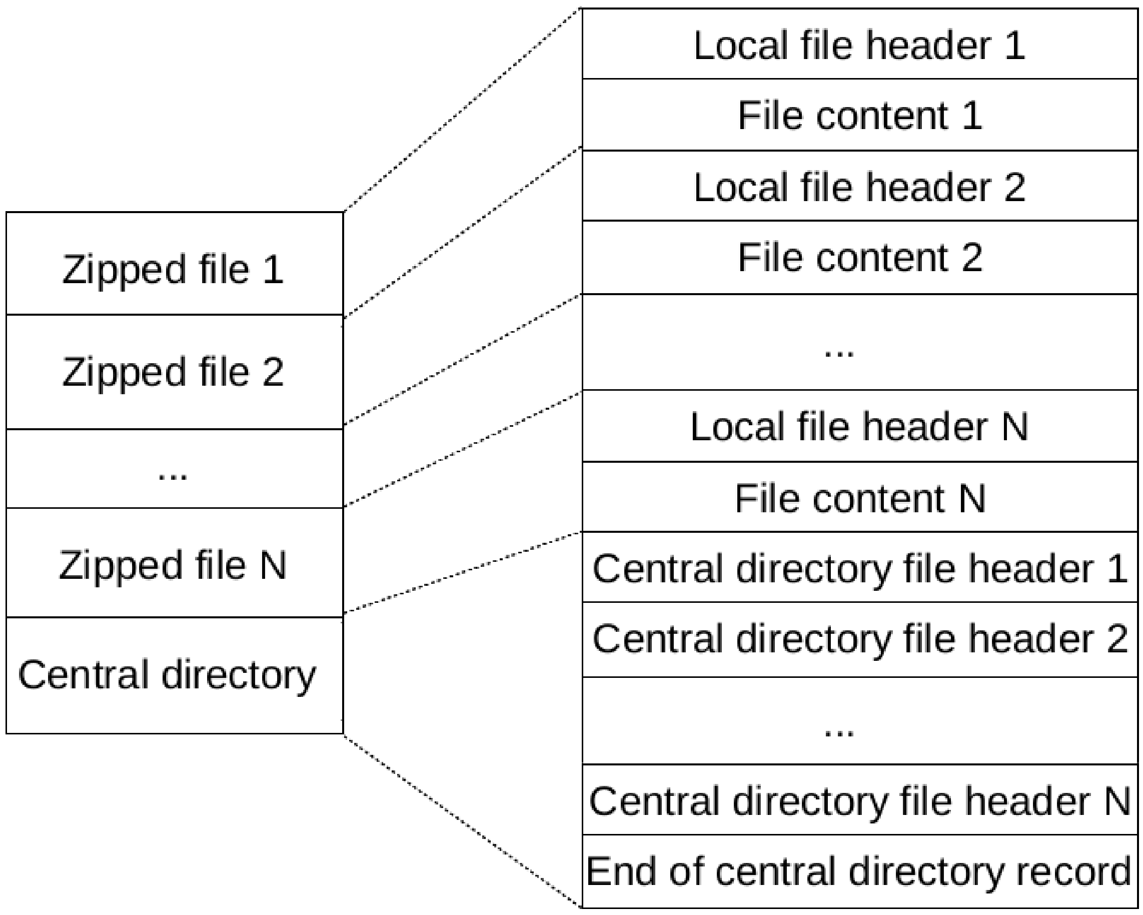


Figure 2.2: Structure of a PKZIP file. Left figure shows overview regions of the PKZIP structure while right figure maps the correct segments to those regions.

Stored or compressed contents of files are enumerated as entries at the beginning of ZIP file. Each entry is prepended with a *local file header*, that describes meta-information, including method of the compression, name and creation date of file, both compressed and uncompressed size, and CRC-32 checksum. Optionally, checksum and size is contained in *data descriptor* instead, that is appended to end of the entry.

Central directory summarizes the local file headers while carrying additional information, like file attributes and also describes the offset of each local headers from start of the ZIP file. The central directory is composed of *central directory file headers* for each stored file, while *end of central directory record* points to the start of the first central directory file header. If encryption is specified, an optional *archive decryption header* is prepended before the whole central directory.

Multibyte values in headers and central directory are stored in little-endian order and compression is only applied to contents of file entries, not PKZIP metadata.

Folders are stored as ZIP entries too, but their content part is empty and is described by the file entries instead.

Florian Bucholz provided a more detailed overview of ZIP file headers, describing each header field and providing examples[5]. Listing A.1 of Appendix A on page 55 shows a representation of PKZIP file format in C programming language.

2.2 Inside of a class file

2.2.1 Class file structure

Class file contains the type information and bytecode instructions for a Java Class. This type information includes definition of class or interface, name of inherited super class, implemented interfaces, defined fields and methods. Unlike PKZIP format before, multi-byte information and structs are contained in big-endian order. Chapter 4 of *The Java Virtual Machine Specification* provides information about class file structure[6]. The file is structured in the following way:

- Class header
- Constant pool
- Class access specifier flags
- Class name
- Inherited super class name
- Implemented interfaces
- Declared fields
- Declared methods
- Attributes

Figure 2.3 visualizes the organization of a class file.

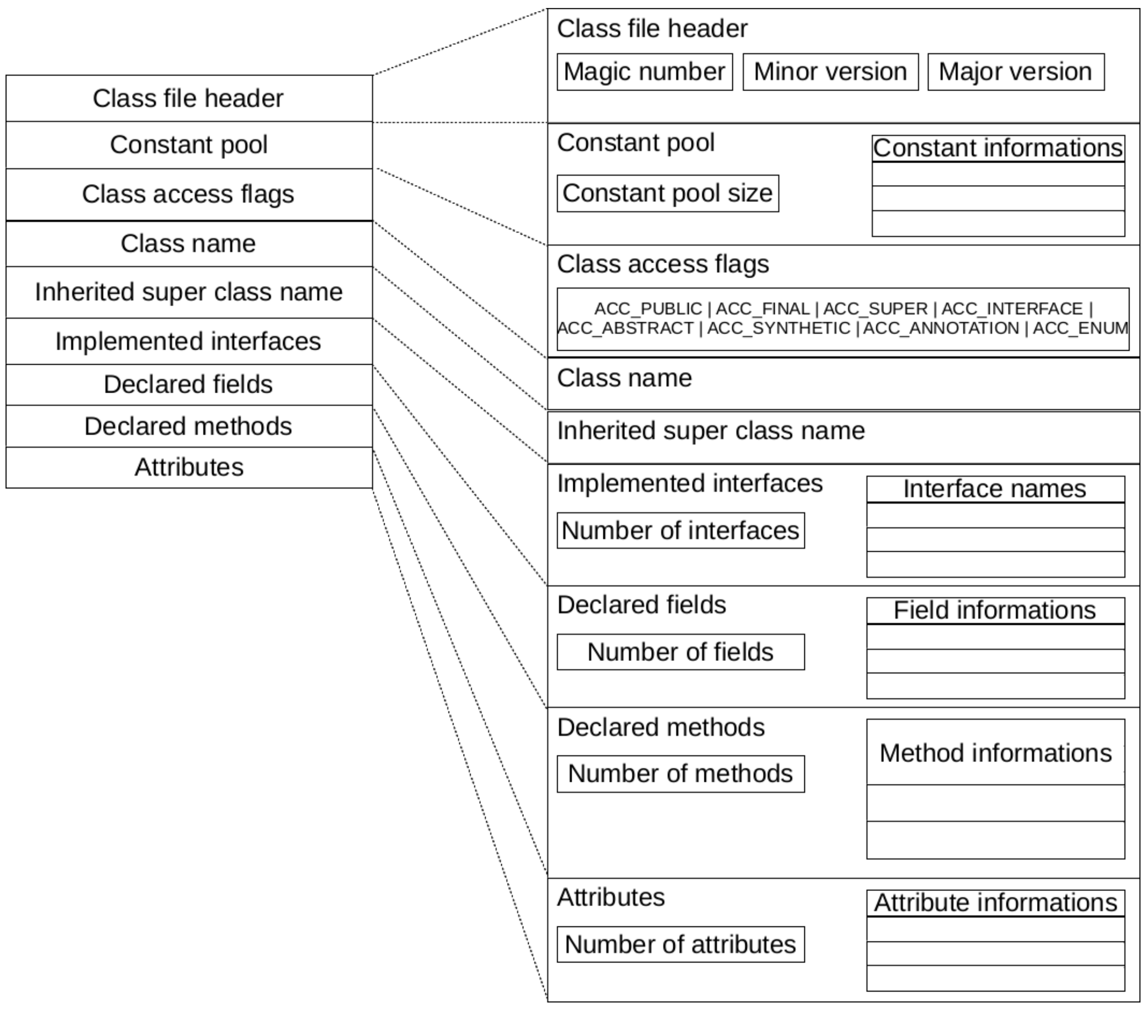


Figure 2.3: General structure of a Java class file. Just like with Figure 2.2, left figure represents an overview while right side lists details.

Header of a class file contains a 32-bit "magic number", the `0xCAFEBABE` signature, which denotes that it's a valid class file. The header also lists major and minor version numbers defining the compiler version which was made by. This makes sure that older Java runtimes wouldn't load incompatible classes generated by newer compilers.

Constant pool contains symbolic UTF-8 encoded string constants like class- and interface names, field descriptors, method descriptors and strings used as values. It even stores floating point values used in Java program. Instead of storing values themselves in further parts of the class structure or in instruction operands, constants are referenced by their index in the constant pool to preserve space in bytecode. Each symbolic constant is denoted by a byte-sized tag, describing what that constant is for, like if it's a method name or just a simple string value used in the program. These two, the byte-tag and the string form the *constant information* (`contant_info`) structure. The JVM builds a *run-time constant pool* for each class to resolve method and field references at run-time.

Access flags are a bitmask of bytes representing the visibility of a class, field or method. Among the available flags is the `SYNTHETIC` flag, which marks that the element was generated by the compiler and is not present in Java source. For example, class files have no concept of nested inner classes in their structure. They are just plain new class files with "$" symbol separating the defining- and nested class name, like `ParentClass$NestedClass.class`. Reference to the defining class is stored in the nested class as a compiler-generated member field.

Below the access flags for class are information about the name of class, name of super class it inherits from and name of implemented interfaces if there are any. As mentioned before, these names are not textual representations, but references to their spot in the constant pool.

Each field and method is represented by their *field information* (`field_info`) and *method information* (`method_info`) structures, described by their *attributes* related to them. Attributes are listed at the end of class structure and they can also be used by the class. They can be exceptions, annotations, information for reflection APIs, or optional debugging information used by IDEs. Instructions of a method are also stored in a `Code` attribute, which contains not only the instructions and operands for the JVM, but maximum size of the method's stack frame and the number of local variables.

Just as with the PKZIP file format, Listing A.2 shows C `struct` representation of a Java class file in Appendix A on page 58.

2.2.2 Disassembly of a Class file

`javap` is a Java Class file disassembler[7] which is issued with every JDK installation. It can be invoked from the command line the following way:

```
$ javap -c HelloWorld.class
```

Listing 2.1: Disassembly output of HelloWorld.class

```
Compiled from "HelloWorld.java"
  public class HelloWorld {
    public HelloWorld();
      Code:
        0: aload_0
        1: invokespecial #1  // Method java/lang/Object."<init>":()V
        4: return

    public static void main(java.lang.String[]);
      Code:
        0: getstatic     #2  // Field java/lang/System.out:Ljava/io/PrintStream;
        3: ldc           #3  // String Hello world!
        5: invokevirtual #4  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        8: return
}
```

Listing 2.1 shows the disassembled Class file and its JVM instructions as JVM assembly. More specifically, the `Code` attribute of each method is displayed. Indices are shown before the JVM instructions, denoting the offsets from beginning of method body and next to the instructions are references to entries in the constant pool. Reason for the jumps between indexes is that operands like arithmetic operands or method invocations are not part of the instruction opcode, but take up separate space in the bytecode. You can write bytecode programs in JVM assembly too by using the Jasmin assembler, which can be found at
`http://jasmin.sourceforge.net`.

In this next command, I used several flags in order to have a more verbose output:

```
$ javap -c -s -l -private -verbose HelloWorld.class
```

Flag `-c` prints the disassembled code of method instructions in JVM assembly output. Flag `-s` prints internal type signatures. Flag `-l` prints line tables and local variable tables if there are any. Flag `-private` prints out all class members, including private ones. Lastly, flag `-verbose` prints the method stack size, number of locals, and arguments for methods. Output provided by this command really showcases the inside of a class file, including the class header, constant pool and attributes. The output is listed in Appendix A as Listing A.3 on page 60.

During compilation, information about local variable names and types are lost and input parameters of a method is treated the same way as local variables. A `LocalVariableTable` attribute can be embedded into class optionally by IDEs or by `javac` using `-g` flag to provide debugging information (especially when starting debugging from IDE) and recover names and types for variables of these scopes.

2.3 Interpreting bytecode

Bytecode is a binary format commonly used to distribute programs in an architecture-neutral way, so they can be executed on any platform that has a runtime installed that interprets it. It can be seen as a language implementation pattern[8]. For example, instead of direct textual interpretation or evaluation of parse trees in scripting languages, most scripting languages like JavaScript, Python, Ruby and Perl assemble text into intermediate bytecode representation at the start of execution and evaluates that code.

The code contains series of bytes representing opcodes and operands. Each opcode is usually one byte, 8-bit sized, hence the name "bytecode". These byte-sized instructions are very easy to parse, there's no need for building a parser tree or using any similar parsing techniques. This code gets loaded into memory, then gets executed by an abstract machine implementation called the *virtual machine* (VM), complete with instruction pointer register pointing to current opcode and a simulated CPU executing these low-level instructions.

The two ways to handle operands in bytecode are to either separate data from instructions and denote the operand literal by an opcode specified for this (like `0x00`), or use 16-bit instructions instead of 8-bit ones and make the operand part of the opcode. Some VM architectures use 16-bit instruction set instead of 8-bit one.

The runtime environment of Java and C# also have bytecode interpreting virtual machines, although their source files must be compiled into binary objects containing bytecode language before execution. Bytecode is low-level enough to be more performant than interpreting text and is easier to be generated than machine code.

The virtual machine for the Java platform is provided by the *Java Virtual Ma-*

chine, or JVM for short. JVM is a specification that has many implementations existing for it, Java HotSpot™VM being the reference implementation[9]. Since JVM only knows about bytecode, it doesn't matter if it was compiled from Java, Scala, Groovy, Clojure, Jython language or if it was made by bytecode optimizers, obfuscators or aspect-oriented compilers. Java's compiler, `javac` only does very little optimization during compiling bytecode, since most of the optimization is done by the *Just-in-Time* (JIT) compiler of the JVM during runtime. In Java version 9 however, there will be an experimental *Ahead-of-Time* (AOT) compilation feature, which generates native code from compiled bytecode before even starting the JVM[10].

There are two ways to implement virtual machines. They can either be *stack-based* or *register-based* machines. Standard JVM implementations use the former architecture.

2.3.1 How stack-based virtual machines work

Since the first widely-known stack-based virtual machine, Pascal's P-code Machine[11], stack machine is a common and traditional way to implement bytecode VMs. Instead of storing local variables, method parameters and temporary results of expressions in registers, a stack-based virtual machine stores them in an *operand stack*. This stack is only for storing these values and is separate from the call stack, which contains stack frame pointers. During executing bytecode program, the values are pushed onto the operand stack and accessed in a last-in-first-out (LIFO) order.

To illustrate the difference between register- and stack-based architecture, let's take the example function from Listing 2.2 that defines arithmetic and variable storing expressions shown in a C-like high-level language and see how it's transformed to instructions for each architecture.

Listing 2.2: Example function defined in a C-style language to be compiled into register- and stack-based architectures. This example can be compiled by both C/C++ and Java compilers.

```
int f(int x, int y, int z) {
  int n = x * (y + z);
  return n;
}
```

In register-based architectures, the function would translate into the instructions shown in Listing 2.3.

Listing 2.3: Function from Listing 2.2 compiled into instructions for register-based hardware architecture, shown in Intel assembly syntax. Code was generated by GNU GCC 6.3 compiler without optimizations enabled (`-O0` flag).

```
; Function prologue
push    rbp                         ; Save stack frame base pointer of calling function
mov     rbp, rsp                    ; Set stack frame base pointer to current function

; f(int x, int y, int z)
mov     DWORD PTR [rbp-20], edi ; Parameter int x
mov     DWORD PTR [rbp-24], esi ; Parameter int y
mov     DWORD PTR [rbp-28], edx ; Parameter int z

; int n = x * (y + z)
mov     edx, DWORD PTR [rbp-24] ; Move y to edx register
mov     eax, DWORD PTR [rbp-28] ; Move z to eax register
add     eax, edx                    ; Add two registers together, store result in eax
imul    eax, DWORD PTR [rbp-20] ; Multiple previous result with x, store result in eax
mov     DWORD PTR [rbp-4], eax  ; Move result to local variable n

; return n
mov     eax, DWORD PTR [rbp-4]  ; Return value of n

; Function epilogue
pop     rbp                         ; Pop stack frame of current function to get calling one
ret                                 ; Return to calling function
```

While in a stack-based architecture, it would otherwise translate to the virtual machine instructions shown in Listing 2.4.

Listing 2.4: Function from Listing 2.2 compiled into instructions for stack-based VM architecture, shown in JVM assembly. Code was first compiled as Java bytecode, then disassembled with `javap`.

```
// f(int x, int y, int z)
0: iload_1          // Push value of x parameter onto stack
1: iload_2          // Push value of y parameter onto stack
2: iload_3          // Push value of z parameter onto stack

// int n = x * (y + z)
3: iadd             // Pop value z and y from stack, add them together,
                    // then push result back onto stack
4: imul             // Pop previous result, pop x, multiple them,
                    // then push result back onto stack
5: istore       4   // Pop result on top of stack and store it in "Local variable 4"

// return n
7: iload        4   // Load and push value from "Local variable 4"
9: ireturn          // Return with value on top of stack
```

Operand access in stack-based architecture is implicit for the instructions. Operations are represented in postfix order and the operator precedence is defined by the ordering of operands in the stack. The expression`n = x * (y + z)` from Listing 2.2 is executed by the stack VM as `n = x y z + *`. Local variables themselves are not located in the operand stack, but they are part of the stack frame in the function stack and values are accessed from them by storing- and loading instructions. Figure 2.4 visualizes these operations on the operand stack.

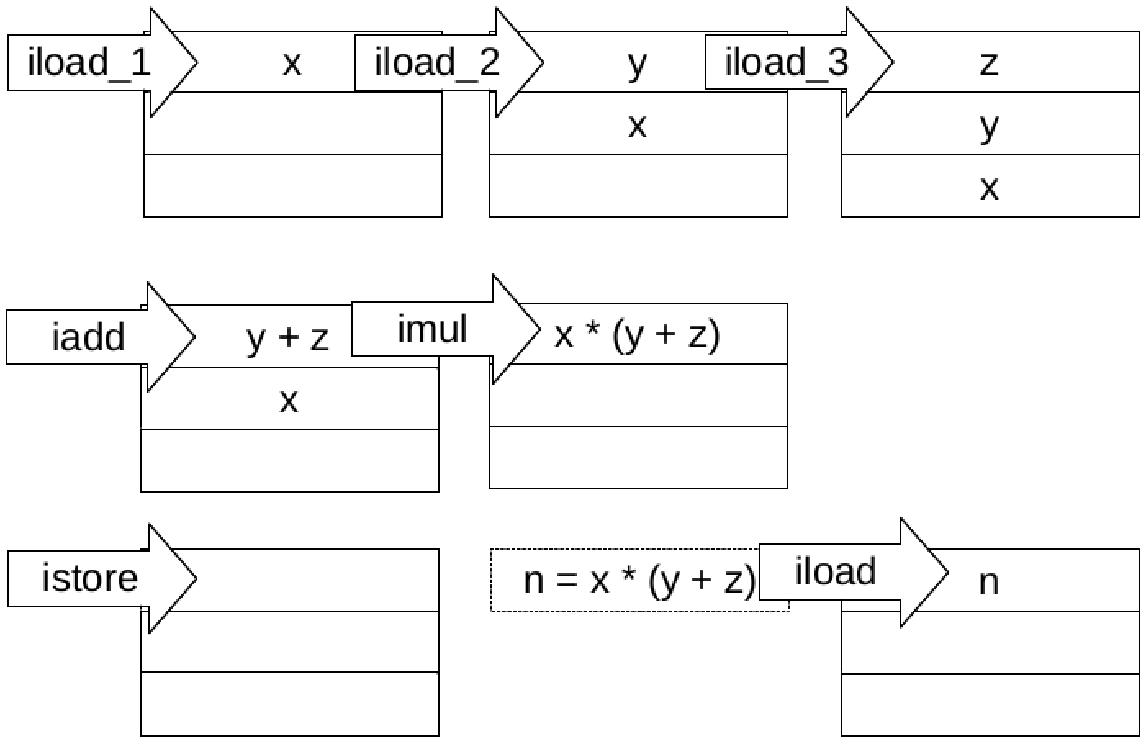


Figure 2.4: Flow of Listing 2.4 operations on operand stack. Dashed box represents result of expression that is not in the stack, but stored in local variable which is in the stack frame.

When a method is invoked, a new stack frame is pushed onto the function stack, values of the parameters are pushed onto the operand stack and instruction pointer is moved to start of invoked method code. At the end of method call, return value is on top of operand stack and stack frame is popped from function stack.

Note, that in the case of JVM running in a concurrent or parallel environment, each thread has their own intruction pointer, operand- and call stack, but heap memory is shared across all threads in the JVM. Even if JVM is a stack machine, hardware registers still have to be taken into consideration when dealing with multiprogramming.

Overall architecture of register machines are similar to stack machines, except operands are stored in simulated registers instead. Operands can be accessed in an array of registers and the number of registers can be infinite in theory, hence it depends on the implementer to decide how many registers the VM should use. The number of operations are fewer, instruction set has a smaller size. Stack operations might be more costly, what might be done in a register machine by one instruction, stack machine requires several more for that because of pushing to- and popping off operands from the stack.

However, register machines need separate register allocator algorithm, and a single register instruction takes up more space, usually 2 or 4 bytes, compared to 1 byte for a stack machine instruction. Register instructions store their operands too, therefore decoding them have an overhead. Stack machines have a simplistic design, they are also easier to write for multiple hardware platforms and easier to generate bytecode for them, as there's no need for the explicit addressing of operands.

Lua used to implement a stack-based virtual machine too, until switching to register-based machine from version 5.0.[12]. Some authors argue in the defense of using register-based architecture for the JVM and implemented their own register-based virtual machines that translate bytecode written for the original JVM into register code[13, 14]. The Dalvik Virtual Machine used in Android is a register-based machine. It was created to run with limited hardware resources and have multiple VM instances of them as processes running at the same time[15, 16]. Although it uses Java as programming language, the JVM bytecode is compiled to it's own format, the Dalvik bytecode with the `dx` tool. Dalvik was later succeeded by the Android Runtime, which uses AOT compilation for translating bytecode to native machine code in ELF format.

For comprehension purposes, I created a simple stack-based virtual machine in C++, which uses `switch` for instruction dispatch. The code is listed in Appendix A as Listing A.4 on page 62. It uses STL `vector` to represent loaded code in memory and `stack` to represent the operand stack. An example program can be "generated" on the fly in a Bash shell with `echo` and `xxd` hexdump utility. The following bytecode corresponds to the arithmetic operations `(6 + 3) * 10` and is written to `mybytecode.bc` file:

```
$ echo -n "00 06 00 03 01 00 0a 03" | xxd -r -p > mybytecode.bc
```

Listing 2.5 illustrates the execution of this expression.

Listing 2.5: Execution of example VM created in Listing A.4.

```
PUSH 6  ; Push value 6 onto stack
PUSH 3  ; Push value 3 onto stack
ADD     ; Pop value 4 and 3 from stack, add them together and push result 9 back onto stack
PUSH 10 ; Push value 10 onto stack
MUL     ; Pop value 10 and 9 from stack, multiply them together
        ; and push result 90 back onto stack
```

2.3.2 Instructions of the JVM

The standard JVM uses 8-bit long, byte-sized instructions. If an operand takes up more size then one byte, like 16-, 32-, or 64-bit long values, multiple bytes in big-endian order are concatenated with bitshift- and bitwise operations. If an operand wouldn't fit into several bytes or if it's a reference type, than the operand would be an index pointing to the symbolic constant in the contant pool.

Chapter 6 of *The Java Virtual Machine Specification* provides reference about these instructions. Because of the byte-size restriction, number of instructions for the JVM is limited to a maximum of 256. Not all of the bytes are used as instructions as some opcodes are reserved for future use. Instructions can be even high-level operations, like method invocation, type casting and using monitors for concurrent programming. Instruction set of the JVM can be split into these categories:[17]

- Stack operations
- Arithmetic operations
- Control flow
- Load and store operations
- Field access
- Method invocation
- Object allocation
- Conversion and type checking
- Synchronization instructions

To keep static typing intact, stack-, arithmetic-, comparison-, conversion-, loading- and storing instructions are typed for primitive types. There are separate instructions for certain primitive types, prefixed with the first letter of the primitive type name, like `iload`, `istore`, `iadd`, `isub`, `imul` and `idiv` for integer types, `fload`, `fstore`, `fadd`, `fsub`, `fmul`, `fdiv` for floating-point types and so on. For loading and storing reference types, `aload` and `astore` are used.

Branching is done with `ifeq`, `ifne`, `iflt`, `ifge`, `ifgt`, `ifle` conditionals. Loop constructs like `while` and `for` are translated to `goto`[1] and branching instructions. Exceptions are literally just `goto`s. When recreating code, it's hard to distinguish them from real control structures. Exception handling is represented by a table containing the starting instruction offset, finishing instruction offset, name of the exception class and offset of the first exception handling instruction. JVM checks this table if the offset of the instruction where the exception occured is in the range between the start and end offset in the table, then moves the instruction pointer to the exception handling code.

Method invocation instructions for Java have the following types:

- `invokestatic`
- `invokevirtual`
- `invokeinterface`
- `invokespecial`

`invokestatic` is used to invoke static methods that don't require instantiation of the class. `invokevirtual` is used to invoke instance methods of an instantiated object and resolving polymorphic methods. During member method call, object reference will be the first parameter to the method as an implicit `this`. Since all Java classes have to implicitly inherit from `java.lang.Object` superclass, all instance method calls are virtual by default. `invokeinterface` is used for calling interface methods and resolving the method to its implementation.

[1]`goto` keyword exists in Java language, but it's not implemented, therefore it's use is illegal. It was supposed to be part of Java, but was removed with the reasoning that `break`/`continue` or extracting code into method can be used instead.

invokespecial is used to call *special methods*. These are instance initialization methods, private methods and methods of superclass. The default constructor is created by the compiler and has the <init> signature. There's also invokedynamic that's left out of the listing. It should not be confused with dynamic method call within polymorphism. This instruction is only added to the JVM in order to make implementing new dynamically typed programming languages other than Java, like Groovy and Clojure possible.

Even concurrency, entering and exiting synchronized blocks in Java are low-level JVM operations, achieved by monitorenter and monitorexit. Java's mutual exclusion and synchronized blocks are based on the *monitor* synchronization pattern, where whole classes, methods or blocks can be wrapped with mutual exclusion. monitorenter sets a monitor to the object reference on top of stack and makes the thread owner of that monitor. When another thread wants to execute monitorenter instruction to lock the same object, that thread blocks until previous thread exits the monitor for the object.

2.4 Concepts for decompiling bytecode instructions

We already saw the usage javap, although it's only just a disassembler, not a decompiler. Decompilation is the act of taking a low-level language and transforming it to a higher-level language representation. In this case, a Java decompiler takes JVM bytecode as input and translates even the method instructions to Java source code that is able to be compiled again.

Decompiling Java bytecode is easier to do than machine code, since type information and members of a class can be recovered easily, knowing the class file structure from Section 2.2.1, and instructions are uniform for all architectures and JDK versions. Correctness of decompilation however depends on the input bytecode and there's no one-hundred percent solution for retrieving original source, as certain informations of the original source get lost during compilation. Moreover, obfuscators may hinder decompilation by renaming symbols, obfuscating control flow and creating bogus Java threads, making decompilers retrieve convoluted Java code or even fail during decompilation. On the other hand, good decompilers can even "outsmart" them and transform obfuscated code to readable valid Java source.

Symbolic information of assembly, C and C++ are in data segments and subroutine names of binary object, while Java symbols are in the constant pool of class files. These can be collected first into a symbol table than can be inserted into the high-level language output. In bytecode, data and instructions are especially well separated. It's trivial to recover arithmetic expressions, variable assignments, method calls and object instantiation just by recognizing certain patterns in compiled code. The main concern however, is recreating the correct high-level control structures or exception handling flow from low-level branching and jump instructions, which is not possible without further code analysis.

In essence, a decompiler can be regarded as a regular compiler, that instead of taking textual source as input and producing binary, it takes binary code as input and produces source. Cristina Cifuentes describes a decompiler architecture originally for decompiling machine code compiled with a C compiler in her PhD thesis[18]. The general idea is to generate a higher-level *intermediate representation* (IR) from compiled code, apply transformations on this IR, then generate the high-level source code from the IR. IR can be data structures representing pseudo code, for example translating several register instructions that store a value into a variable to `Assignment(x, y)`. The decompiler is divided into the following phases:

- **Front-end parser** loads the binary file, analyzes the machine code syntactically and semantically, then produces the higher-level IR as inner data structures.
- **Universal Decompiler Machine (UDM)** analyzes data- and control flow through a *control flow graph* (CFG) created from the IR and apply transformations on the IR according to analysis. This is part of a decompiler is the most difficult part to implement.
- **Back-end code generator** finally maps the IR into the target high-level language and generates textual source.

A paper about Krakatoa decompiler describes a method of building a CFG, eliminating stack operations and synthesizing `goto` instructions to compilable Java control structures, then transforming that to readable source. It uses *Ramshaw's goto elimination algorithm*, which was originally used to replace `goto`s with loop exit statements in Pascal source code[19]. In Krakatoa, `goto`s are replaced with infinite loops with appropriate `break` and `continue` statements[20]. This results in valid, but convoluted source, on which further "beautifying" gets applied to recover readable source code. Dava decompiler uses a similar approach. Because Dava is built on top of the Soot bytecode optimizing framework, Soot transforms bytecode to multiple intermediate representeations before the resulting source code[21]. Before source code generation, Dava rewrites the AST and applies language idiomatic transformations and code beautification, like writing `i++` instead of `i = i + 1` and using short-circuit logic instead of nested `if` and `else` clauses.

Figure 2.5 shows a design and phases of an abstract Java decompiler based on Cifuentes's decompiler architecture and the techniques used by Krakatoa and Dava.

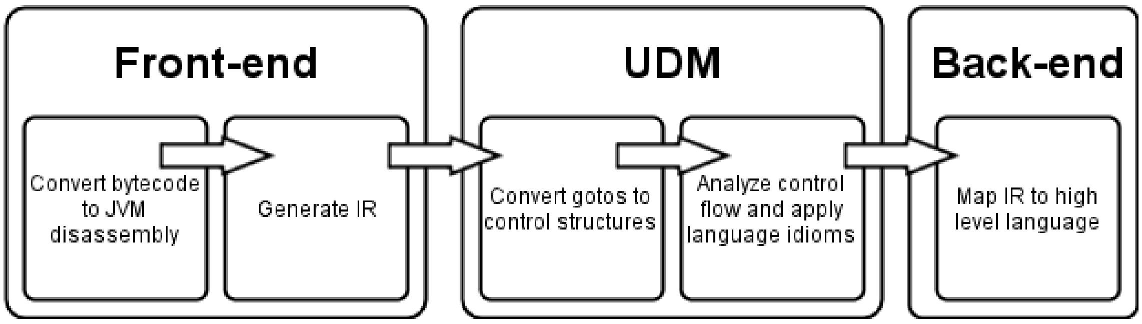

Front-end
Convert bytecode to JVM disassembly
Generate IR
UDM
Convert gotos to control structures
Analyze control flow and apply language idioms
Back-end
Map IR to high level language

Figure 2.5: Phases of an abstract Java bytecode decompiler.

Even if the control flow is recreated, there are still problems with local variables, type inference and synchronized blocks[22]. Translating bytecode instructions to Java source is often times considered harder than regular compiling.

Chapter 3

Overview of existing solutions

3.1 Decompilers

The two types of decompilers are *javac-specific decompilers* and *tool-independent decompilers*. Javac-specific decompilers target bytecode that was made specifically with `javac` and uses compilation patterns related to the compiler. Arbitrary bytecode can be generated by obfuscators, optimizers, aspect-oriented compilers or can be even made by hand. Javac-specific decompilers can be broken by giving them non-standard or obfuscated, yet verifiable bytecode, while tool-independent decompilers can apply analysis and transformation to recreate readable Java source.

The very first public mention of a working decompiler was in Daniel Ford's whitepaper called *Jive: A Java Decompiler*, although the decompiler itself was not made public. The first publicly accessible decompiler was **Mocha** back in 1996. It's release caused controversy and was initially pulled from the public internet. It's still available for download from here: `http://www.brouhaha.com/~eric/software/mocha/`

Java Decompiler or **JAD** for short is a free for non-commercial use, closed source javac-specific decompiler written in C, that was widely used, but it's not maintained anymore. It's implemented to use compiler patterns for earlier JDK versions.

Java Decompiler Project, also known as **JD Project** is compilation of open-source modules written in Java and Groovy. JD-Core is the core library supporting decompilation of Java class files from Java version 5 to version 8, while JD-GUI is the front-end that allows reading JAR and ZIP files and browsing the package hierarchy. It offers plugins for Eclipse and JetBrains IntelliJ IDEA. The project can be found at `http://jd.benow.ca/`

Dava was mentioned before in Section 2.4, it is a tool-independent decompiler developed at McGill University by the Sable Research Group and it's based on the Soot Java bytecode optimizing framework. This decompiler is able to handle obfuscated code and applies transformations on the decompiled code to make it more idiomatic and reflect best practices. The project can be found here:
`http://www.sable.mcgill.ca/dava`

Fernflower is an analytical decompiler engine, which is an integral part of JetBrains IntelliJ IDEA. The engine can be used as a standalone decompiler by building it from source and invoking it from the command line.

```
$ java -jar fernflower.jar -hes=0 -hdc=0 \
<class folder, JAR or ZIP to read> \
<result source folder>
```

It has an extensive customizability with switches. In the above command, `-hes` is a switch for hiding empty super constructor invocation while `-hdc` is switch for hiding empty default constructor. Giving them 0 as value will turn off hiding. There are other switches specified aimed for professional reverse engineers, like decompiling generic signatures, decompiling lambda expressions to anonymous classes and etc. It's also capable of decompiling obfuscated code and renaming obfuscated symbols. It seems the closest of recreating original Java source, as it even gives meaningful names to local variables depending on the context of how they are used in code. The plugin itself bundled with IDEA can be invoked too by specifying the plugin executable JAR in the classpath and running `ConsoleDecompiler` class that holds the `main()` function.

```
$ java \
-cp <IDEA installation>/plugins/java-decompiler/lib/java-decompiler.jar \
org.jetbrains.java.decompiler.main.decompiler.ConsoleDecompiler \
-hes=0 -hdc=0 \
<class folder, JAR or ZIP to read> \
<result source folder>
```

The engine can be found here as part of IntelliJ IDEA Community Edition: `https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine`

3.2 Bytecode libraries

Bytecode libraries are libraries for extracting and manipulating classes in JVM bytecode form. Not only they are capable of reading and analyzing bytecode, but also of generating and dynamically modifying them. Bytecode libraries can be used to create code generators for compilers, implement domain-specific languages (DSL), aspect-oriented programming, reflection and obfuscation.

ASM is the de facto standard bytecode library, created by Eric Bruneton in 2000, during his PhD studies. It's capable of manipulating and analyzing JVM bytecode and aside from modifying or generating existing classes, it also contains common transformation and analysis algorithms. It utilizes `visitors` to traverse class file structure, has a low-level API and focuses more on performance as it's specifically designed for dynamic runtime use[23]. The library is used by Groovy, Kotlin and Scala to emit bytecode. Two plugins that use ASM library are *Bytecode Outline* plugin for Eclipse and *ASM Bytecode Outline* plugin for IntelliJ IDEA. ASM can be accessed from here: `http://asm.ow2.org`.

Apache Commons Byte Code Engineering Library, or **BCEL** for short is another effective bytecode manipulation tool. Loaded binary class files are deserialized into Java objects, as well as fields and methods. It also utilizes visitors for traversals. There's a "generic" part of the library, containing classes to dynamically modify class structure, constant pool and bytecode instructions. Apache Commons BCEL can be accessed from
`http://commons.apache.org/proper/commons-bcel`.

Javassist is a bytecode library aimed to make bytecode analysis and manipulation easier for the user. It contains two levels of API. The higher *source level API* provides functionality like inserting new bytecode instructions as string containing arbitrary Java source, while class files themselves can be edited with the lower *bytecode level API.*

Javassis can be accessed at
`http://jboss-javassist.github.io/javassist/`.

With bytecode libraries, field and method information of a class can be accessed from bytecode, although reverse engineering instructions is still up to the library user. There are external tools and applications, but currently there's no existing library solution for decompiling bytecode back to Java source, except for the integration of Fernflower as library into an application. For the prototype, I have chosen Apache BCEL since I found its API straightforward and its abstraction level is low enough for retrieving type information of compiled classes in JAR files. ASM could have also been a good choice. Apache BCEL is used by FindBugs tool described in the introduction.

Chapter 4

Prototype JAR reader

With the use of Apache Commons BCEL as third-party bytecode library, I was able to implement a prototype application in Java as a proof-of-concept for retrieving code information from compiled Java classes inside JAR files. The program can be used through a simplistic graphical user interface shown in Figure 4.1.

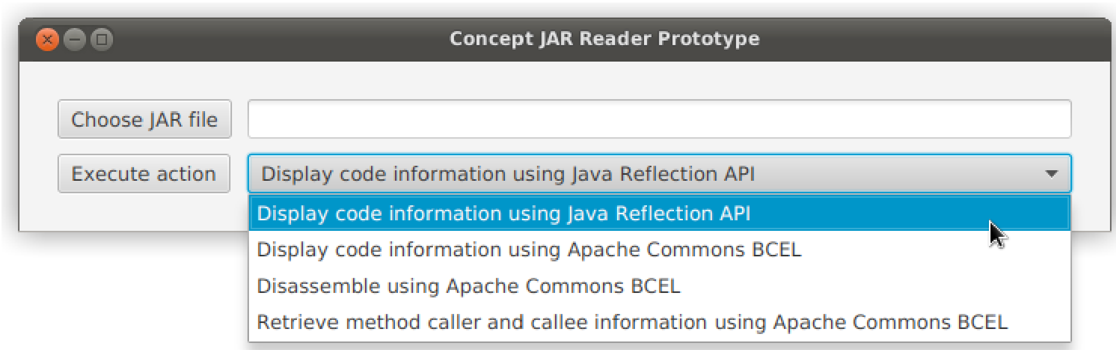

Concept JAR Reader Prototype
Choose JAR file
Execute action
Display code information using Java Reflection API
Display code information using Java Reflection API
Display code information using Apache Commons BCEL
Disassemble using Apache Commons BCEL
Retrieve method caller and callee information using Apache Commons BCEL

Figure 4.1: Graphical user interface for the prototype.

This prototype provides the following functionalities:

- Retrieving type information from classes inside JAR using Java Reflection API.
- Retrieving type information from classes inside JAR the same way, but this time using Apache Commons BCEL.
- Disassembling classes inside JAR with BCEL.
- Recreate method caller and callee listing functionality found in CodeCompass.

The prototype lacks functionalities such as decompiling bytecode instructions to Java source, extracting local variables, handling exception tables, dynamic polymorphism and generic data types. Instructions are instead translated to commented out JVM assembly output, mirroring the functionality of `javap`. This prototype is not supposed to be a real solution or decompiler, it's just for showcasing methods for retrieving class structure information. However, one of it's component was later reused in the integration as can be seen in Chapter 5.

Source code of the prototype, as well as the API documentation generated by `javadoc` utility can be found on the attachment of this thesis. Although CodeCompass is written for compatibility with JDK version 7, in my prototype, I used JDK version 8 and took advantage of functional-style programming. UML class diagram of the prototype is listed in Appendix B.1 on page 72.

4.1 `java.util.jar` package

Luckily, there is no need for third-party provision to handle JAR file traversal, as it's part of the standard Java API in the `java.util.jar` package, which is based on `java.util.zip`. The package provides two classes, `JarFile` to open and read contents of a JAR file, and `JarEntry` representing a file entry in JAR. Each of the classes inherit from their ZIP variants, like `ZipFile` for `JarFile` and `ZipEntry` for `JarEntry`. `JarEntry` objects can still be treated as their ZIP counterparts, like querying PKZIP header information, for example compressed size or CRC-32 checksum. These classes complement further solutions in the prototype, as their responsibility is just to handle iteration of class files.

Listing 4.1 shows example for iterating over all entries in JAR, using `try`-with-resources statement from Java 7. `JarFile.entries()` returns an enumeration of JAR entries, ordered as they appear in the central directory part of JAR file.

Listing 4.1: Read all entries from JAR.

```java
import java.nio.file.Path;
import java.util.Enumeration;
import java.io.IOException;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

public class ExampleJarReader {
  public void readJar(final Path jarPath) {
    final Path absolutePath = jarPath.toAbsolutePath();

    try (final JarFile jar = new JarFile(absolutePath.toFile())) {
      Enumeration<JarEntry> entries = jar.entries();
      while (entries.hasMoreElements()) {
        JarEntry entry = entries.nextElement();
        processEntry(entry);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  void processEntry(final JarEntry entry) {...}
}
```

The above example includes manifest files, directories and resource files besides classes. Listing 4.2 filters only for Java class files and also uses a `for`-based idiom for looping through enumeration.

Listing 4.2: Read only classes from JAR.

```java
import java.nio.file.Path;
import java.util.Enumeration;
import java.io.IOException;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

public class ExampleJarReader {
  public void readClassesInJar(final Path jarPath) {
    final Path absolutePath = jarPath.toAbsolutePath();

    try (final JarFile jar = new JarFile(absolutePath.toFile())) {
      for (Enumeration<JarEntry> entries = jar.entries(); entries.hasMoreElements();) {
        JarEntry entry = entries.nextElement();

        if (!entry.isDirectory() && entry.getName().endsWith(".class")) {
          processClass(entry);
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  void processClass(final JarEntry entry) {...}
}
```

With Java 8, entries can be retrieved as an ordered stream and can be used with functional-style programming, filtering out non-classes and mapping processing function over the entries. Listing 4.3 shows an example of this.

Listing 4.3: Class reading with Java 8.

```java
import java.nio.file.Path;
import java.util.Enumeration;
import java.io.IOException;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

public class ExampleJarReader {
  public void readClassesInJar(final Path jarPath) {
    final Path absolutePath = jarPath.toAbsolutePath();

    try (final JarFile jar = new JarFile(absolutePath.toFile())) {
      jar.stream()
         .filter(entry -> !entry.isDirectory() && entry.getName().endsWith(".class"))
         .forEach(this::processClass);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  void processClass(final JarEntry entry) {...}
}
```

4.2 Retrieving class information using Java Reflection API

Reflection is a general term among programming languages meaning the ability of a program to retrieve and manipulate its types during runtime. Java's reflection feature can be used to get type information about a class like its name, super class, interfaces, fields and methods as described in Section 2.2.1 and even modify them while the program is running. The JVM doesn't know anything about compiled classes at the start of execution until they are loaded by the *class loader*, but after that, the programmer has access to the internals of them by using Java's built-in Reflection API.

Although it's similar to the Run-time Type Information (RTTI) used in C++, the latter is limited as it is only used for determining types of classes and executing dynamic casts. Moreover, RTTI can only be used for polymorphic classes[1]. Reflection can only be used on reference types though in Java, not on primitives.

We can use reflection for retrieving internal information from classes inside a JAR file by loading each class with the ClassLoader, instantiate `Class<?>` objects from them and calling their getter methods about class properties. Listing 4.4 shows the beginning of a class reading method that receives the path of JAR file and a class entry from it. The ".class" file extension has to be stripped and file separators in package folder hierarchy have to be swapped to "." symbols in order to get the canonical class name. The canonical name consist of the name of the class and the package name separated by "." symbols as defined by the *Java Language Specification*[24].

Listing 4.4: Stripping ".class" extension and replacing folder path with package hierarchy to get canonical Java class name.

```java
private static String readClass(final Path jarPath, final JarEntry entry)
  throws NoClassDefFoundError, ClassNotFoundException {

  // Get class name and cut off ".class" file extension
  final int CLASS_EXTENSION_LENGTH = 6;
  final String canonicalClassName =
        entry.getName()
             .substring(0, entry.getName().length() - CLASS_EXTENSION_LENGTH)
             .replace('/', '.');

    ...
}
```

`URLClassLoader` is a class loader that can accept a search path URL referring to either a JAR file or a directory containing classes. The protocol part of the URL must start with "jar:file" and end the JAR file name with a "!" symbol. After the "!" symbol, JAR can be used just like a regular directory. By specifying only the root inside the JAR file, the class loader is able to resolve the package folders just from the given canonical class name. This can be seen in Listing 4.5.

[1]Although in a sense, classes in Java are polymorphic by default because of inheritance from `java.lang.Object` as described in Section 2.3.2.

Listing 4.5: Retrieve class from JAR file with `URLClassLoader`.

```java
// Get class
final URL[] urls = {new URL("jar:file:" + jarPath + "!/")};
final URLClassLoader classLoader = URLClassLoader.newInstance(urls);
final Class<?> clazz = classLoader.loadClass(canonicalClassName);
```

After loading the class during runtime as an object containing type information, these properties can be queried with simple getters as shown in Listing 4.6.

Listing 4.6: Retrieve type information and class members with Reflection API.

```java
// Get class information using reflection
final Package classPackage = clazz.getPackage();
final Class<?> superClass = clazz.getSuperclass();
final Class<?>[] interfaces = clazz.getInterfaces();
final Class<?>[] innerClasses = clazz.getDeclaredClasses();

final Field[] fields = clazz.getDeclaredFields();
final Constructor<?>[] constructors = clazz.getDeclaredConstructors();
final Method[] methods = clazz.getDeclaredMethods();
```

When retrieving fields or methods, one needs to be aware that there are two variants for each getter. Plain `getFields()` and `getMethods()` only return members that have public visibility, while `getDeclaredFields()` and `getDeclaredMethods()` return all declared members, including private and protected ones. Also, `getMethods()` will include methods of `java.lang.Object` too, which would only clutter our output.

After retrieving reflected types, their properties can be further queried. `Method.getName()` only returns the simple name of the method. *Fully qualified* field or method names keep track of their declaring class and the package hierarchy that class belongs to, therefore simple names should be concatenated with canonical class name in order to achieve that.

For handling access modifiers, the API contains `java.lang.reflect.Modifier`[2]

[2]Eclipse JDT also has it's own `org.eclipse.jdt.core.dom.Modifier` class with similar functionality, but uses different bytes for accessor flags. The Reflection API and Eclipse JDT variant should not be interchanged because of this.

utility class which provides functionality to decode access modifiers bitmask to their name, even if the bits represents multiple modifier keywords. It can be used like `Modifier.toString(field.getModifiers())` and `Modifier.toString(method.getModifiers())`.

Reflection API cannot retrieve bytecode instructions or `Code` attribute of a method. Primary use of reflection in Java is not to reverse engineer class files, but to inspect them during runtime and query annotations. It has a performance overhead because it resolves types dynamically. Furthermore, it's not usable for our case, because the collected field and method elements returned in an array are not sorted in the order they are declared, nor in any particular order. This is stated in the API documentation for `java.lang.Class<T>`[25] too. It breaks functionality when recovering method caller and callee information, where I want to keep the order of method declarations. Reflection is a naive, but good enough solution for retrieving basic code and class structure information. For things like bytecode analysis, disassembly and decompilation however, it lacks functionality.

4.3 Visitors in BCEL

Classes and class components in BCEL are represented as data structures just like in Reflection API with the difference that instead of the class loader, BCEL's wrapper class, `ClassParser` loads the class file itself, parses the bytes and returns the BCEL representation. This representation of a class mirrors the Java class file structure of the JVM specification as described in Section 2.2.1, complete with major and minor version numbers, constant pool, attributes and bytecode itself. This can be seen in Listing 4.7. These structures are more suited for examining bytecode information. Some parts of BCEL's API is reminiscent of Reflection API if we look at `Field` and `Method` classes, but BCEL's classes contain more internal information about them.

Listing 4.7: Loading class with BCEL, retrieving class structure elements and saving bytecode as array of bytes at the end.

```java
import org.apache.bcel.classfile.ClassParser;
import org.apache.bcel.classfile.JavaClass;
import org.apache.bcel.classfile.Field;
import org.apache.bcel.classfile.Method;
import org.apache.bcel.classfile.ConstantPool;
import org.apache.bcel.classfile.Attribute;

...

try {
  // Load class
  ClassParser parser = new ClassParser(classFilePath, className);
  JavaClass javaClass = parser.parse();

  // Class package, inheritance and interface information
  String packageName = javaClass.getPackageName();
  JavaClass superClass = javaClass.getSuperclass();
  JavaClass[] interfaces = javaClass.getInterfaces();

  // Class members represented as BCEL objects too
  Field[] fields = javaClass.getFields();
  Method[] methods = javaClass.getMethods();

  // Meta information of class
  int majorVersion = javaClass.getMajor();
  int minorVersion = javaClass.getMinor();
  String originalSourceFile = javaClass.getSourceFileName();

  // Constan pool and class attributes
  ConstantPool constantPool = javaClass.getConstantPool();
  Attribute[] classAttributes = javaClass.getAttributes();

  // Original bytecode itself
  byte[] bytecode = javaClass.getSource();
} catch (IOException e) {
  e.printStackTrace();
}
```

Parts of the class code information in JAR can be represented as an abstract traversable tree where root is defined as the JAR file itself, contained class files are subtrees, and nodes of the class subtrees represent fields and methods. Figure 4.2 shows such representation.

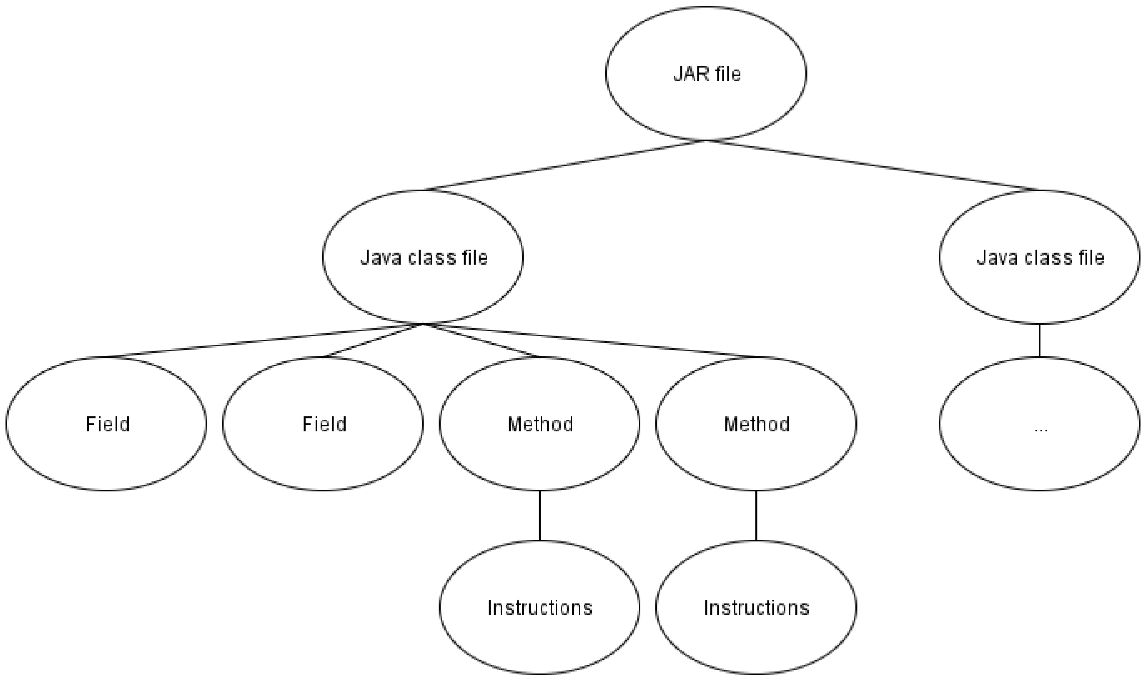


Figure 4.2: Tree representation of a traversable JAR file and structure of its classes.

Using visitor design pattern[26], a tree-like data structure like this can be walked node-by-node with a visitor object, the nodes can accept this visitor, and node-specific actions can be executed. This is a customizable and reusable approach as we can define visitors with different behaviours and choose the appropriate kind of visitor for the task or even swap them in runtime.

The pattern is used by most language-related applications like interpreters and compilers for applying visitors on parse trees. Most decompilers and bytecode libraries also use visitor-based traversals. Eclipse JDT even has it's own `ASTVisitor` abstract visitor, that walks the a Document Object Model (DOM) generated from Java source. A realization of `ASTVisitor` can be found in the source parser of CodeCompass.

BCEL has it's own `Visitor` interface which defines abstract methods that accepts BCEL node references as seen in Listing 4.8. Although to avoid obligatorily implementing actions for all existing BCEL node types when using this interface, it's better to extend `EmptyVisitor` instead to optionally implement actions for nodes we use.

Listing 4.8: Abstract methods of `Visitor` interface in BCEL, only nodes important for the task are listed. Source is from Visitor.java file in codebase of BCEL.

```java
public interface Visitor {
  ...

  void visitJavaClass(JavaClass obj);

  void visitField(Field obj);

  void visitMethod(Method obj);

  ...
}
```

Each BCEL structure implements the `Node` interface, which contains the `accept()` method. That way a node can call the visitor on itself using double-dispatch.

Listing 4.9: `accept()` method of `JavaClass`, implementing `Node` interface. Source is from JavaClass.java file in codebase of BCEL.

```java
public class JavaClass implements Node {
  ...

  @Override
    public void accept( final Visitor v ) {
    v.visitJavaClass(this);
  }

  ...
}
```

With the use of this pattern, I was able to define an abstract visitor called `JarVisitor` seen in Listing A.5, where I made `JarFile` and `JarEntry` as nodes. This visitor contains a default JAR reading operation, so concrete visitor classes specializing from this don't have to reimplement it, keeping the Don't Repeat Yourself (DRY) principle. Although it can be used by calling `visitJarFile()` and prividing a `JarFile` as parameter, I defined a constructor that accepts the relative or absolute path of JAR file and I also added a `start()` method, which creates the `JarFile` object and starts traversing it. Part of the abstract JAR visitor code can be found in Appendix A on page 68 as Listing A.5.

The implemented visitors in the JAR reader prototype are the following:

- `CodeInfoVisitor` implements the code information listing behavior as seen with the Reflection API example, but uses BCEL's visitor instead. Since BCEL can read the whole structure of a class, this visitor also retrieves major and minor versions, name of source file and the constant pool for each class.

- `DisassembleVisitor`[3], that generates Java code for each visited node of the tree. Upon visiting the method nodes, method signature is generated with commented out JVM assembly code enclosed between braces.

- `MethodCallInfoVisitor`, which upon visiting the method nodes of each class, adds them into a collection and connects methods with their callers and callees. It doesn't generate a true function callgraph, however, the simple caller-callee information about each method is sufficient for mimicking the function caller-callee retrieval in CodeCompass.

4.4 "Generic" BCEL

A naive way to print JVM assembly instructions in `DisassembleVisitor` and get method invocations in `MethodCallInfoVisitor` would be to get the `Code` attribute of each method as `org.apache.bcel.classfile.Code` object, use `Code.getCode()` to retrieve the real bytes and manually check the opcode bytes. Although `Code` has a `toString()` method implemented which prints out this attribute like with `javap`, it also prints out meta data like number of local variables, maximum stack size and exception table. Instead of handling code as string[4], I wanted more control over the instructions with BCEL. Furthermore, I wanted to keep the constant pool of each method in `MethodCallInfoVisitor` in order to retrieve invoked method symbols.

[3]The design decision was made by me to not call it DecompileVisitor, as this visitor recreates the functionality of `javap` and doesn't apply any decompilation of instruction code. See Section 2.4 for more details.

[4]In `DissassembleVisitor`, there's a `visitCode()` method using this approach, which was marked `@Depracated` in favor to use `visitInstructions()` instead.

Enter `org.apache.bcel.generic` package, which contains the "generic" part of BCEL, meaning as quoted from the official *Apache Commons BCEL Manual*[27]:

> *"It makes the static constraints of Java class files like the hard-coded byte code addresses "generic"."*

This package has classes for dynamically modifying class objects, manually synthesizing new Java types from BCEL nodes and providing a finer control over bytecode instructions. Each BCEL node has its own generic counterpart, like `ClassGen` for `JavaClass`, `ConstantPoolGen` for `ConstantPool`, `FieldGen` for `Field`, `MethodGen` for `Method` and so on. Listing A.6 shows a way to manually create a class with the use of "generic" classes in Appendix A on page 69. There's even a `BCELifier` visitor in BCEL that recreates the visited Java class as if the bytecode was written by hand with "generic" BCEL.

The package has a certain class called `InstructionList`, which is a container holding a method's bytecode instructions as abstract `Instruction` classes. To access a method's `InstructionList`, the method has to be converted into `MethodGen` first by supplying the method itself, name of the method's class and a `ConstantPoolGen` created from the constant pool of method's class.

Listing 4.10: Retrieving `InstructionList` of a method by creating `MethodGen` from `Method`.

```java
@Override
public void visitMethod(final Method method) {
  MethodGen methodG = new MethodGen(method,
                                    visitedClass.getClassName(),
                                    new ConstantPoolGen(visitedClass.getConstantPool()));

  InstructionList instructions = methodG.getInstructionList();
  visitInstructionList(instructions);
  ...
}
```

Instructions then can be iterated with `InstructionHandle`, which acts as an iterator. It's advised to use it, because `InstructionList` is implemented as a doubly-linked list with `InstructionHandle`s being the doubly-linked list nodes. Listing 4.11 shows part of `DisassembleVisitor`, where I iterate over the instructions with `InstructionHandle`, check for constant pool reference at each instruction and form a disassembly string.

Listing 4.11: Snippet from `DisassembleVisitor`.

```java
@Override
public void visitInstructionList(final InstructionList instructions) {
  // Initial instruction iterator
  InstructionHandle ihandle = instructions.getStart();

  // Iterate over instructions
  while (ihandle != null) {
    Instruction instruction = ihandle.getInstruction();

    // If instruction uses index to the constant pool, get the
    // name of the symbolic constant. Otherwise leave it empty.
    String constantPoolReference = "";
    if (instruction instanceof CPInstruction) {
      int poolIndex = ((CPInstruction) instruction).getIndex();
      Constant constant = constantPool.getConstant(poolIndex);
      constantPoolReference = constantPool.constantToString(constant);
    }

    // Return disassembled output in
    // "// [instruction index] [instruction name] [optional constant pool reference]" form
    String disassemblyOutput = String.format("\t\t// %1$-4d: %2$-20s %3$s\n",
                                             ihandle.getPosition(),
                                             instruction.getName(),
                                             constantPoolReference);

    // <print handling code>

    // Move iterator
    ihandle = ihandle.getNext();
  }
}
```

Listing 4.12 shows part of `MethodCallInfoVisitor`, where I examine using `switch` with fallthrough if the instruction's opcode is a method invocation and retrieve the name of the invoked method from the constant pool. The reason why `invokedynamic` is omitted is because it doesn't used in Java language, but just to support JVM languages using dynamic typing, as mentioned in Section 2.3.2.

Listing 4.12: Snippet from `MethodCallInfoVisitor`.

```java
...

InstructionHandle ihandle = method.getInstructionList().getStart();
while (ihandle != null) {
  Instruction instruction = ihandle.getInstruction();
  switch (instruction.getOpcode()) {
    case Const.INVOKEINTERFACE:
    case Const.INVOKESPECIAL:
    case Const.INVOKESTATIC:
    case Const.INVOKEVIRTUAL:
      // Retrieve called method name
      InvokeInstruction invokeInstruction = (InvokeInstruction) instruction;
      String calleeName = invokeInstruction.getReferenceType(methodConstantPool) + "." +
                          invokeInstruction.getMethodName(methodConstantPool);

    // <caller and callee information handling code>
...
```

Chapter 5

Integration with CodeCompass

5.1 Original parsing process

CodeCompass is composed of two main parts, the parser and the web server. To use the tool, the user has to issue the following actions:

1. Start the parser executable on software project's codebase that is to be parsed. On the command line, name of the software project, build commands used to compile the project and connection arguments of a running relational database have to be given. Build commands are recognized relatively to the path where the parser is invoked. Furthermore, a working directory (workdir) is also needed to be specified where the parser writes logs and configuration files.

   ```
   CodeCompass parse -w <location of workdir> \
   -n <project name> \
   -b "<compilation commands>" \
   --dbhost <database host>
   --dbport <database port number>
   --dbuser <database user name> \
   --dbpass <database password>
   ```

2. After the parsing is successfully completed, the server can be executed by giving it a port number to be started on, the location of the workdir and the relational database connection arguments as before.

```
CodeCompass server -w <location of workdir> \
-p <port number for CodeCompass server> \
--dbhost <database host>
--dbport <database port number>
--dbuser <database user name> \
--dbpass <database password>
```

This separation can be seen in Figure 5.1, where parser stores information in the relational database and server retrieves them through a service layer.

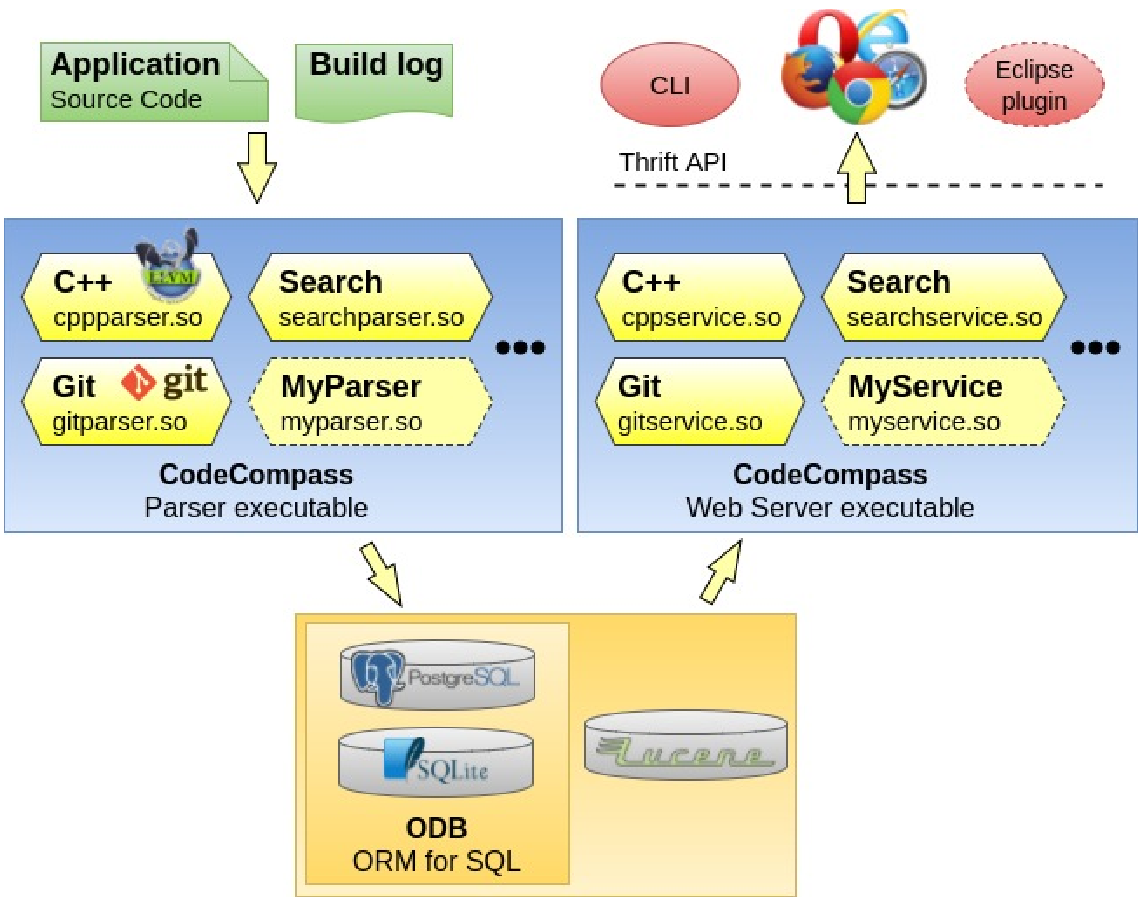


Figure 5.1: Overview of the architecture of CodeCompass. Apache Lucene™is used by search parser as a text search engine library. Because of the open API of Apache Thrift, presentation layer is not restricted to just web front-end, but CodeCompass can be extended to be used on command line or in Eclipse. Diagram is provided by Tibor Brunner.

The parser executable first builds a *compilation database* as a build log, which is a JSON file consisting of entries about translation units. Each entry contains the translation unit's main source file and compilation commands with arguments for it. One tool that is capable of generating compilation databases is CMake with the CMAKE_EXPORT_COMPILE_COMMANDS variable. However, CMake's compilation database generation only works for C/C++ projects. CodeCompass uses a separate build logger called ld logger, which is used in the CodeChecker project and is able to build compilation database from Java sources too.

After writing the build log, parser creates a new relational database with the name of the project, reads entries from the JSON file and starts the appropriate parsing process for each translation unit. One parser process runs for one translation unit and processes are provided by dynamically linked libraries. For example, cppparser.so realizes the C/C++ parsing, which with the help of LibTooling and LibASTMatchers of Clang, traverses the AST of the C/C++ source file and saves the parse results into the running relational database. Git parser process by gitparser.so is used to parse Git-related information to display git blame output on the web front-end. Other parser processes are also available, like parsing for search and metrics.

Supported relational databases are PostgreSQL and SQLite. Persistence of parse information is done with Object-Relational Mapping (ORM) frameworks, ODB is used for C/C++ and EclipseLink JPA for Java. Each AST node is represented as an entity, a table in the relational database. CodeCompass even stores the line position of AST node entities to be recreated on the web front-end. Categories of data in the relational database are the following:

- Project and file-related entities
- Build actions with source, target and logs
- C/C++-related AST nodes as entities
- Java-related AST nodes as entities

To undestand the relationships between database entities, refer to the relational model diagrams of a sample project parsed by CodeCompass in Appendix B on page 74.

For commmunication, CodeCompass uses services implemented with Apache Thrift. Thrift is an interface definition language for building Remote Procedure Call (RPC) services[28], similarly as CORBA. Thrift has a compiler that is able to generate

C++ header- and implementation files, Java source files, and source code for several other languages from .thrift files containing the interface definition. The two kinds of services are parser services, which provide an interface between parser processes and parsers themselves, and services that are separated from the parser in the web server executable. The latter provide a logical layer between the database and web front-end, as their responsibility is to read stored parse information from database, apply transformations and forward it to the GUI. For example, inheritance- or function callgraph diagrams are not stored in database, but the service layer assembles them from parse information.

Original Java source parsing processes for each source file were programmed in Java and started by `javaparserprocess.so` binary, which calls them like they were called from the command line with `java` command. During development, I refactored this module and ordered the original classes into appropriate package hierarchies. For the UML class diagram of this structure, refer to Figure B.3 in Appendix B on page 74. The entry point of Java parsing is `JavaParserServiceImpl`, which implements a Thrift interface. This interface delegates parsing tasks to `Parser`, which initiates relational database connection, verifies build log and applies a visitor on the AST called `AstVisitor`. This visitor extends the abstract visitor class from Eclipse JDT, and it's responsibility is to create an entity for each AST node and persist them into relational database.

5.2 Implementation

Because server-side services are capable of reading parsed information from the database and functions like displaying info tree or querying method caller and callee information are implemented on the server side, if the database is populated the right way, there's no need to touch the web front-end or the service part of Code-Compass. The only thing that has to be done from my side is reading the JAR file, iterating over classes, parsing compiled bytecode and storing the AST elements into the database, just like with regular Java source parsing.

Fortunately, when logging `javac` build actions, the build log not only contains the source files and the compilation command, but compile arguments like the specified classpath too. For a Java source that links with classes in a JAR file, the classpath containing the JAR filename must be specified with `-classpath` or `-cp` arguments. The generated compilation database would look like Listing 5.1. This classpath can be recognized by the Java parser and parsing actions can be defined for the JAR file contained in classpath.

Listing 5.1: Compilation database containing directory of compilation, issued build command and source file used for compilation. Command includes specifying JAR file for classpath.

```
[
{
"directory": "/home/balintkiss/cc/projects/hello_jar",
"command": "javac -cp /home/balintkiss/cc/projects/hello_jar/classinjar.jar:\
/home/balintkiss/cc/projects/hello_jar \
-sourcepath /home/balintkiss/cc/projects/hello_jar \
/home/balintkiss/cc/projects/hello_jar/Main.java",
"file": "/home/balintkiss/cc/projects/hello_jar/Main.java"
}
]
```

First thing I had to accomplish is to recognize a JAR files in the classpath and persist them as folder entities with ".jar" suffix, as seen in Figure 5.2 with one JAR file. This way it can be opened in the file explorer of the web front-end and further package hierarchies and class files could be stored there.

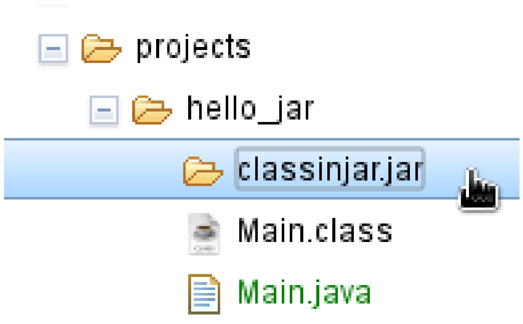


Figure 5.2: Displaying JAR file as a folder in file explorer of CodeCompass web front-end next to Main.java and Main.class. Currently it's empty.

Classes of the original Java parser architecture had very tight coupling. Design was not intended to be extended with .class file and bytecode parsing, as it heavily relies on information found in textual format with Eclipse JDT. The solution was to first disassemble source from class as text, then use the existing source AST visitor on it. Instead of both disassemble classes and persist AST nodes in one class, I was able to verbatim reuse `DisassembleVisitor` from the prototype to generate disassembled Java source code, then give this input for the existing AST visitor class.

The original parsing pipeline is visualized in Figure 5.3.

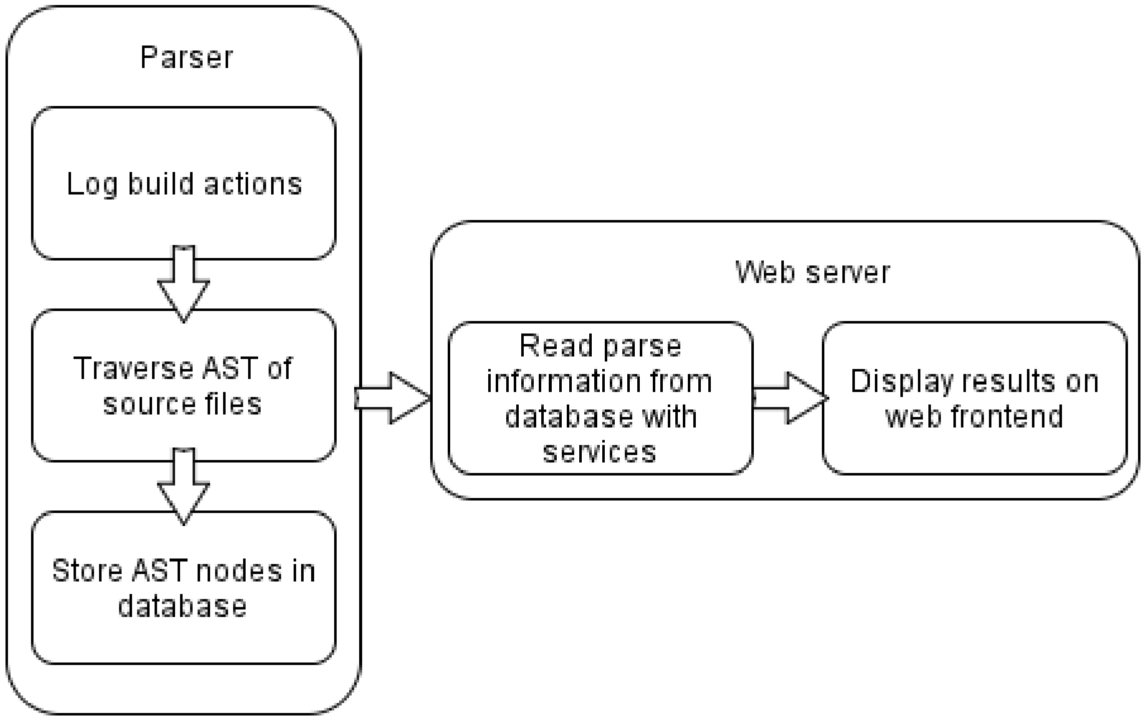

Parser
Log build actions
Traverse AST of source files
Store AST nodes in database
Web server
Read parse information from database with services
Display results on web frontend

Figure 5.3: CodeCompass Java parsing process pipeline.

Now with my modifications, disassembling classes would come after creating and reading build log. Then the pipeline continues on the disassembled classes in Figure 5.4 the same way, as in Figure 5.3.

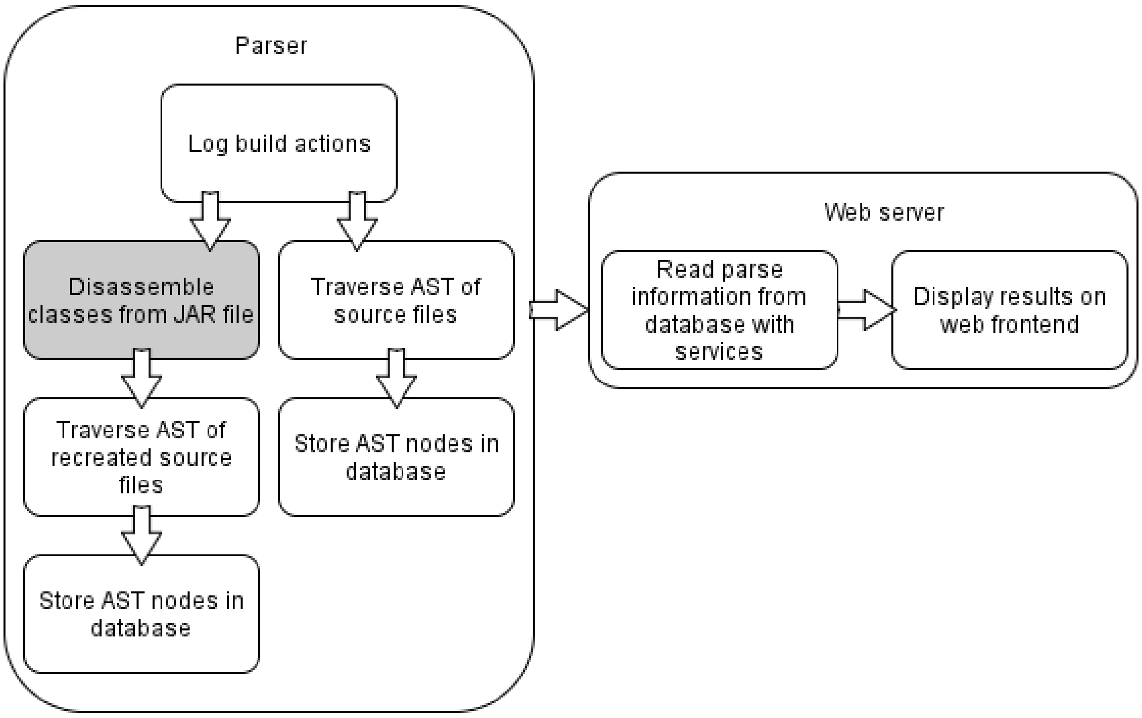

Parser
Log build actions
Disassemble classes from JAR file
Traverse AST of source files
Traverse AST of recreated source files
Store AST nodes in database
Store AST nodes in database
Web server
Read parse information from database with services
Display results on web frontend

Figure 5.4: How disassembling classes from JAR files would fit into Java parsing process pipeline.

The JAR parsing module is implemented as a package containing `JarParser` and `DisassembleVisitor` classes. Recognizing JAR files in classpath and calling `JarParser` for each one is done by `Parser` class from the original Java parser. Responsibility of `JarParser` is opening the JAR file and iterating over class files, while coordinating disassembling with `DisassembleVisitor` and traversing AST with `AstVisitor`. Figure 5.5 shows how the new JAR parsing module interacts with existing classes.

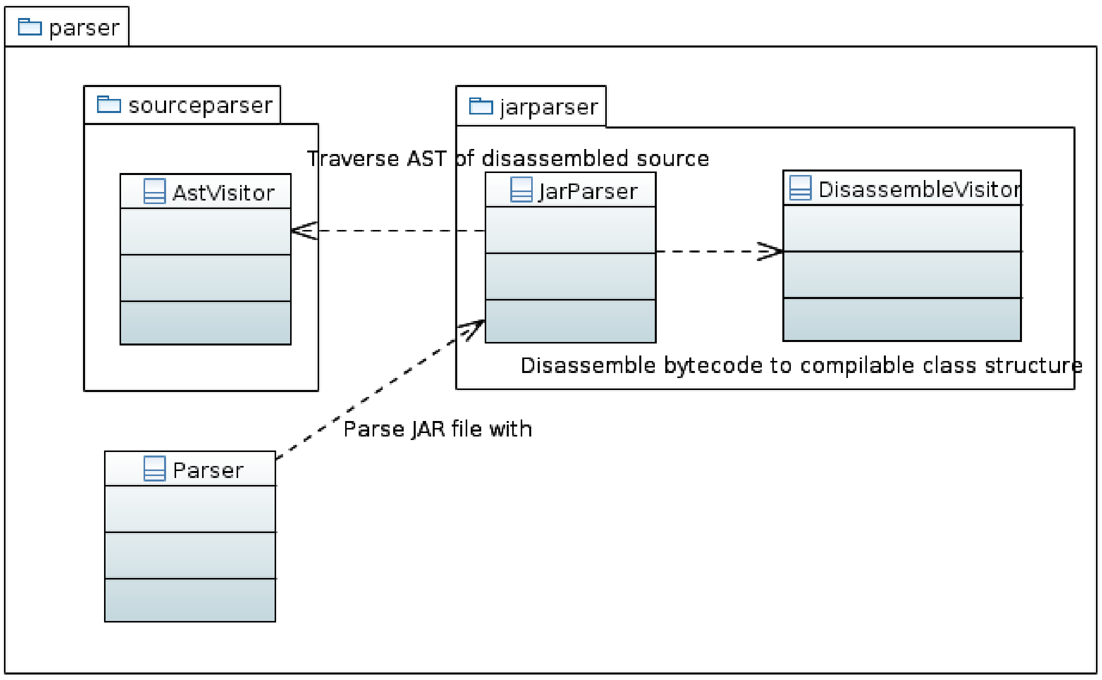


Figure 5.5: UML class diagram of JAR parsing module I made and how it fits into Java parser of CodeCompass.

5.3 Initial results

With my integration, I managed to parse a JAR file containing one class, as can be seen in Figure 5.6. The class file got successfully persisted into database while keeping the package hierarchy as folder structure.

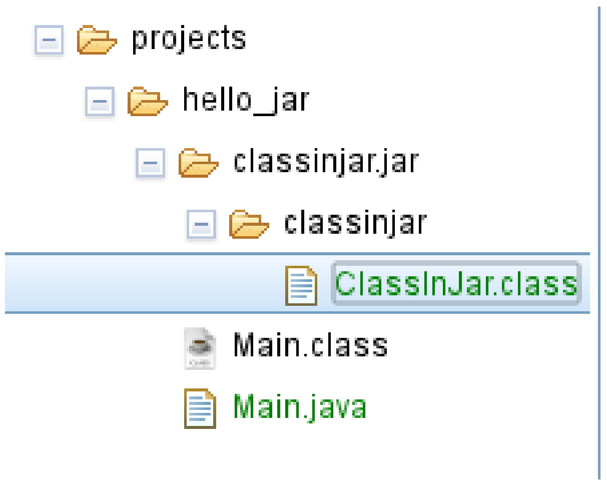
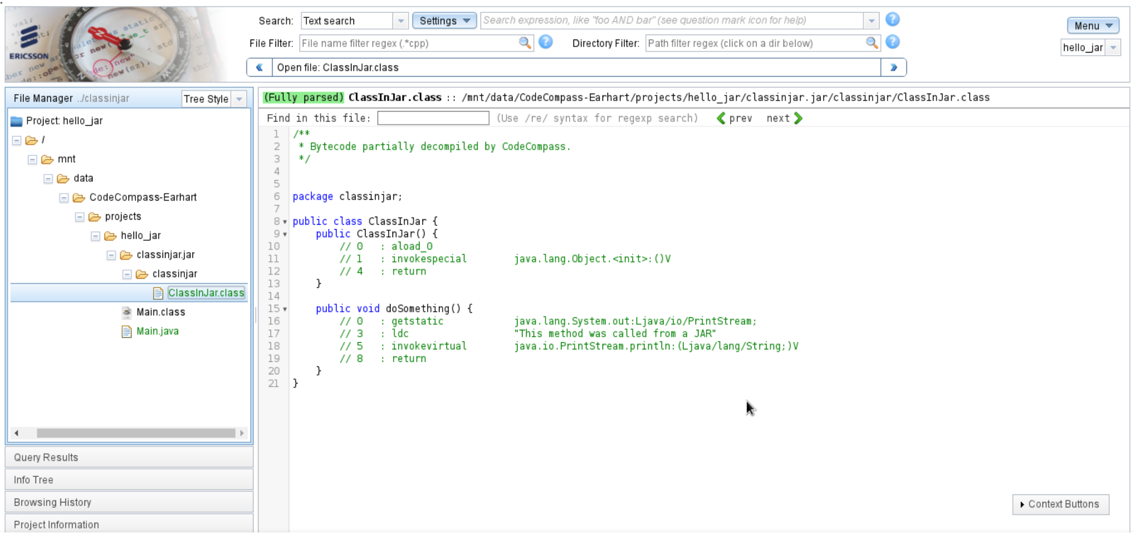


Figure 5.6: Displaying content of JAR file with stored package hierarchy in file explorer of CodeCompass web front-end.

Figure 5.7 shows output generated by `DisassembleVisitor` in front-end of CodeCompass.



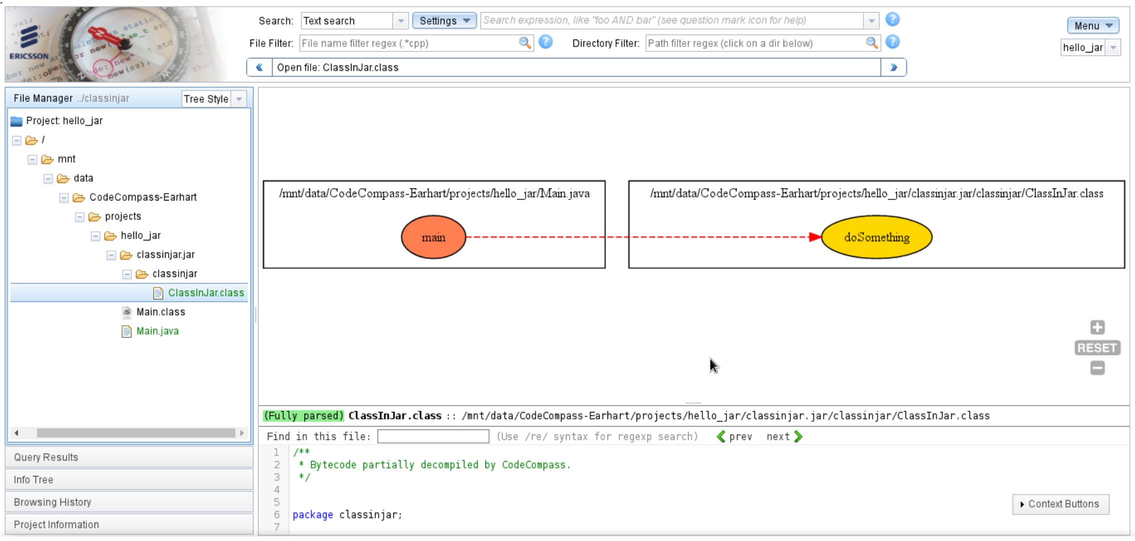
(Fully parsed) **ClassInJar.class** :: /mnt/data/CodeCompass-Earhart/projects/hello_jar/classinjar.jar/classinjar/ClassInJar.class

Find in this file: (Use /re/ syntax for regexp search) prev next

```
/**
 * Bytecode partially decompiled by CodeCompass.
 */


package classinjar;

public class ClassInJar {
    public ClassInJar() {
        // 0   : aload_0
        // 1   : invokespecial        java.lang.Object.<init>:()V
        // 4   : return
    }

    public void doSomething() {
        // 0   : getstatic            java.lang.System.out:Ljava/io/PrintStream;
        // 3   : ldc                  "This method was called from a JAR"
        // 5   : invokevirtual        java.io.PrintStream.println:(Ljava/lang/String;)V
        // 8   : return
    }
}
```

Figure 5.7: Successfully parsed class from JAR file, showing it's disassembled code in the source display.

Figure 5.8 illustrates that callgraph can be displayed between methods from source files and methods from compiled class files contained in JAR file.



Search: Text search
Settings
Search expression, like "foo AND bar" (see question mark icon for help)
Menu
File Filter: File name filter regex (.*cpp)
Directory Filter: Path filter regex (click on a dir below)
hello_jar
Open file: ClassInJar.class
File Manager ./classinjar
Tree Style
Project: hello_jar
/
mnt
data
CodeCompass-Earhart
projects
hello_jar
classinjar.jar
classinjar
ClassInJar.class
Main.class
Main.java
/mnt/data/CodeCompass-Earhart/projects/hello_jar/Main.java
main
/mnt/data/CodeCompass-Earhart/projects/hello_jar/classinjar.jar/classinjar/ClassInJar.class
doSomething
RESET
Query Results
Info Tree
Browsing History
Project Information
Context Buttons

(Fully parsed) **ClassInJar.class** :: /mnt/data/CodeCompass-Earhart/projects/hello_jar/classinjar.jar/classinjar/ClassInJar.class

Find in this file: (Use /re/ syntax for regexp search) prev next

```
/**
 * Bytecode partially decompiled by CodeCompass.
 */


package classinjar;
```

Figure 5.8: Call information between `main()` method from `Main.java` source file and `doSomething()` method from `ClassInJar.class`.

Chapter 6

Conclusion

To have more undestanding about the problem domain, I learned about the structure of JAR and Java class files and studied the inner workings of Java bytecode. I looked at decompilation techniques and searched for existing solutions to aid my task. With the use of a third-party bytecode library, I managed to create a standalone prototype application, which is able to traverse the structure of compiled Java classes contained in JAR file and recover it as Java source code. Finally, I integrated part of this prototype into the Java parsing process of CodeCompass and demonstrated the initial parsing results. While recreating class structure and type information is easy, decompiling instruction code is not a trivial task and it's a wide field of research. Even the solution I made doesn't do real decompiliation, but disassembly.

At the time of writing this thesis, the open-source reference implementation of the Java SE 9 Platform is about to be released for production use in the following months[29]. One of it's features is the inclusion of Project JigSaw, which is a higher-level module system to resolve JAR hell. JDK version 9 would also include `jlink`, a linker tool for this module system and the experimental switch from JIT to AOT compilation.

In CodeCompass, the disassembling visitor of JAR reader module can be changed anytime into a decompiling visitor using the techniques from Section 2.4. It could be useful to generate an intermediate AST from source file and bytecode to loosen the coupling of Java source- and JAR parser. In hindsight, using Fernflower decompiler engine as third-party library would have been a simpler and time-sparing solution and also a more complete one, but utilizing a bytecode library was a good learning experience. It is also not far-fetched to have the possibility of retrieving symbolic information or even callgraphs from binary objects and libraries compiled from C/C++.

For anyone who is interested in starting out with reverse engineering applications, Java is a good choice, as symbolic information is easy to retrieve from class file structure and decompiling bytecode would yield more initial success than decompiling binary machine code.

References

[1] *JAR File Specification.* URL: `https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html` (visited on 04/13/2017).

[2] David Hovemeyer and William Pugh. “Finding Bugs is Easy”. In: *ACM SIGPLAN Notices* Volume 39 Issue 12 (2004), pp. 92–106.

[3] *.ZIP File Format Specification, Version 6.3.4, PKWARE Inc.* URL: `https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT` (visited on 04/13/2017).

[4] P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3.* Internet Engineering Task Force. 1996. URL: `http://www.ietf.org/rfc/rfc1951.pdf` (visited on 04/13/2017).

[5] Florian Bucholz. *The structure of a PKZip file.* URL: `https://users.cs.jmu.edu/buchhofp/forensics/formats/pkzip.html` (visited on 04/13/2017).

[6] Tim Lindholm et al. *The Java Virtual Machine Specification.* Java SE 8. Oracle America Inc. 2015. Chap. 4: The class file format, pp. 70–74.

[7] *javap, Java Platform, Standard Edition Tools Reference.* URL: `https://docs.oracle.com/javase/8/docs/technotes/tools/unix/javap.html` (visited on 04/13/2017).

[8] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages.* 1st. Pragmatic Bookshelf, 2009. Chap. 10: Building Bytecode Interpreters, pp. 237–238. ISBN: 978-1-93435-645-6.

[9] *The Java HotSpot Performance Engine Architecture.* URL: `http://www.oracle.com/technetwork/java/whitepaper-135217.html` (visited on 04/14/2017).

[10] Vladimir Kozlov. *JEP 295: Ahead-of-Time Compilation.* URL: `http://openjdk.java.net/jeps/295` (visited on 04/14/2017).

[11] Steven Pemberton and Martin Daniels. *Pascal Implementation: The P4 Compiler and Interpreter.* Ellis Horwood, 1982. Chap. 10: The P-code Machine. ISBN: 0-13-653-0311. URL: `http://homepages.cwi.nl/~steven/pascal/book/10pcode.html` (visited on 05/06/2017).

[12] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. "The Implementation of Lua 5.0". In: *J. UCS* 11 (2005), pp. 1159–1176.

[13] David Gregg et al. "The Case for Virtual Register Machines". In: *IVME '03 Proceedings of the 2003 workshop on Interpreters, Virtual Machines and Emulators* (2003), pp. 41–49.

[14] Yunhe Shi et al. "Virtual machine showdown: Stack versus registers". In: *ACM Transactions on Architecture and Code Optimization (TACO)* Volume 4 Issue 4 (2008). Article No. 2, pp. 153–163.

[15] Dan Bornstein. *Dalvik VM Internals.* 2008. URL: `https://sites.google.com/site/io/dalvik-vm-internals` (visited on 04/13/2017).

[16] Sohail Khan et al. *Analysis of Dalvik Virtual Machine and Class Path Library.* Tech. rep. Security Engineering Research Group, Institute of Management Sciences, 2009.

[17] Apache Commons BCEL. *The Java Virtual Machine.* 2016. URL: `https://commons.apache.org/proper/commons-bcel/manual/jvm.html` (visited on 04/13/2017).

[18] Cristina Cifuentes. "Reverse Compilation Techniques". PhD thesis. QUEENSLAND UNIVERSITY OF TECHNOLOGY, 1994.

[19] Lyle Ramshaw. "Eliminating go to's while preserving program structure". In: *Journal of the ACM (JACM)* Volume 35 Issue 4 (1988). Article No. 2, pp. 893–920.

[20] Todd A. Proebsting and Scott A. Watterson. "Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?)" In: *COOTS'97 Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)* Volume 3 (1997), pp. 14–14.

[21] Nomar A. Naeem and Laurie Hendren. *Programmer-friendly Decompiled Java.* Tech. rep. McGill University, School of Computer Science, Sable Research Group, 2006.

[22] Jerome Miecznikowski and Laurie J. Hendren. “Decompiling Java Bytecode: Problems, Traps and Pitfalls”. In: *CC '02 Proceedings of the 11th International Conference on Compiler Construction* (2002), pp. 111–127.

[23] Eric Bruneton. *ASM 4.0 A Java bytecode engineering library.* Version 2.0. 2011. URL: `http://download.forge.objectweb.org/asm/asm4-guide.pdf` (visited on 04/15/2017).

[24] James Gosling et al. *The Java Language Specification.* Java SE 8. Oracle America Inc. 2015. Chap. 6: Names.

[25] *API documentation of java.lang.Class<T>.* URL: `https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html` (visited on 04/15/2017).

[26] Eric Gamma et al. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., 1995. Chap. 5: Behavioral Patterns, p. 366. ISBN: 0-201-63361-2.

[27] *Apache Commons BCEL Manual – The BCEL API.* URL: `https://commons.apache.org/proper/commons-bcel/manual/bcel-api.html` (visited on 05/01/2017).

[28] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. *Thrift: Scalable Cross-Language Services Implementation.* Tech. rep. Facebook, 2007. URL: `http://thrift.apache.org/static/files/thrift-20070401.pdf` (visited on 04/13/2017).

[29] *JDK 9.* URL: `http://openjdk.java.net/projects/jdk9/` (visited on 05/04/2017).

Appendix A

Supplementary code listings

Listing A.1: PKZIP file structure representation in C. Pointers represent arrays to be allocated on the heap.

```c
#include <stdint.h>

/** Signatures in little-endian order **/
#define LOCAL_HEADER_SIGNATURE      (0x04034b50)
#define CENTRAL_DIRECTORY_SIGNATURE (0x02014b50)
#define ENDRECORD_SIGNATURE         (0x06054b50)

/** Supported compression method flags **/
#define STORE                 (0x00)  /* The file is stored (no compression)          */
#define SHRINK                (0x01)  /* The file is Shrunk                           */
#define COMPRESSION_FACTOR_1  (0x02)  /* The file is Reduced with compression factor 1 */
#define COMPRESSION_FACTOR_2  (0x03)  /* The file is Reduced with compression factor 2 */
#define COMPRESSION_FACTOR_3  (0x04)  /* The file is Reduced with compression factor 3 */
#define COMPRESSION_FACTOR_4  (0x05)  /* The file is Reduced with compression factor 4 */
#define IMPLODE               (0x06)  /* The file is Imploded                         */
#define RESERVED_TOKENIZING   (0x07)  /* Reserved for Tokenizing compression algorithm */
#define DEFLATE               (0x08)  /* The file is Deflated                         */
#define DEFLATE64             (0x09)  /* Enhanced Deflating using Deflate64           */
#define PKWARE_DCLI           (0x0a)  /* PKWARE Data Compression Library Imploding    */
#define RESERVED_0            (0x0b)  /* Reserved by PKWARE                           */
#define BZIP2                 (0x0c)  /* File is compressed using BZIP2 algorithm     */
#define RESERVED_1            (0x0d)  /* Reserved by PKWARE                           */
#define LZMA                  (0x0e)  /* LZMA (EFS)                                   */
#define RESERVED_2            (0x0f)  /* Reserved by PKWARE                           */
#define RESERVED_3            (0x10)  /* Reserved by PKWARE                           */
```

```c
#define RESERVED_4            (0x11)  /* Reserved by PKWARE                         */
#define IBM_TERSE             (0x12)  /* File is compressed using IBM TERSE         */
#define LZ77                  (0x13)  /* IBM LZ77 z Architecture (PFS)              */
#define WAVPACK               (0x61)  /* WavPack compressed data                    */
#define PPMD                  (0x62)  /* PPMd version I, Rev 1                      */

/** Supported version flags **/
#define MS_DOS_OS2        (0x00)  /* MS-DOS and OS/2 (FAT / VFAT / FAT32 file systems) */
#define AMIGA             (0x01)  /* Amiga                                          */
#define OPEN_VMS          (0x02)  /* OpenVMS                                        */
#define UNIX              (0x03)  /* UNIX                                           */
#define VM_CMS            (0x04)  /* VM/CMS                                         */
#define ATARI_ST          (0x05)  /* Atari ST                                       */
#define OS2_HPFS          (0x06)  /* OS/2 H.P.F.S.                                  */
#define MACINTOSH         (0x07)  /* Macintosh                                      */
#define Z_SYSTEM          (0x08)  /* Z-System                                       */
#define CP_M              (0x09)  /* CP/M                                           */
#define WINDOWS_NTFS      (0x0a)  /* Windows NTFS                                   */
#define MVS               (0x0b)  /* MVS (OS/390 - Z/OS)                            */
#define VSE               (0x0c)  /* VSE                                            */
#define ACORN_RISC        (0x0d)  /* Acorn Risc                                     */
#define VFAT              (0x0e)  /* VFAT                                           */
#define ALTERNATE_MVS     (0x0f)  /* Alternate MVS                                  */
#define BEOS              (0x10)  /* BeOS                                           */
#define TANDEM            (0x11)  /* Tandem                                         */
#define OS400             (0x12)  /* OS/400                                         */
#define OSX               (0x13)  /* OS X (Darwin)                                  */

/** Local file header */
typedef struct LocalFileHeader {
  uint32_t signature;
  uint16_t version;
  uint16_t general_flag;
  uint16_t compression_method;
  uint16_t mod_time;
  uint16_t mod_date;
  uint32_t crc32;
  uint32_t compressed_size;
  uint32_t uncompressed_size;
  uint16_t filename_length;
  uint16_t extra_field_length;
  char* filename;
  char* extra_field;
} LocalFileHeader;
```

```c

/** Stored file entry */
typedef struct FileEntry {
  LocalFileHeader header;
  uint8_t* data;
} FileEntry;

/** Central directory file header */
typedef struct CDirFileHeader {
  uint32_t signature;
  uint16_t version_made_by;
  uint16_t version_needed;
  uint16_t general_flag;
  uint16_t compression_method;
  uint16_t last_mod_time;
  uint16_t last_mod_date;
  uint32_t crc32;
  uint32_t compressed_size;
  uint32_t uncompressed_size;
  uint16_t filename_length;
  uint16_t extra_field_length;
  uint16_t file_comment_length;
  uint16_t disk_number_start;
  uint16_t internal_file_attributes;
  uint32_t external_file_attributes;
  uint32_t local_header_offset;
  char* filename;
  uint8_t* extra_field;
  char* comment;
} CDirFileHeader;

/** End of the central directory record */
typedef struct EndRecord {
  uint32_t signature;       // 0x06054b50, little endian
  uint16_t disk_number;
  uint16_t cdir_disk_number;
  uint16_t disk_cdir_entries;
  uint16_t total_entries;
  uint32_t cdir_size;
  uint32_t cdir_offset;
  uint16_t comment_length;
  char* comment;
} EndRecord;

```

```c
/** ZIP file structure */
typedef struct ZipFile {
  FileEntry* files;
  CDirFileHeader* cdir_headers;
  EndRecord cdir_endrecord;
} ZipFile;
```

Listing A.2: Java class structure representation as a struct in C. This is similar to the one shown in *The Java Virtual Machine Specification*, but it's real usable C code. Pointers represent arrays to be allocated on the heap.

```c
#include <stdint.h>

typedef struct AttributeInfo {
  uint16_t attribute_name_index;
  uint32_t attribute_length;
  uint8_t* info;
} AttributeInfo;

typedef struct ConstantInfo {
  uint8_t tag;
  uint8_t* info;
} ConstantInfo;

typedef struct FieldInfo {
  uint16_t access_flags;
  uint16_t name_index;
  uint16_t descriptor_index;
  uint16_t attributes_count;
  AttributeInfo* attributes;
} FieldInfo;

typedef struct MethodInfo {
  uint16_t access_flags;
  uint16_t name_index;
  uint16_t descriptor_index;
  uint16_t attributes_count;
  AttributeInfo* attributes;
} MethodInfo;



```

```
typedef struct JavaClassFile {
  /* Class file header */
  uint32_t magic_number;
  uint16_t minor_version;
  uint16_t major_version;

  /* Constant pool */
  uint16_t constant_pool_size;
  ConstantInfo* constant_pool;

  /* Class access specifier flags */
  uint16_t access_flags;

  /* Class name */
  uint16_t this_classname;

  /* Inherited super class name */
  uint16_t super_classname;

  /* Implemented interfaces */
  uint16_t interfaces_count;
  uint16_t* interfaces;

  /* Declared fields */
  uint16_t fields_count;
  FieldInfo* fields;

  /* Declared methods */
  uint16_t methods_count;
  MethodInfo* methods;

  /* Attributes */
  uint16_t attributes_count;
  AttributeInfo* attributes;
} JavaClassFile;
```

Listing A.3: Most verbose disassembly of a "Hello world" Java program using `javap`.

```
$ javap -c -s -l -private -verbose HelloWorld.class
Classfile HelloWorld.class
  Last modified Apr 14, 2017; size 426 bytes
  MD5 checksum 7a3ee81e13319873b7ac05812c3e5571
  Compiled from "HelloWorld.java"
public class HelloWorld
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
  #1 = Methodref          #6.#15         // java/lang/Object."<init>":()V
  #2 = Fieldref           #16.#17        // java/lang/System.out:Ljava/io/PrintStream;
  #3 = String             #18            // Hello world!
  #4 = Methodref          #19.#20        // java/io/PrintStream.println:(Ljava/lang/String;)V
  #5 = Class              #21            // HelloWorld
  #6 = Class              #22            // java/lang/Object
  #7 = Utf8               <init>
  #8 = Utf8               ()V
  #9 = Utf8               Code
  #10 = Utf8              LineNumberTable
  #11 = Utf8              main
  #12 = Utf8              ([Ljava/lang/String;)V
  #13 = Utf8              SourceFile
  #14 = Utf8              HelloWorld.java
  #15 = NameAndType       #7:#8          // "<init>":()V
  #16 = Class             #23            // java/lang/System
  #17 = NameAndType       #24:#25        // out:Ljava/io/PrintStream;
  #18 = Utf8              Hello world!
  #19 = Class             #26            // java/io/PrintStream
  #20 = NameAndType       #27:#28        // println:(Ljava/lang/String;)V
  #21 = Utf8              HelloWorld
  #22 = Utf8              java/lang/Object
  #23 = Utf8              java/lang/System
  #24 = Utf8              out
  #25 = Utf8              Ljava/io/PrintStream;
  #26 = Utf8              java/io/PrintStream
  #27 = Utf8              println
  #28 = Utf8              (Ljava/lang/String;)V
{
  public HelloWorld();
```

```
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
      0: aload_0
      1: invokespecial #1     // Method java/lang/Object."<init>":()V
      4: return
    LineNumberTable:
      line 1: 0

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
      0: getstatic     #2     // Field java/lang/System.out:Ljava/io/PrintStream;
      3: ldc           #3     // String Hello world!
      5: invokevirtual #4     // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      8: return
    LineNumberTable:
      line 3: 0
      line 4: 8
}
SourceFile: "HelloWorld.java"
```

Listing A.4: Simple bytecode interpreting stackmachine written in C++.

```cpp
#include <iostream>
#include <fstream>
#include <cstdint>
#include <vector>
#include <stack>
#include <iterator>

/** Supported byte instructions **/
enum Instruction
{
  LITERAL                = 0x00,
  INSTRUCTION_ADD        = 0x01,
  INSTRUCTION_SUBSTRACT = 0x02,
  INSTRUCTION_MULTIPLY  = 0x03,
  INSTRUCTION_DIVIDE    = 0x04
};

/**
 * Simplistic bytecode interpreting virtual stack machine,
 * which evaluates arithmetic operations on
 * unsigned decimal numbers in a reverse Polish notation
 * fashion.
 */
class SimplestVirtualStackMachine
{
  public:

    /**
     * Read bytecode from STL vector.
     *
     * @param[in] bytecode    Container with a series of bytes
     *                        representing bytecode instructions for
     *                        this virtual machine.
     */
    void readByteCode(const std::vector<uint8_t>& bytecode);





    /**
     * Read bytecode from file.
     *
     * @param[in] filepath    Path of file containing bytecode
     *                        instructions for this virtual machine.
     */
    void readByteCode(const std::string& filepath);

    /**
     * Iterate over bytes in buffer as instructions and literals,
     * use read byte and update the state of this virtual machine
     * accordingly.
     */
    void evaluate();

  private:

    /**
     * Addition operation
     */
    void add();

    /**
     * Substraction operation
     */
    void substract();

    /**
     * Multiplication operation
     */
    void multiply();

    /**
     * Division operation
     */
    void divide();

    // Buffer that contains bytecode read from input
    std::vector<uint8_t> byte_buffer_;

    // Instruction pointer (represented now as iterator to byte buffer)
    std::vector<uint8_t>::const_iterator instruction_pointer_;


  // Pushdown stack for storing operands
  std::stack<uint8_t> stack_;
};

void SimplestVirtualStackMachine::readByteCode(const std::vector<uint8_t>& bytecode)
{
  byte_buffer_ = bytecode;             // Read bytecode to buffer
  std::stack<uint8_t>().swap(stack_); // Reset stack to be empty
}

void SimplestVirtualStackMachine::readByteCode(const std::string& filepath)
{
  // Open file
  std::ifstream in(filepath, std::ios::in | std::ios::binary);
  in.unsetf(std::ios::skipws);  // Important: don't skip bytes that represent newlines

  // Set buffer size for efficiency
  std::streampos file_start = in.tellg();
  in.seekg(0, std::ios::end);
  std::streampos file_end = in.tellg();
  in.seekg(0, std::ios::beg);
  byte_buffer_.reserve(file_end - file_start);

  // Read bytes to buffer
  byte_buffer_.assign(std::istream_iterator<uint8_t>(in), std::istream_iterator<uint8_t>());

  // Reset stack to be empty
  std::stack<uint8_t>().swap(stack_);
}

void SimplestVirtualStackMachine::evaluate()
{
  // Fetch-decode-execute cycle
  for (instruction_pointer_ = byte_buffer_.cbegin();
       instruction_pointer_ != byte_buffer_.cend();
       ++instruction_pointer_)
  {
    switch (*instruction_pointer_)
    {
      // Numeric literal
      case LITERAL:
      {
        int value = *(++instruction_pointer_); // Read value
        stack_.push(value);                    // Push value onto stack
        break;
      }

      // Arithmetic operations
      case INSTRUCTION_ADD:        add();       break;
      case INSTRUCTION_SUBSTRACT:  substract(); break;
      case INSTRUCTION_MULTIPLY:   multiply();  break;
      case INSTRUCTION_DIVIDE:     divide();    break;

      // Handle unknown bytecode
      default:
      {
        std::cout << "Unsupported operation." << std::endl;
      }
    }
  }
}

inline void SimplestVirtualStackMachine::add()
{
  // Pop operands
  int rhs = stack_.top();
  stack_.pop();
  int lhs = stack_.top();
  stack_.pop();

  // Execute arithmetic operation
  int result = lhs + rhs;

  // Print state and push result onto stack
  std::cout << lhs << " + " << rhs << " = " << result << std::endl;
  stack_.push(result);
}

inline void SimplestVirtualStackMachine::substract()
{
  // Pop operands
  int rhs = stack_.top();
  stack_.pop();
  int lhs = stack_.top();
  stack_.pop();

  // Execute arithmetic operation
  int result = lhs - rhs;

  // Print state and push result onto stack
  std::cout << lhs << " - " << rhs << " = " << result << std::endl;
  stack_.push(result);
}

inline void SimplestVirtualStackMachine::multiply()
{
  // Pop operands
  int rhs = stack_.top();
  stack_.pop();
  int lhs = stack_.top();
  stack_.pop();

  // Execute arithmetic operation
  int result = lhs * rhs;

  // Print state and push result onto stack
  std::cout << lhs << " * " << rhs << " = " << result << std::endl;
  stack_.push(result);
}

inline void SimplestVirtualStackMachine::divide()
{
  // Pop operands
  int rhs = stack_.top();
  stack_.pop();
  int lhs = stack_.top();
  stack_.pop();

  // Execute arithmetic operation
  int result = lhs / rhs;

  // Print state and push result onto stack
  std::cout << lhs << " / " << rhs << " = " << result << std::endl;
  stack_.push(result);
}

int main(int argc, char* argv[])
{
  SimplestVirtualStackMachine vm;



  if (1 < argc)
  {
    // Interpret bytecode from file
    vm.readByteCode(argv[1]);
    vm.evaluate();
  }
  else
  {
    // Interpret bytecode from STL vector
    std::cout << "Input bytecode file was not given. Using built-in example." << std::endl;
    std::vector<uint8_t> mybytecode = {
      LITERAL,                // 0x00
      20,                     // 0x14
      LITERAL,                // 0x00
      5,                      // 0x05
      INSTRUCTION_SUBSTRACT,  // 0x02
      LITERAL,                // 0x00
      3,                      // 0x03
      INSTRUCTION_DIVIDE      // 0x04
    };
    vm.readByteCode(mybytecode);
    vm.evaluate();
  }

  return 0;
}
```

Listing A.5: Part of source from abstract **JarVisitor** class.

```java
/**
* Abstract Visitor class for traversing JAR file.
*/
public abstract class JarVisitor extends EmptyVisitor {
  private final Path absoluteJarPath;

  /**
   * Constructor for JAR visitor.
   *
   * @param jarPath   Relative or absolute file path
   */
   public JarVisitor(final Path jarPath) {
     absoluteJarPath = jarPath.toAbsolutePath();
   }

  /**
   * Start visitor and start traversing JAR file.
   *
   * @return    Reference to self
   */
   public JarVisitor start() {
     try (final JarFile jar = new JarFile(absoluteJarPath.toFile())) {
       visitJarFile(jar);
     } catch (IOException e) {
       e.printStackTrace();
     }

     return this;
   }

  /**
   * Visit JAR file.
   *
   * @param jar   JAR file to visit
   */
   public void visitJarFile(final JarFile jar) {
     jar.stream()
        .filter(entry -> !entry.isDirectory() && entry.getName().endsWith(".class"))
        .forEach(this::visitJarEntry);
   }

  /**
   * Visit entry in JAR file.
   *
   * @param entry   JAR entry to visit
   */
   public void visitJarEntry(final JarEntry entry) {
     try {
       ClassParser parser = new ClassParser(absoluteJarPath.toString(), entry.getName());
       JavaClass javaClass = parser.parse();

       javaClass.accept(this);
     } catch (IOException e) {
       e.printStackTrace();
     }
  }

  /**
   * Visit Java class.
   *
   * @param javaClass   BCEL representation of class to visit
   */
   @Override
   public abstract void visitJavaClass(final JavaClass javaClass);

   ...
```

Listing A.6: Manual synthesis of a "Hello world" class using `org.apache.bcel.generic`.

```
import org.apache.bcel.Const;
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;

...

// Generate class
ClassGen classG = new ClassGen("SynthesizedHelloWorld",           // Class name
                               "java.lang.Object",                // Super class name
                               "SynthesizedHelloWorld.java",      // Source file name
                               Const.ACC_PUBLIC,                  // Access modifier flag
                               null);                             // Interfaces
```

```

// Generate constant pool
// Get reference from class
ConstantPoolGen constantPoolG = classG.getConstantPool();
int cpIndex1 = constantPoolG.addFieldref("java.lang.System",      // Field reference
                                         "out",
                                         "Ljava/io/PrintStream;");
int cpIndex2 = constantPoolG.addString("Hello world!");           // String constant
int cpIndex3 = constantPoolG.addMethodref("java.io.PrintStream",  // Method reference
                                          "println",
                                          "(Ljava/lang/String;)V");

// Generate JVM instructions
InstructionList instructions = new InstructionList();

// Get "System.out" static field for "println()"
instructions.append(new GETSTATIC(cpIndex1));

// Push "Hello world!" contant onto stack
instructions.append(new LDC(cpIndex2));

// Invoke "System.out.println()"
instructions.append(new INVOKEVIRTUAL(cpIndex3));

// Return from method
instructions.append(new RETURN());

// Generate method
MethodGen methodG =
  new MethodGen(Const.ACC_PUBLIC |                          // Access modifier flags
                Const.ACC_STATIC,                           // (public static)
                Type.VOID,                                  // Return type
                new Type[]{new ArrayType(Type.STRING, 1)},  // Parameter types
                new String[]{"args"},                       // Parameter names
                "main",                                     // Method name
                "SynthesizedHelloWorld",                    // Class name
                instructions,                               // JVM instructions
                constantPoolG);                             // Constant pool
methodG.setMaxStack();    // Calculate and set maximum stack size

// Add method to class
classG.addMethod(methodG.getMethod());


```

```
// Create "non-generic" class from "generic" class
JavaClass javaClass = classG.getJavaClass();

...
```

Appendix B

Supplementary diagrams

B.1 UML class diagram of JAR reader prototype

UML class diagrams are made in Eclipse with Papyrus modeling tools.

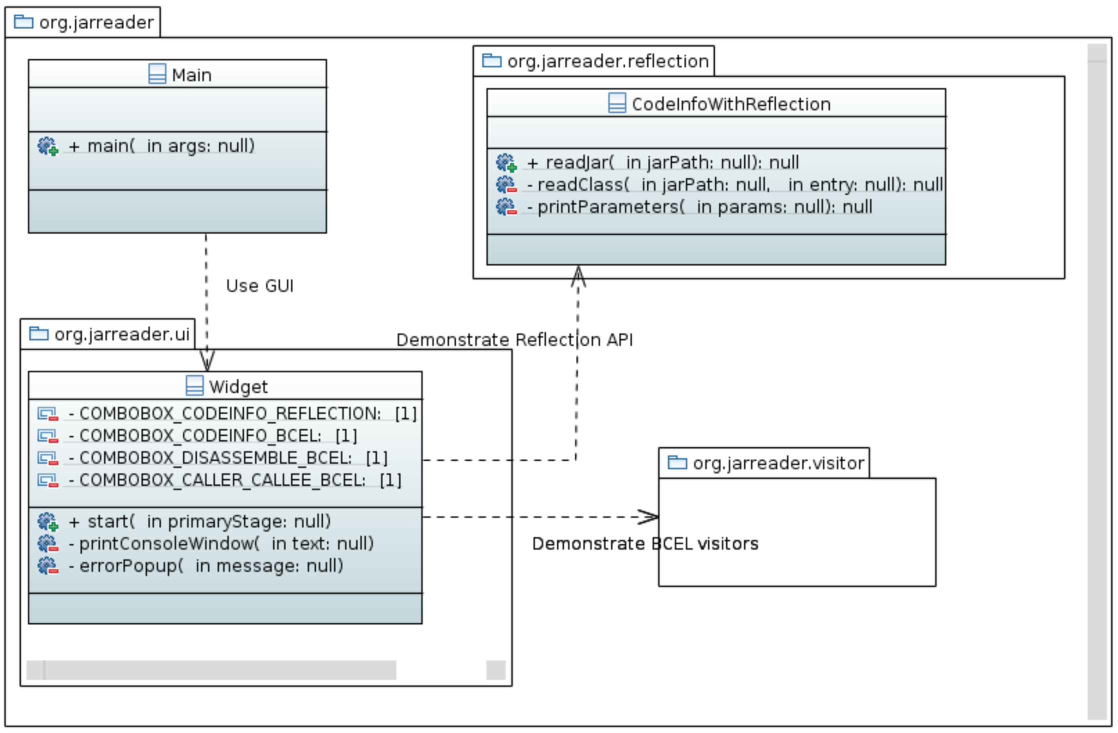
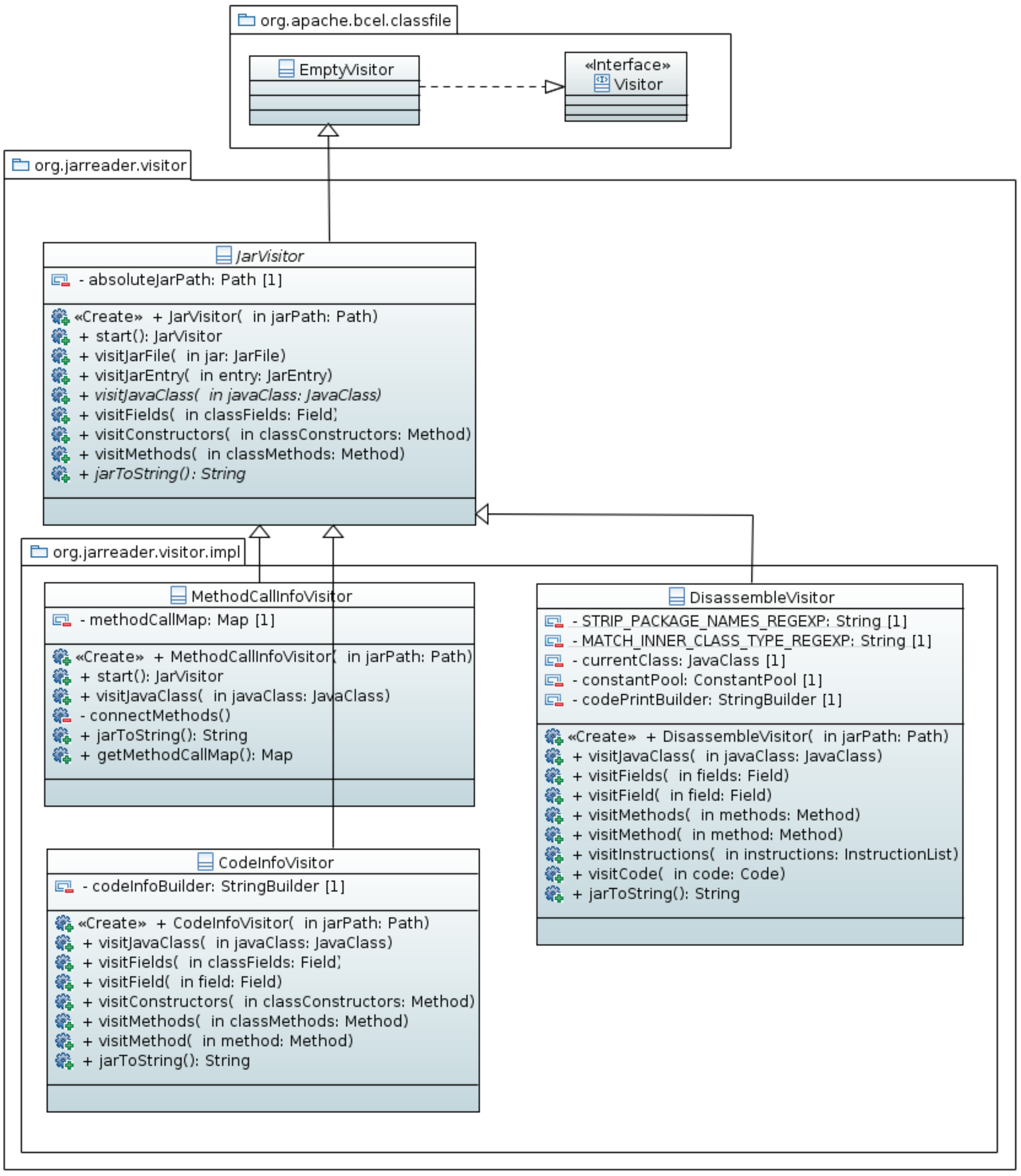

org.jarreader
Main
+ main(in args: null)
Use GUI
org.jarreader.reflection
CodeInfoWithReflection
+ readJar(in jarPath: null): null
- readClass(in jarPath: null, in entry: null): null
- printParameters(in params: null): null
Demonstrate Reflection API
org.jarreader.ui
Widget
- COMBOBOX_CODEINFO_REFLECTION: [1]
- COMBOBOX_CODEINFO_BCEL: [1]
- COMBOBOX_DISASSEMBLE_BCEL: [1]
- COMBOBOX_CALLER_CALLEE_BCEL: [1]
+ start(in primaryStage: null)
- printConsoleWindow(in text: null)
- errorPopup(in message: null)
org.jarreader.visitor
Demonstrate BCEL visitors

Figure B.1: UML class diagram of overall JAR reader prototype. For details of `org.jarreader.visitor` package, refer to Figure B.2

.



org.apache.bcel.classfile
EmptyVisitor
«Interface» Visitor
org.jarreader.visitor
JarVisitor
- absoluteJarPath: Path [1]
«Create» + JarVisitor(in jarPath: Path)
+ start(): JarVisitor
+ visitJarFile(in jar: JarFile)
+ visitJarEntry(in entry: JarEntry)
+ visitJavaClass(in javaClass: JavaClass)
+ visitFields(in classFields: Field)
+ visitConstructors(in classConstructors: Method)
+ visitMethods(in classMethods: Method)
+ jarToString(): String
org.jarreader.visitor.impl
MethodCallInfoVisitor
- methodCallMap: Map [1]
«Create» + MethodCallInfoVisitor(in jarPath: Path)
+ start(): JarVisitor
+ visitJavaClass(in javaClass: JavaClass)
- connectMethods()
+ jarToString(): String
+ getMethodCallMap(): Map
CodeInfoVisitor
- codeInfoBuilder: StringBuilder [1]
«Create» + CodeInfoVisitor(in jarPath: Path)
+ visitJavaClass(in javaClass: JavaClass)
+ visitFields(in classFields: Field)
+ visitField(in field: Field)
+ visitConstructors(in classConstructors: Method)
+ visitMethods(in classMethods: Method)
+ visitMethod(in method: Method)
+ jarToString(): String
DisassembleVisitor
- STRIP_PACKAGE_NAMES_REGEXP: String [1]
- MATCH_INNER_CLASS_TYPE_REGEXP: String [1]
- currentClass: JavaClass [1]
- constantPool: ConstantPool [1]
- codePrintBuilder: StringBuilder [1]
«Create» + DisassembleVisitor(in jarPath: Path)
+ visitJavaClass(in javaClass: JavaClass)
+ visitFields(in fields: Field)
+ visitField(in field: Field)
+ visitMethods(in methods: Method)
+ visitMethod(in method: Method)
+ visitInstructions(in instructions: InstructionList)
+ visitCode(in code: Code)
+ jarToString(): String

Figure B.2: UML class diagram of abstract `JarVisitor` and its sub classes. `JarVisitor` extends `EmptyVisitor` from BCEL.

B.2 UML class diagram of refactored Java source parser

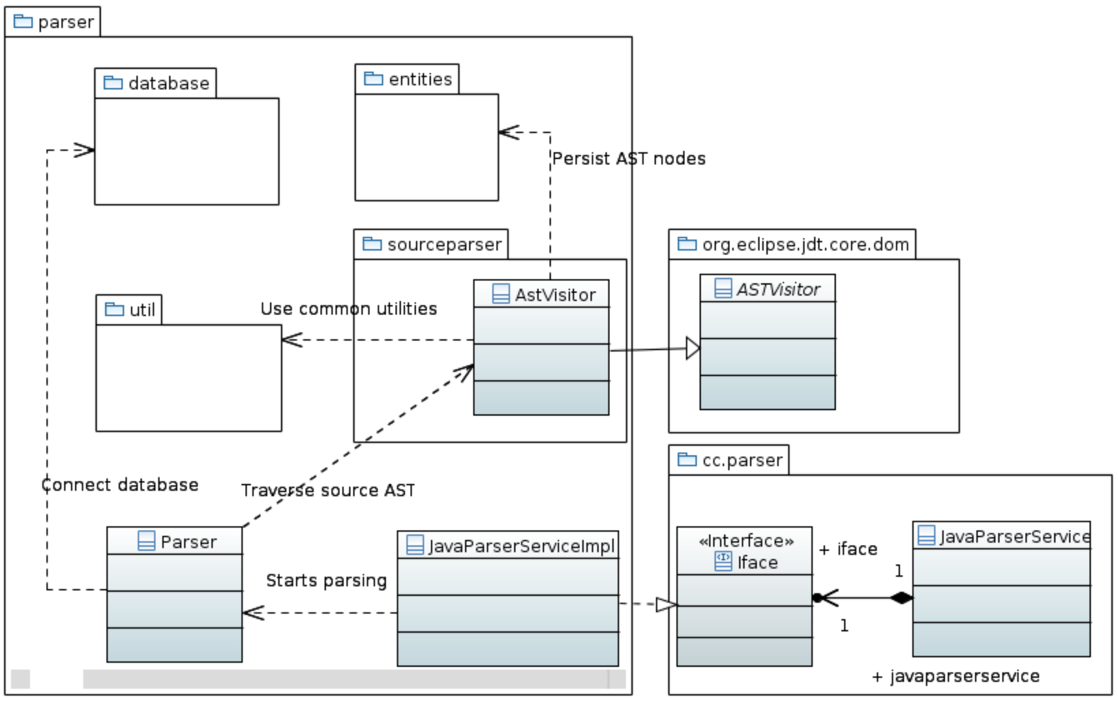


Figure B.3: UML class diagram of refactored Java source parser module of CodeCompass. Only classes important for the parsing pipeline are listed.

B.3 Relational model diagram of project database made by CodeCompass

The relational model diagrams are generated by Schemaspy program from the schema of a running PostgreSQL database. This software can be found here:
`http://schemaspy.sourceforge.net`

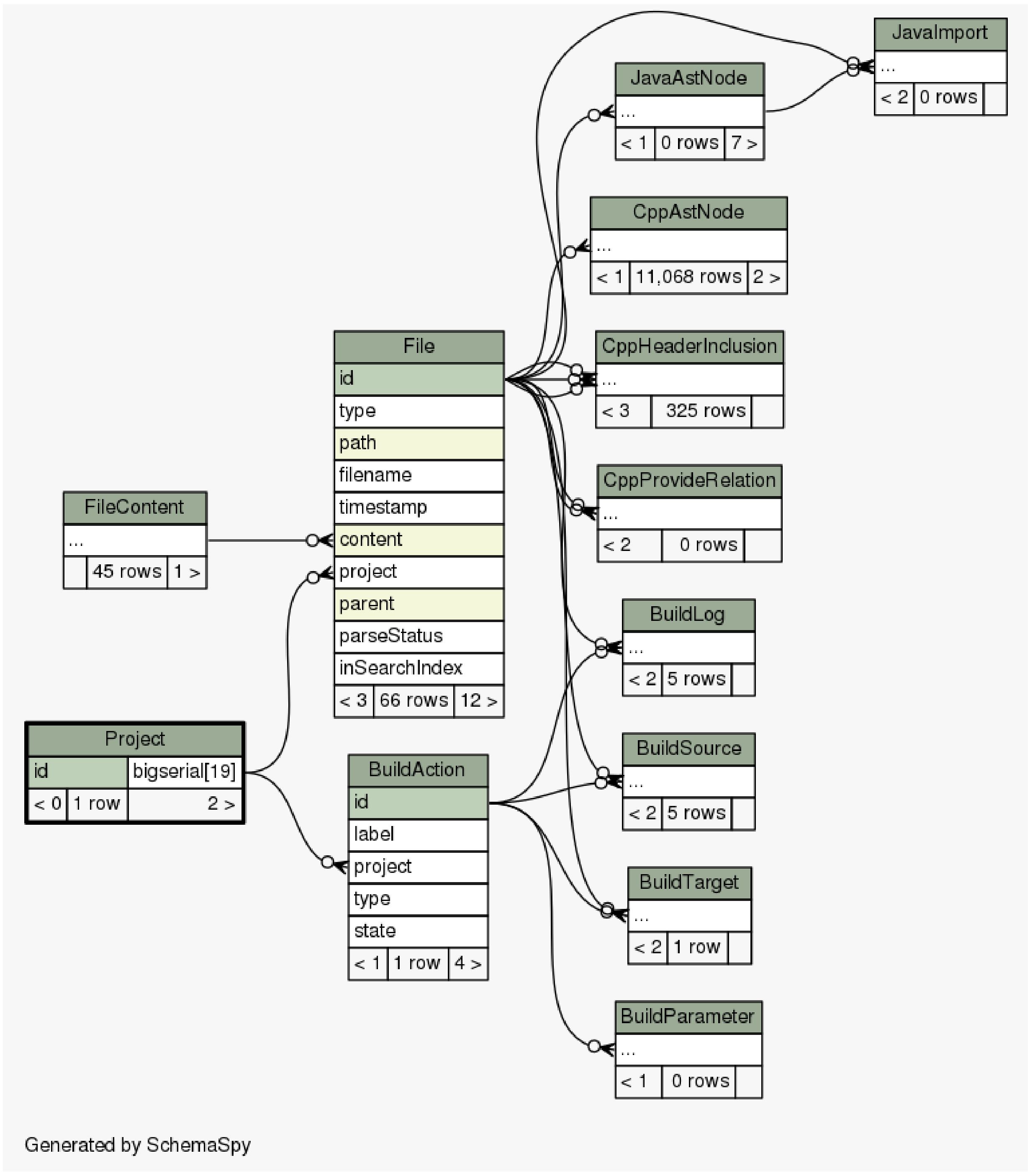


Figure B.4: Project entity and it's relations.

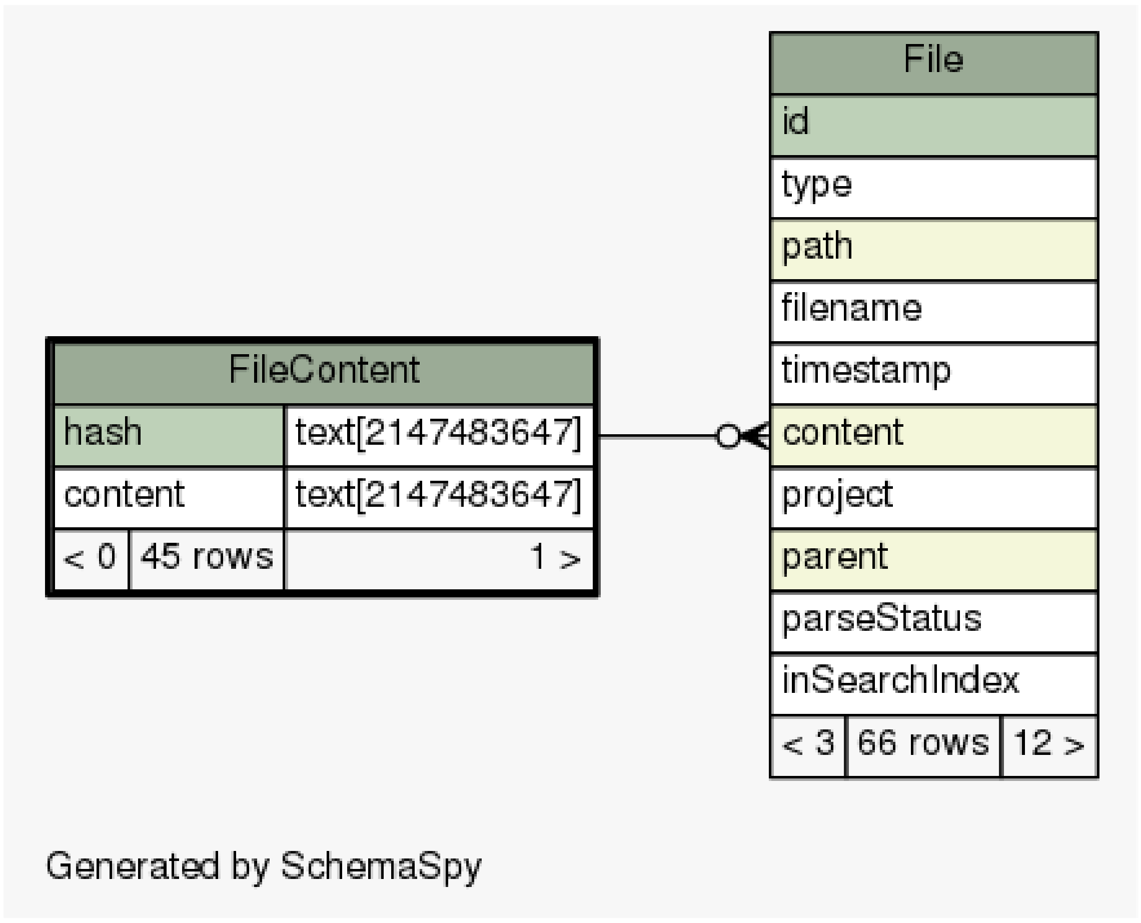


Figure B.5: Relationship between File and FileContent entities. Content of a file is stored as a Fowler-Noll-Vo hash.

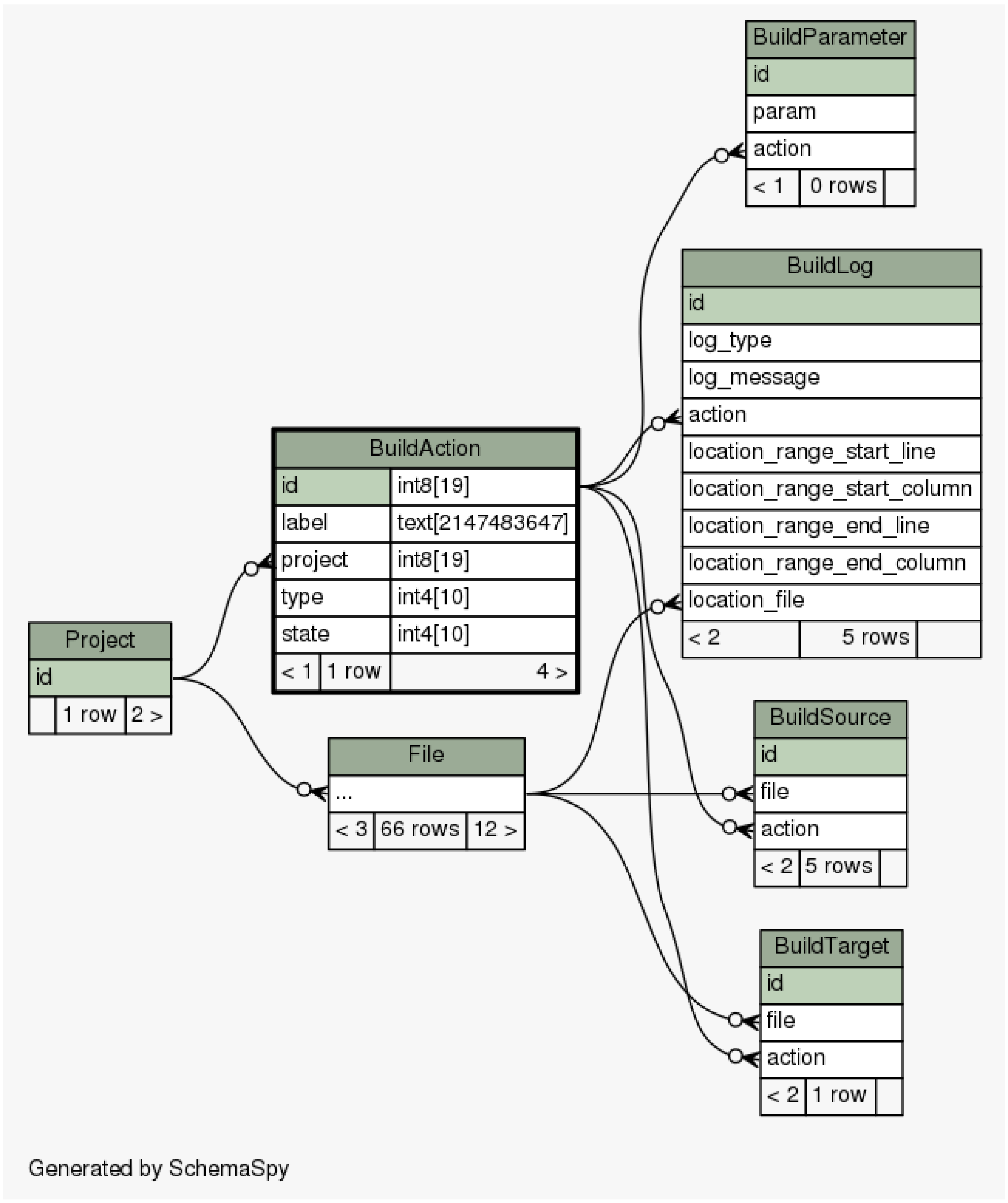


Figure B.6: BuildAction entity and it's relations.

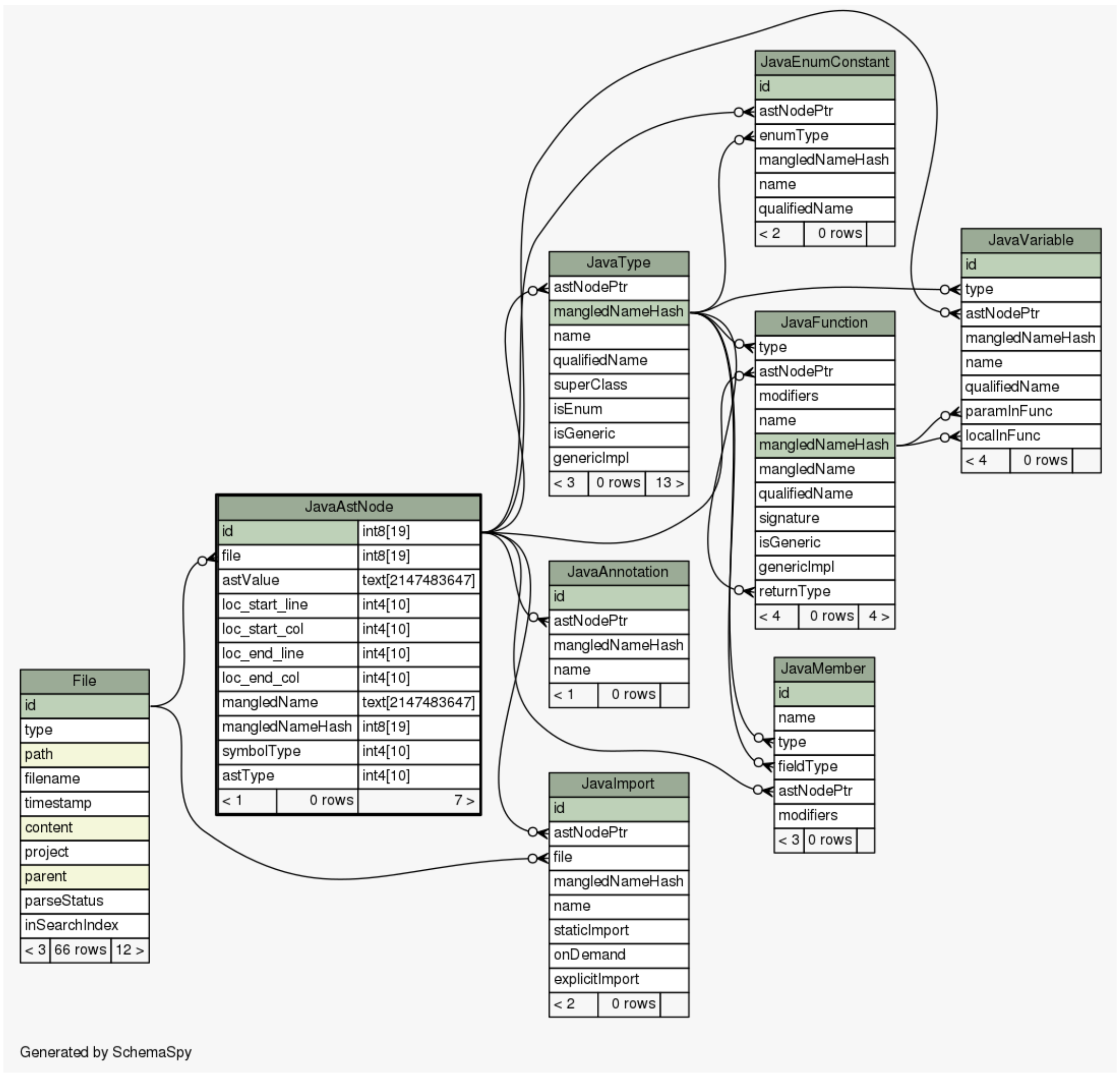


Figure B.7: JavaAstNode entity and it's relations.

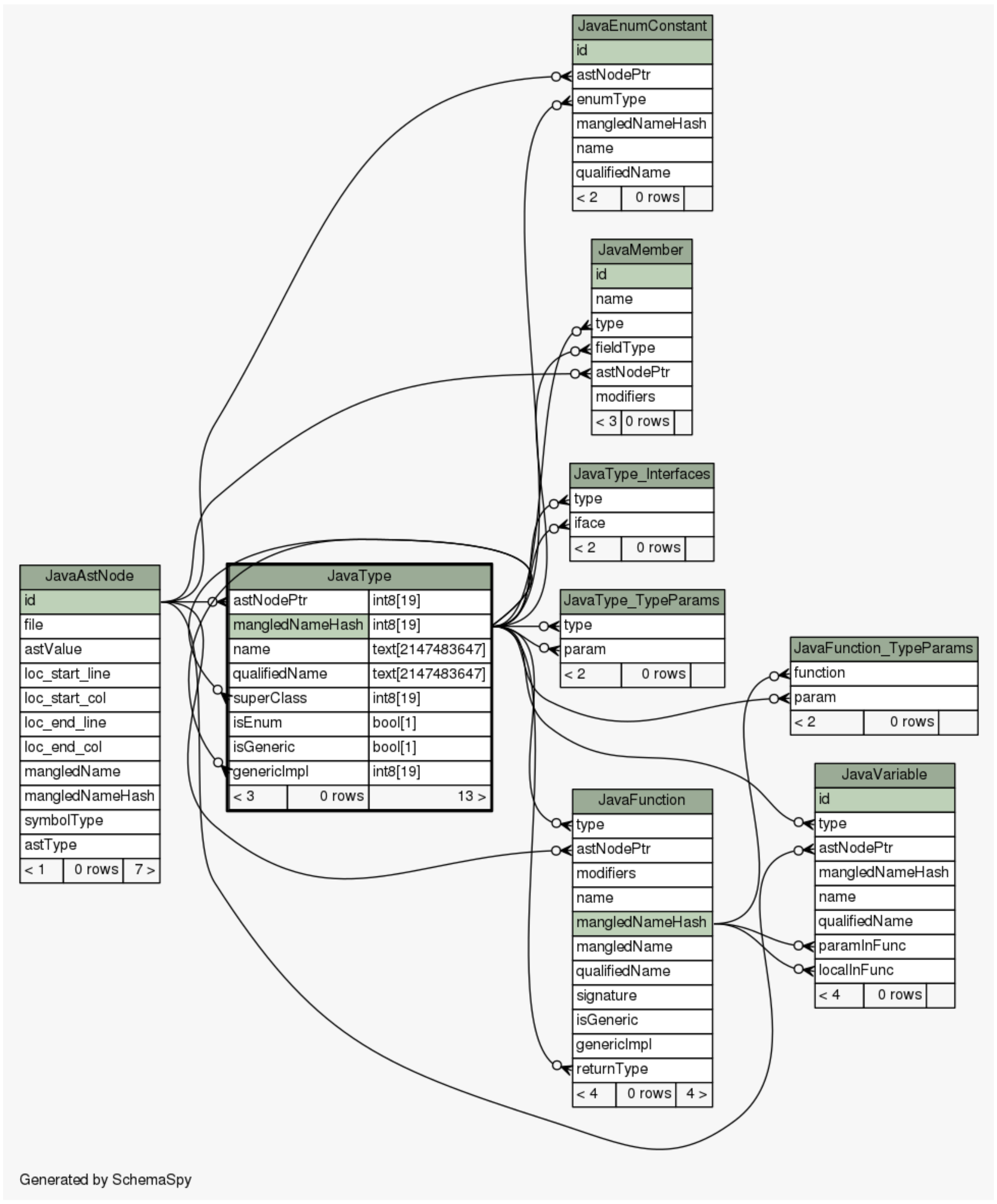


Figure B.8: JavaType entity and it's relations.