JOHANNES KEPLER
UNIVERSITY LINZ

Author
Mathias Wöß, BSc
k11709064

Submission
Institute for System Software

Thesis Supervisor
o.Univ.-Prof. Dr.
Hanspeter Mössenböck

February 2024

A Profiler for Java Programs

Master Thesis

to obtain the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Abstract

Profilers are an important aid for developers to find bottlenecks and unused code in programs. Over the years many such tools have been developed for the Java language and ecosystem, each with their own advantages and caveats. We present a novel profiling tool that is fully open-source, has a permissive license and is simple to set up and use. While most Java profilers use bytecode instrumentation, our profiler inserts counter statements directly into the source code. To analyze raw Java files we leverage the Coco/R compiler generator, coupled with a small and reusable fuzzy grammar. The grammar is designed to be robust against future language changes and is tailored to gather only necessary metadata about Java source files. In its core, our profiler is a multi-platform command-line tool that generates HTML reports. The report serves as a structured, sorted and detailed overview of how often methods, blocks and statements were executed. To ensure correctness and the compatibility of our profiler to Java Version 17, we relied upon an extensive unit test suite and the successful instrumentation of large code repositories (like jUnit 5). For improved usability and comfort, we additionally provide a JavaFX user interface to easily configure the parameters for our tool and to automatically execute it with the press of a button.

Kurzfassung

Profiler sind eine wichtige Hilfe für Entwickler, um Engpässe und unbenutzten Code in Programmen zu finden. Über die Jahre wurden viele solche Werkzeuge für Java und sein Ökosystem entwickelt, jedes mit seinen eigenen Vor- und Nachteilen. Wir stellen ein neuartiges Profiling-Tool vor, das Open Source ist, über eine freizügige Lizenz verfügt und einfach einzurichten und zu verwenden ist. Während fast alle anderen Java-Profiler Bytecode-Instrumentierung verwenden, fügt unser Profiler Zähleranweisungen direkt in den Quellcode ein. Für die Analyse von Java-Code-Dateien verwenden wir den Parsergenerator Coco/R zusammen mit einer kleinen und wiederverwendbaren "fuzzy" Grammatik. Diese ist so gestaltet, dass sie robust gegen zukünftige Sprachänderungen ist und nur die notwendigen Metadaten aus Java Dateien extrahiert. Im Kern ist unser Profiler ein plattformübergreifendes Befehlszeilentool, das HTML-Berichte generiert. Die Berichte dienen als eine strukturierte, sortierte und detaillierte Übersicht darüber, wie oft Methoden, Blöcke und Anweisungen ausgeführt wurden. Um die Korrektheit und Kompatibilität unseres Profilers zur Java-Version 17 sicherzustellen, verlassen wir uns auf eine umfangreiche Unit-Test-Suite und das erfolgreiche Instrumentieren von großen Code-Repositories (wie jUnit 5). Für Benutzerfreundlichkeit und Komfort wurde zusätzlich eine JavaFX-Benutzeroberfläche bereitgestellt, mit der die Parameter für das Tool einfach konfiguriert und per Knopfdruck ausgeführt werden können.

Contents

1 Introduction

Software profilers are commonly used to find hot-spots in programs and to determine code coverage. Hot-spots are frequently accessed methods and code blocks that contribute most to the execution time of a program. When trying to optimize run-time performance, they are the main focus, as improving the associated code of these methods should induce the biggest gains in efficiency. Re-writing rarely accessed code usually leads to less significant results. Ain et al. [1] describe in their work, how the Just-in-Time (JIT) compiler of Java already identifies hotspot methods automatically. It then tries to compile their bytecode to native machine code to speed up programs. This is achieved by keeping counters for each method and by optimizing frequently accessed ones first [2].

Finding parts of the code that are never executed during a *representative* execution of a software program helps us identify dead and obsolete code, that might be due to removal or require further testing. Often the underlying problem of code not being executed at all, is due to semantic errors in the source code. Without the help of coverage analysis, this can lead to unintended behavior and bugs.

Tengeri et al. [3] write how code coverage is used as an effective white-box testing tool and as a measure for test completeness. It is a general software quality assessment and can be used for fault localization, test case generation and test prioritization.

Profilers can be grouped into two main categories: Sampling and instrumenting profilers. In Table 1.1 the two main differences between these types are listed.

	Sampling	Instrumenting
Run-time overhead	Minimal	Significant
Accuracy	Representative	Exact

Table 1.1: Main differences between sampling and instrumenting profilers

According to Mytkowicz et al. [4], sampling profilers work by interrupting the program in intervals and recording which method the program is currently executing. Depending on the sampling rate, this commonly results in a low run-time impact but inaccurate results.

While hot-spots will still be found, rarely executed parts might never be sampled, especially if their execution times are short. These type of profilers also require longer runs to get representative statistics. The longer the software is executed, the closer we get to the actual distribution of method executions.

Instrumenting profilers work by inserting counter increment instructions into the source code. Whenever a piece of instrumented code is executed, the program increases its associated counter value by one. This way, we get an exact number of how often each block or method was entered during execution [5]. The consequence is a bigger slow-down of the program, because the original control flow is interrupted more frequently. This approach, however, leads to exact results for quick, single-run programs, providing reliable accuracy about the non-covered code parts.

In the scope of this thesis, an *instrumenting* profiler for Java programs was developed. The target version at the time of writing was Java 17. When implementation efforts began, it was the most recent long-term-support (LTS) release. While most existing Java profilers are implemented using bytecode instrumentation, our profiler uses source code instrumentation with a parser generated by Coco/R [6]. Figure 1.1 sketches the main difference between these types of instrumentation.

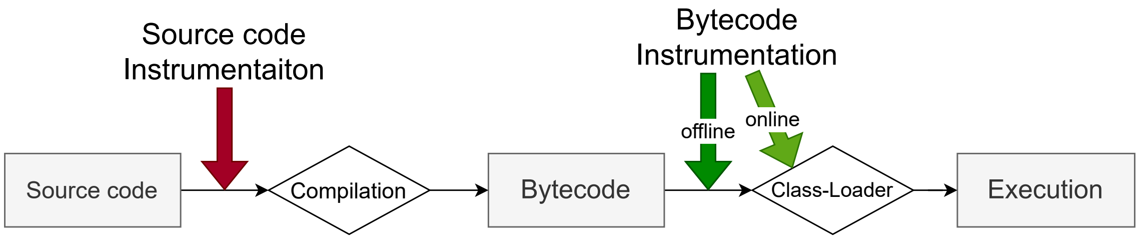


Figure 1.1: Difference between source code instrumentation and bytecode instrumentation. Source code instrumenters are modifying the code before compilation, bytecode instrumenters inject counter code into the bytecode before execution (either offline or online [3]).

Source code instrumentation is done on the raw `*.java` project files by inserting additional statements into the code. For bytecode instrumentation we first have to translate the code into `*.class` files (using the Java bytecode format [7]). These are then altered before usage by injecting instrumentation code. Modifying and writing new class files *before* starting the program is called *offline* bytecode instrumentation. Dynamically adding counters during the class-loading phase is referred to as *online* instrumentation [3]. Offline can also be referred to as *static* while online is called *dynamic* instrumentation [8].

In the end, a profiler should generate overviews of the entire project for a program execution and allow interactive exploration. The user should be presented with statistics about the most frequently used classes, methods and statements in the program.

1.1 Motivation

Java is a widely used language and Java profiling tools are crucial to optimize application performance and to find bottlenecks. There are many existing profiler tools available for the Java ecosystem, but many have drawbacks like being proprietary, expensive or complex to setup and use.

To address these limitations, we present a novel Java profiler using source code instrumentation, that is open-source and free to use. While most available profilers use bytecode instrumentation due to its many advantages, ours needs access to the source code itself. This offers several advantages.

Source code instrumentation over bytecode instrumentation

The decision to utilize source code instrumentation allows for more precise profiling as it can access and modify the actual source files. Tengeri et al. [3] mention how bytecode instrumentation can be problematic for *correctly* mapping the profiling data back to the sources. When we translate to bytecode, we lose the direct correlation between the source code positions of statements and the executed code. It is then challenging to annotate the sources correctly with the coverage data. Thus, source code instrumentation is more intuitive, straightforward and comprehensible.

Accessibility and Compatibility

Many existing tools are restricted by the target Java version, the operating system or a prohibitive price tag. This often makes their usage unfit for developers and researchers alike. Our tool is compatible with Java code up to Version 17, independent of the runtime or VM implementation used. It is platform-independent and has open and free access. Altering bytecode online or offline can lead to unintended side-effect (like preventing the Java VM to successfully bootstrap [8]), which is also avoided by our approach.

Maintainability and Simplicity

The core of our profiler uses fuzzy parsing, tailored to its use case, providing high maintainability and robustness against Java language changes. The parser specification grammar can handle diverse code bases reliably and can be easily customized and re-used for new applications, related to Java source code analysis and processing.

In the spirit of simplicity, our profiler generates HTML reports over the command-line interface, as opposed to the conventional GUI-based tooling. User interfaces are often cluttered, unintuitive or slow. They usually depend on platform-specific and fast-aging toolkits that pose extra maintainability overhead. Our reports can be inspected on practically any device, without installing additional software (just a browser is needed) and remain future-proof for a long time.

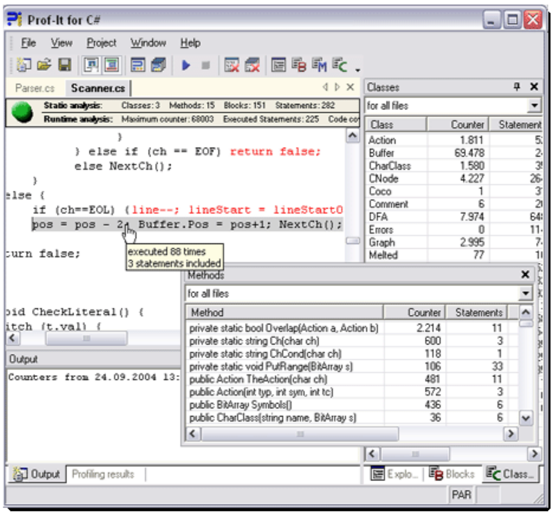
The intention of our work was to create an open-source, simple to understand and free to use tool to analyze Java projects. The profiler code itself does not depend on any external libraries.

1.2 Related Work

A plethora of Java profiling and coverage tool already exist, working in different ways and having diverse advantages and disadvantages. We looked into several of them and compared some key points.

Prof-It

A similar project to ours was developed in 2004 at the Institute of System Software (SSW)[1] of the Johannes Kepler University of Linz. It was called "Prof-It" [9] and is a source code instrumenting profiler for C# 3.0 programs. As seen in Figure 1.2, it uses a dated graphical user interface for all of its usecases and only works on Windows. Upgrading it to a new UI toolkit, while keeping all functionality intact, would be a tedious task.



Static analysis: Classes: 3 Methods: 15 Blocks: 151 Statements: 282
Runtime analysis: Maximum counter: 68003 Executed Statements: 225 Code co

```
            }
        } else if (ch == EOF) return false;
        else NextCh();
    }
else {
    if (ch==EOL) (line--; lineStart = lineStart0
    pos = pos - 2; Buffer.Pos = pos+1; NextCh();

turn false;


oid CheckLiteral() {
itch (t.val) {
```

executed 88 times
3 statements included

Class	Counter	Statement
Action	1.811	5
Buffer	69.478	2
CharClass	1.580	3
CNode	4.227	26
Coco	1	3
Comment	6	2
DFA	7.974	64
Errors	0	11
Graph	2.995	7
Melted	77	1

Method	Counter	Statements
private static bool Overlap(Action a, Action b)	2.214	11
private static string Ch(char ch)	600	3
private static string ChCond(char ch)	118	1
private static void PutRange(BitArray s)	106	33
public Action TheAction(char ch)	481	11
public Action(int typ, int sym, int tc)	572	3
public BitArray Symbols()	436	6
public CharClass(string name, BitArray s)	36	6

```
Counters from 24.09.2004 13:
```

Figure 1.2: Screenshot of the Prof-It application's main window (taken from its website [9]).

Prof-It also uses (a modified version of) Coco/R as a parser generator and a fully specified attributed grammar (ATG). Our project is roughly inspired by this work, although we tried to provide all functionality as a cross-platform command-line tool for Java programs. Compared to our Java 17 fuzzy grammar of roughly 200 lines of code, their fully-fledged C# 3.0 grammar consists of several thousands of lines (including the semantic actions).

Java Flight Recorder

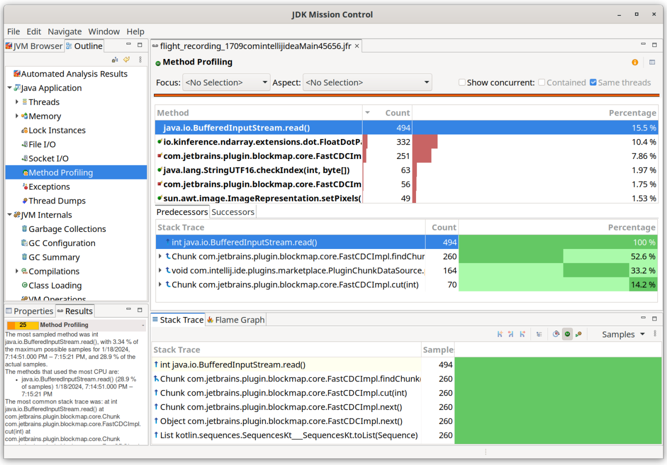
This tool for collecting diagnostic and profiling data is part of the HotSpot JVM itself. It was released and open-sourced with the release of OpenJDK 11 (see also JEP 328 [10]). For production usage it requires a commercial license [11]. The efficiently stored binary

[1] https://ssw.jku.at/

data reported by this utility can be visualized and explored using Java Mission Control (JMC) [12] and other tools.

Java Flight Recorder (JFR) collects an abundance of data by recording VM events. Event types include locks, garbage collection, methods, thread states, I/O operations, class loading and many others. The tool and the API were designed to have a minimal run-time performance overhead of roughly 1%. Which events to listen for can be specified when starting or attaching the recorder.

In Figure 1.3 we see a screenshot of the Eclipse-based user interface of JMC showing the visualization of a sample dump, recorded while opening a Java project using IntelliJ IDEA. JMC is free-to-use and can be downloaded from its Oracle homepage[2].



Focus: <No Selection> Aspect: <No Selection> □ Show concurrent: □ Contained ☑ Same threads

Method	Count	Percentage
java.io.BufferedInputStream.read()	494	15.5 %
io.kinference.ndarray.extensions.dot.FloatDotP	332	10.4 %
com.jetbrains.plugin.blockmap.core.FastCDCIm	251	7.86 %
java.lang.StringUTF16.checkIndex(int, byte[])	63	1.97 %
com.jetbrains.plugin.blockmap.core.FastCDCIm	56	1.75 %
sun.awt.image.ImageRepresentation.setPixels(	49	1.53 %

Stack Trace	Count	Percentage
int java.io.BufferedInputStream.read()	494	100 %
Chunk com.jetbrains.plugin.blockmap.core.FastCDCImpl.findChur	260	52.6 %
void com.intellij.ide.plugins.marketplace.PluginChunkDataSource.	164	33.2 %
Chunk com.jetbrains.plugin.blockmap.core.FastCDCImpl.cut(int)	70	14.2 %

25 Method Profiling

The most sampled method was int java.io.BufferedInputStream.read(), with 3.34 % of the maximum possible samples for 1/18/2024, 7:14:51.000 PM – 7:15:21 PM, and 28.9 % of the actual samples.
The methods that used the most CPU are:

- java.io.BufferedInputStream.read() (28.9 % of samples) 1/18/2024, 7:14:51.000 PM – 7:15:21 PM

The most common stack trace was: at int java.io.BufferedInputStream.read() at com.jetbrains.plugin.blockmap.core.Chunk com.jetbrains.plugin.blockmap.core.FastCDCImpl.cut(int) at com.jetbrains.plugin.blockmap.core.Chunk

Stack Trace	Sample
int java.io.BufferedInputStream.read()	494
Chunk com.jetbrains.plugin.blockmap.core.FastCDCImpl.findChunk	260
Chunk com.jetbrains.plugin.blockmap.core.FastCDCImpl.cut(int)	260
Chunk com.jetbrains.plugin.blockmap.core.FastCDCImpl.next()	260
Object com.jetbrains.plugin.blockmap.core.FastCDCImpl.next()	260
List kotlin.sequences.SequencesKt___SequencesKt.toList(Sequence)	260

Figure 1.3: Screenshot of the JDK Mission Control UI, showing method profiling data from opening a Java project using IntelliJ IDEA. The data indicate that `java.io.BufferedInputStream.read()` was the most used method.

The JFR and JMC utilities are very advanced, provide detailed information about all kinds of useful events, support versatile visualizations and allow for detailed exploration. It is not an exact profiler, but rather designed for gathering data about production-level applications without causing significant slowdowns.

The important distinction between JFR and our profiling software is that recording VM events is independent of source code. No coverage and statements hit-counts can be retrieved directly from this diagnostics data.

[2] `https://www.oracle.com/java/technologies/javase/products-jmc8-downloads.html`

VisualVM

Similar to JFR and JMC, VisualVM [13] was also once part of the JDK itself, but starting with JDK 9 it has moved to GitHub as a standalone distribution.

This NetBeans-platform-based software works in a similar way as these previously mentioned tools and is focused on directly attaching to programs running in the JVM. It allows for CPU, memory and other profiling methods. It is also open-source and free to use.

VisualVM is also not source-code-oriented and should primarily be used to monitor run-time performance and memory management behavior.

Coverage tools

Cobertura [14] is a code coverage library, providing line and branch coverage. It can generate detailed HTML reports from the result data, gathered by running bytecode-instrumented code. It also features an Eclipse plugin to directly add it into the Integrated Development Environment (IDE). Its main disadvantage is that it has problems with Java 8 bytecode and is not compatible with recent Java versions.

JaCoCo [15] delivers source code and branch coverage and the cyclomatic complexity [16] measure. It supports the most recent Java versions and is actively maintained. Many integrations into build tools and IDEs are available[3], making it an excellent choice for anyone interested in Java code coverage. Again, this is achieved by bytecode instrumentation of Java class files, compiled with debug information available. However, JaCoCo focuses only on coverage. No hit counts and execution frequency measures are provided. Rather, the report focuses on coverage and missed instructions. Also, its documentation[4] states that:

> Not all Java language constructs can be directly compiled to corresponding byte code. In such cases the Java compiler creates so called *synthetic* code which sometimes results in unexpected code coverage results.

[3] `https://www.jacoco.org/jacoco/trunk/doc/integrations.html`

[4] `https://www.jacoco.org/jacoco/trunk/doc/counters.html`

Proprietary profiling software

Many additional profilers for the Java ecosystem exist. Most of them come with a price tag. jProfiler[5] is a popular commercial tool, similar to JMC and VisualVM, providing excellent insight into the VM state while running Java programs. YourKit[6] is another fully-featured profiling software for the JVM. It offers many integrations and modern visualizations. The popular development tool IntelliJ IDEA, in its *Ultimate* edition, has a Java profiler [17] built-in. Many popular inspections like call trees, flame graphs and CPU and memory visualizations are available. It is also interoperable with JFR. One can start an application directly with it or attach the profiler to running programs. Again bytecode profiling is used. A separate *coverage* analysis tool is also available in this IDE and is included in the free-to-use Community Edition.

1.3 Outline

The further structure of this thesis is as follows: In Chapter 2 we introduce the technologies used and mention sources of truth, relied upon during the implementation of the parser. In Chapter 3 we describe the general idea and steps of our profiler and provide insight into how the object model is build. It contains some details about how counters are placed and incremented and what the profiler outputs to the file system. In Chapter 4 we elaborate further on the implementation, how the project is structured and the special handling of tricky language features. Chapter 5 explains how to install and use the software. Our systematic tests, evaluation of run-time performance and current limitations can be found in Chapter 6. Finally, we give a short summary and an outlook into future work in Chapter 7.

[5] `https://www.ej-technologies.com/products/jprofiler/overview.html`

[6] `https://www.yourkit.com/java/profiler/features/`

2 Background

In this section we go over the software and resources we relied upon to create our profiler and to make sure that it is working correctly.

2.1 Sources

The foundation on which we based our initial approach was the advanced knowledge of the Java language acquired through many years of school and university teachings, but also personal corporate experience. We already knew how programs can look like, which statements exist and how they are executed. We also knew how to correctly use the Java compiler, how directories should be structured and much more.

Additionally, the main source of truth while designing our ATG was the official Java Language Specification [18] from the Oracle website. It defines fixed rules of which types of statements can appear where and what types of symbols can follow. The Java compiler follows these definitions exactly, giving us an error if any part of the source code does not adhere to these rules.

All implementation and test code was written using IntelliJ IDEA. Its pre-compile checker and semantic validity analysis provides instant aid before even attempting to compile a sample. It can be used to experiment and verify possible statement combinations.

The Coco/R ATG for Java 1.4 from the SSW homepage[1] has also given us a jump-start and some hints for how to design the basic grammar elements. Although it was made for a very outdated Java version and is not designed for fuzzy parsing, it gave us invaluable pointers for designing the initial scanner token definitions (like `floatLit` and `charLit`).

[1] `https://ssw.jku.at/Research/Projects/Coco/Java/Java.ATG`

Lastly, internet resources like the Java Documentation and Help Center [19] and tutorial sites like Baeldung[2] provided quick and helpful elaboration about what is possible, using code-samples-based explanations.

2.2 Attributed grammars and Coco/R

Programs are written as structured text in programming languages defined by a set of grammatical rules. To be able to parse and interpret a program's instructions and declarations, these rules are specified in a language grammar. From these rules alone, we can generate a parser for this language. Parsers can be used to check the syntax of a source file, find errors and create compilers.

Coco/R [6] is a recursive descent compiler generator. It builds a scanner and a parser from an attributed grammar. Both the scanner and parser specification should be written in the Extended Backus-Naur Form (EBNF) [20].

The scanner description defines character sets and terminal classes. Listing 2.1 shows a minimal Coco/R scanner definition, declaring the character sets `letter` and `digit` and a token declaration `ident`. An `ident` always starts with a letter (or an underscore) and is followed by any number of letters and digits. The recognition of the token will be terminated when any other character is encountered. This scanner would correctly identify any variable, type or class name in a program. The specification also defines which symbols to ignore, namely the tabulator character, line-feed and carriage-return. Comments start with `"//"` and tehir contents will be ignored until the next line-feed character.

```
CHARACTERS
  letter = ’a’..’z’ + ’A’..’Z’ + ’_’.
  digit  = ’0’..’9’.
TOKENS
  ident = letter {letter | digit}.
COMMENTS FROM "//" TO ’\n’
IGNORE ’\t’ + ’\r’ + ’\n’
```

Listing 2.1: Scanner definition for the token `ident` and its character sets. It also specifies how comments are made and what characters to ignore.

[2] https://www.baeldung.com/

The generated scanner is a deterministic finite automaton (DFA). It provides a stream of tokens that can be consumed by the parser. The parser has to be specified as a set of productions following a context-free EBNF format.

An example can be found in Listing 2.2. This parser specification will produce a parser that can process Java files and find all class definitions, including nested classes, in a fuzzy way. A java file starts with an optional package declaration, followed by zero or more class definitions. The `ANY` token will match any token that is not an alternative to this `ANY`, i.e., any other token except for `"class"`. After `"class" ident` in the non-terminal symbol (NTS) `Class`, we skip all tokens until the next opening brace `"{"`. Inside the class body there can be zero or more nested class declarations, curly-brace expressions or any other tokens. A `BraceExpr` can be arbitrarily nested and lets us ignore all methods and inner blocks without missing any following class definitions.

```
PRODUCTIONS
JavaFile = [PackageDecl] {Class | ANY}.
PackageDecl = "package" ident {"." ident} ";".
Class = "class" ident {ANY} "{" {Class | BraceExpr | ANY} "}".
BraceExpr = "{" {BraceExpr | ANY} "}".
```

Listing 2.2: Parser definition to find all class definitions in a Java file including nested classes.

We can then augment these productions with semantic actions and attributes to add custom logic into the generated parser. Attributes are specified in angle brackets `"<[out] attr>"`, which will be added to productions as method parameters. Semantic actions can be defined inside `"(."` and `".)"` and can contain any code in the target language.

```
PRODUCTIONS
JavaFile = [PackageDecl] {Class<null> | ANY}.
PackageDecl = ...
Class<String parent> =
    "class" ident    (. String className = t.val;
                        System.out.println("found class " + className
                          + (parent != null)
                            ? " inside parent class " + parent
                            : ""); .)
    {ANY} "{" {Class<className> | BraceExpr | ANY} "}".
BraceExpr = ...
```

Listing 2.3: Augmented parser definition with a semantic action to print out the name of each class and its parent.

In Listing 2.3 we inserted a `parent` attribute to the `Class` NTS and a semantic action to print the current class name and its parent (if not null) to demonstrate this.

We heavily used these semantic actions and attributes in our full fuzzy grammar to gather all necessary metadata for our profiler about each Java file. Attributes in combination with recursive descent are especially useful, because we store scoped information on the Java method stack itself without additional data structures.

2.3 Other used software

To build our full working profiler we relied upon a few additional dependencies besides Coco/R and the predefined classes of the Java SDK.

javac the Java compiler command-line utility, used to compile our instrumented code into executable class files. It is part of every JDK installation. Without any kind of wrapper-library, we run it on the system command-line using a `ProcessBuilder`[3] instance. Calling it on any Java source file will automatically detect and locate other imported source files and compile them in the process.

java the command-line utility to execute Java programs. In the default mode of our profiler we execute the instrumented program automatically. For this we again use a `ProcessBuilder` to run this utility.

jUnit 5 [4] is the unit test framework we use to ensure the quality and stability of our profiler. We used it heavily during the construction of our grammar to verify continued correctness after any extension.

JavaFX [21] is a GUI framework to build modern graphical Java applications. We leveraged it to create a user interface for configuring and running our tool in a convenient and user-friendly way.

[3] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/ProcessBuilder.html`

[4] `https://junit.org/junit5/`

Gradle [5] is used to build and run our source code and to create release binaries. It is a popular cross-platform build tool that can automatically retrieve dependencies, specified in a declarative way.

jQuery [6] a widely known and used JavaScript library for traversal and modification of HTML documents, animations and event handling. We used it to do the initial highlighting in our annotated source code detail reports and to process mouse-over hovering events.

[5] https://gradle.org/

[6] https://jquery.com/

3 Architecture

3.1 General idea

There are five main steps involved to successfully profile a Java project using our tool. Figure 3.1 shows them in their sequential order.

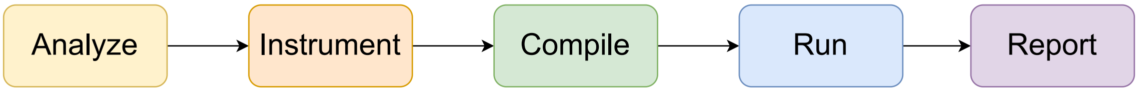


Figure 3.1: The five main steps of the profiler. First we parse all Java files and instrument them. Then we compile and run the instrumented program. Finally, we create a report from the results.

Analyze

First we use our generated parser program to extract metadata about all `*.java` files of a project. This data is then persisted to the file system and contains information about every file's classes, methods and code blocks. We use it to successfully instrument the program, to initialize the counters before execution and for the final report generation.

Instrument

In the next phase we create copies of each source file and insert counter statements at the beginning of code blocks. This is demonstrated on the example of a Fibonacci program in Listing 3.1.

```java
class Fibonacci {
  static int fib(int n) {
    if (n <= 1) {
      return n;
    }
    return fib(n - 1) + fib(n - 2);
  }
  public static void main(String[] args) {
    int N = Integer.parseInt(args[0]);
    for (int i = 1; i < N; i++) {
      System.out.print(fib(i) + " ");
    }
  }
}
```

Listing 3.1: A Fibonacci Java program

Using the line numbers and character positions from the metadata, we add additional counter increment instructions `__Counter.inc` into every executable code block. The same program, in its *instrumented* version, would look like the version in Listing 3.2.

```java
import __Counter;
class Fibonacci {
  static int fib(int n) {__Counter.inc(0);
    if (n <= 1) {__Counter.inc(1);
      return n;
    }
    return fib(n - 1) + fib(n - 2);
  }
  public static void main(String[] args) {__Counter.inc(3);
    int N = Integer.parseInt(args[0]);
    for (int i = 1; i < N; i++) {__Counter.inc(4);
      System.out.print(fib(i) + " ");
    }
  }
}
```

Listing 3.2: Instrumented version of the Fibonacci program

The argument to the `inc` method is the unique ID of each block. IDs are assigned to blocks in the order of their appearance. For more details see Sections 3.2 and 3.3.

Note, that we also have to import the `__Counter` class to use it. The import statement is inserted at the beginning of every Java file right after the package declaration. Section 3.3 describes this in more detail.

Numerous Java language constructs require special handling for this approach to work. For example, single-statement blocks can be used in common control flow statements such as `if`, `else`, `for` and `while`. If the block consists of only a single statement, it can be written without the opening and closing braces. This makes it impossible for us to add a counter statement at the beginning of this block, without wrapping the statement in a curly-brace block.

Again referencing our Fibonacci example, the contents of the `fib` method can be written in two lines of code as demonstrated in Listing 3.3:

```java
static int fib(int n) {
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
}
```

Listing 3.3: Version of method `fib` with a single-statement `if`

In this case we have to insert an opening `'{'` before our increment instruction and a closing `'}'` character after the statement ends. The code snippet in Listing 3.4 shows how the result would look like.

```java
static int fib(int n) {__Counter.inc(0);
  if (n <= 1){__Counter.inc(1); return n;}
  return fib(n - 1) + fib(n - 2);
}
```

Listing 3.4: Instrumented version of `fib` with a single-statement `if`

This also applies to arbitrarily nested single-statement blocks, where after instrumenting the inner blocks we still have to close the surrounding one with a closing brace.

More details on how we handled problematic Java language features can be found in Section 4.3.

Compile and execute

In this step of the program workflow, we simply use the `javac` and `java` command line utilities, installed with every JDK.

Calling the Java compiler on the instrumented copy of the main file (containing the main entry point) will automatically find and compile other imported source files. For successfully locating said files, we must follow the Java directory hierarchy rules [22]. See Section 3.5 for information about how the program output directory is structured.

After successful compilation we use the `java` [23] utility to run the compiled version of the main class. Any additional arguments to the profiler are forwarded to the executed program. At the end of the program execution the counter values are automatically written to the file system by the `__Counter` class (described in Section 3.3).

Report

Finally, we generate a report from the metadata of the original sources and the resulting counter values. The main page of the report is an overview page of all Java classes in the project. From there we can drill-down into any of the classes. Selecting one of them will forward us to a list of methods in this class, sorted by execution frequency. When trying to inspect a method, an annotated view of the source code is presented, giving insight into how often each statement was executed. Section 5.4 describes the navigation through the report in more detail.

3.2 Blocks and regions

Blocks

The most important class in our object model is the `Block` class. It contains fields holding information about the following key data:

- `id` - the unique id of a block (corresponds to the counter index)
- `beg` - the code position where the block begins (in the original file)

- `end` - the code position where the block ends
- `blockType` - the type of the current block (one of 11 types)
- `isSingleStatement` - whether or not the block is a single-statement (important for instrumentation)
- `class` - the Java class object this block is contained in
- `method` - the parent method object
- `parentBlock` - a pointer to the outer block when nested
- `innerBlocks` - the list of blocks directly nested inside this one
- `controlBreak` - the control flow break statement of this block (null by default and set by the parser if this block ends with `break/continue/return/throw/yield`)
- `labels` - the (list of) labels assigned to this block
- `codeRegions` - a list of regions in this block
- `incInsertOffset` - the char-position offset to insert the `__Counter.inc` statement at (relative to `beg`)
- `hits` - the number of times the block was entered during execution (will be set before report generation; extracted from resulting counts file)
- `innerJumpBlocks` - needed for region count calculation (see Section 4.4)

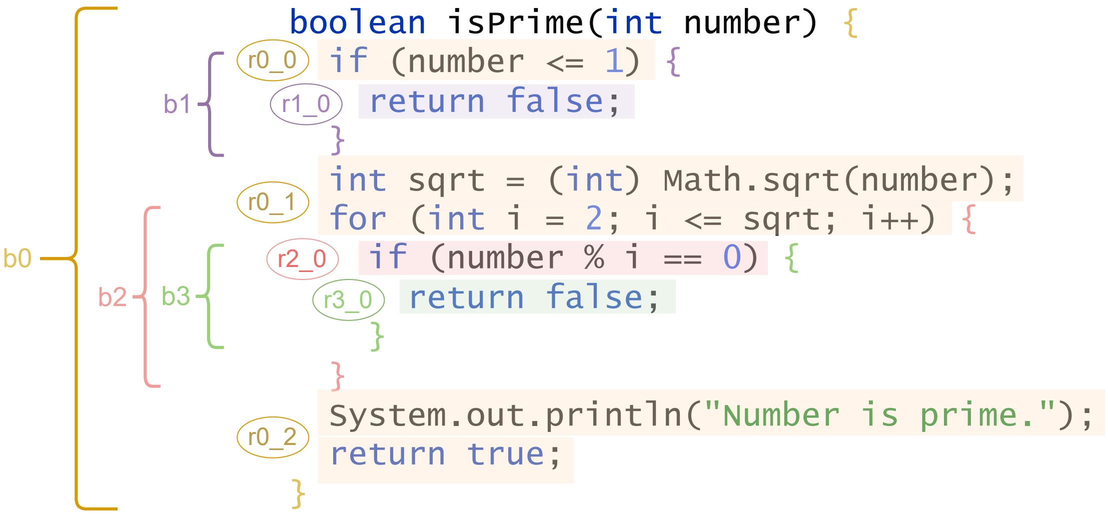
There are many types of blocks in Java programs. For example, class members are defined in the class block. Classes may feature a `static` initializer block. Methods and constructors have a method block containing executable statements and any number of nested code blocks inside it. Array definitions can have an initializer block, switch statements list their cases in a block. In case of the last two we do not add an increment statement.

To correctly instrument a program, we need to store the type of each found block in our metadata. Only blocks that are *executable* can and should be extended with a counter instruction.

The object model of our profiler currently defines a total of 11 block types. 10 for special kinds of blocks and one default `BLOCK` type. While there are a lot more block kinds to be considered, these suffice to correctly handle all use cases that the profiler was designed for. Other block kinds (such as array initializers) are skipped by the parser without being added to the metadata.

Regions

Each block contains 1..n `CodeRegion` objects and each region has exactly one parent block. A region spans over a range of statements, inside said block, that share the same hit count. In the simplest form a block contains exactly one region that spans over all its statements. When encountering an inner block, we end the current region and start a new region when we get back to the outer block. Inner blocks can contain control flow break statements such as `return`, that might reduce the effective hit-count of later statement regions (elaborated further in Section 4.4). Figure 3.2 shows the relationship between blocks and regions on an example method.



```
boolean isPrime(int number) {
  if (number <= 1) {
    return false;
  }
  int sqrt = (int) Math.sqrt(number);
  for (int i = 2; i <= sqrt; i++) {
    if (number % i == 0) {
      return false;
    }
  }
  System.out.println("Number is prime.");
  return true;
}
```

Figure 3.2: A simple `isPrime` method demonstrating how blocks and regions are related. The curly brackets depict spans of a *block* (named bx where x is the block id). Highlighted parts with a bubble next to them correspond to code *regions* (named rx_y where x is the parent block id and y the region id inside this block. In this example, only block b1 has more than one region.

For source code visualization and annotation in the report, regions are our most actively used resource. Only *they* contain the real effective hit-counts for each statement group. Regions start with the first *statement* of a block and end with the last one, purposely omitting the opening and closing braces. As curly braces of a block commonly do not contain a statement in the same line, this reduces visual overhead and confusion. Also see Section 5.4 for more information on how the report is structured.

3.3 Counter class

The `auxiliary.__Counter` class is used for storing and updating block counter data. It has to be imported in all instrumented source files and must exist at the expected path (relative to the source directory) during compilation. Its compiled `.class` version will be extracted automatically from the profiler's JAR file before compilation. The "__" prefix is used to avoid naming conflicts with any possibly preexisting "Counter" classes of the project to profile.

```java
public class __Counter {
  static {
    init(".profiler/metadata.dat");
    Runtime.getRuntime().addShutdownHook(
        new Thread(() -> save((".profiler/counts.dat"))));
  }
  private static void init(String fileName) { /* init array */ }
  private static void save(String fileName) { /* save counts */ }

  private static long[] blockCounts;
  public static void inc(int n) { blockCounts[n]++; }
  synchronized public static void incSync(int n) { inc(n); }
  ...
}
```

Listing 3.5: Source code excerpt of the `__Counter` class

Listing 3.5 previews the class's most important members. Only the increment methods are accessible from outside the class, the rest is handled internally. Before the first use, when the class is loaded, the class's `static` block calls `init` to initialize the `blockCounts` array with a size that is equal to the amount of blocks found in the analyzed project. This number is read from the metadata file. Additionally, a shutdown hook [24] is added to the runtime to save block hit-counts when the VM shuts down. The 64-bit integer array represents the hit-counts of each block. By indexing the array with a block's `id`, the value for how often this block was entered can be accessed.

If the same block is accessed very frequently by different threads in a multi-threaded application, the final counts might not be accurate. Therefore instrumentation can be done with atomically incrementing counters (see Section 5.2 for details on the `--synchronized` option). In this case, instead of the `inc` method, the `incSync` method will be inserted at the beginning of each block. This will ensure correct counter values, but will lead to a significant performance slowdown.

After conducting a series of run time experiments (evaluated in Section 6.3), we came to the conclusion, that our approach of locking the `__Counter` class for each `incSync` invocation (by use of the `synchronized` modifier) can result in a severe overhead for highly concurrent programs. Therefore we improved our initial approach by adopting the `AtomicLongArray`[1] of the Java standard library. We now create two arrays of the same size in the `init` method. One is used by `inc` as before while `incSync` uses the `AtomicLongArray`. When saving the results, we write the *sum* of both arrays to the `counts.dat`. In Listing 3.6 the updated version of the class can be found. A comparison of the run-time impact of these two approaches can be found in Section 6.3.

```
public class __Counter {
  private static void init(String fileName) { /* init both arrays */ }
  private static void save(String fileName) { /* save sum of counts */ }

  private static long[] blockCounts;
  private static AtomicLongArray atomicBlockCounts;

  public static void inc(int n) { blockCounts[n]++; }
  ~~synchronized~~ public static void incSync(int n) {
    atomicBlockCounts.incrementAndGet(n);
  }
  ...
}
```

Listing 3.6: Usage of the AtomicLongArray in the `__Counter` class instead of methods with the `synchronized` modifier

[1] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/atomic/AtomicLongArray.html

3.4 Java fuzzy parsing

The ATG for generating the Scanner and Parser files is using so-called "fuzzy" parsing. We skip non-relevant parts of the input file using Coco/R's ANY symbol [6]. This way our grammar is kept small in size and is more robust and maintainable against many updates and extensions to the Java Language Specification [18]. The ATG consists of a reduced set of NTS, that cover the most important aspects of the Java 17 syntax tailored for our main use case (finding the begin and end position of blocks).

Some of the non-relevant tokens we skip with the ANY symbol are:

- access modifiers (public, private, ...)
- interfaces that a class implements (or superclasses)
- class-level constants and member variables
- generic type definitions within angle brackets <Type1<Type2,...>, ...>
- array initializer blocks (starting with "{"), but we do not insert a counter here
- a method's argument list (within the parentheses)
- remaining tokens in a GenericStatement up to the semicolon
- the switch-case label(s) or constant(s) before the colon

Our JavaFile.atg completely omits the common Expression and Term productions. The language specification defines complex variants for expressions, that are not essential for finding blocks. Instead we attempted to capture all these statements in one short GenericStatement NTS. [2]

[2] for the entire grammar see https://github.com/matwoess/java-profiler/blob/main/profiler-tool/src/main/java/tool/instrument/JavaFile.atg

3.5 Directory structure

Every step of the workflow adds new files to the output directory. Subsequent steps depend on the previous files. By default, all output of the profiler is written to the (hidden) `.profiler` subdirectory inside the current working directory. The tool will automatically create folders that do not exist yet. Results of previous runs are removed before writing new files.

Based on the directory structure of the minimal sample project in Listing 3.7, we will now describe which files are created when and where inside the output directory.

```
project
├── Main.java
└── util
    └── Helper.java
```

Listing 3.7: Sample project directory structure

Figure 3.3 illustrates the changes and additions of every step of the profiling workflow (the five steps described in the “General idea” of Section 3.1). Executing the program from within the `project` directory will store all files in the `project/.profiler` subdirectory.

In the first step, all Java files are found and analyzed. The resulting project metadata will be stored in `metadata.dat` directly in the output directory. At the same time, an instrumented copy of every Java file is created inside the `.profiler/instrumented/` subdirectory. The original structure of the project is replicated. Thus, the instrumented version of `Helper.java` will also be located in a `util` subfolder. At the root of the `instrumented` directory, an `auxiliary` package subfolder is created containing only the `__Counter.class` file. It is important that `__Counter` is not in the default package to be able to import it anywhere.

Next, all instrumented sources are compiled and their `.class` files are written to a new `classes` directory. This is done to declutter the contents of the `instrumented` directory and to separate functionally different files. The `auxiliary/__Counter.class` file is also present within the `classes` directory to successfully load it during execution with a single classpath target folder.

After executing the instrumented and compiled version of the project, the `counts.dat` results file should automatically be written to the `.profiler` directory.

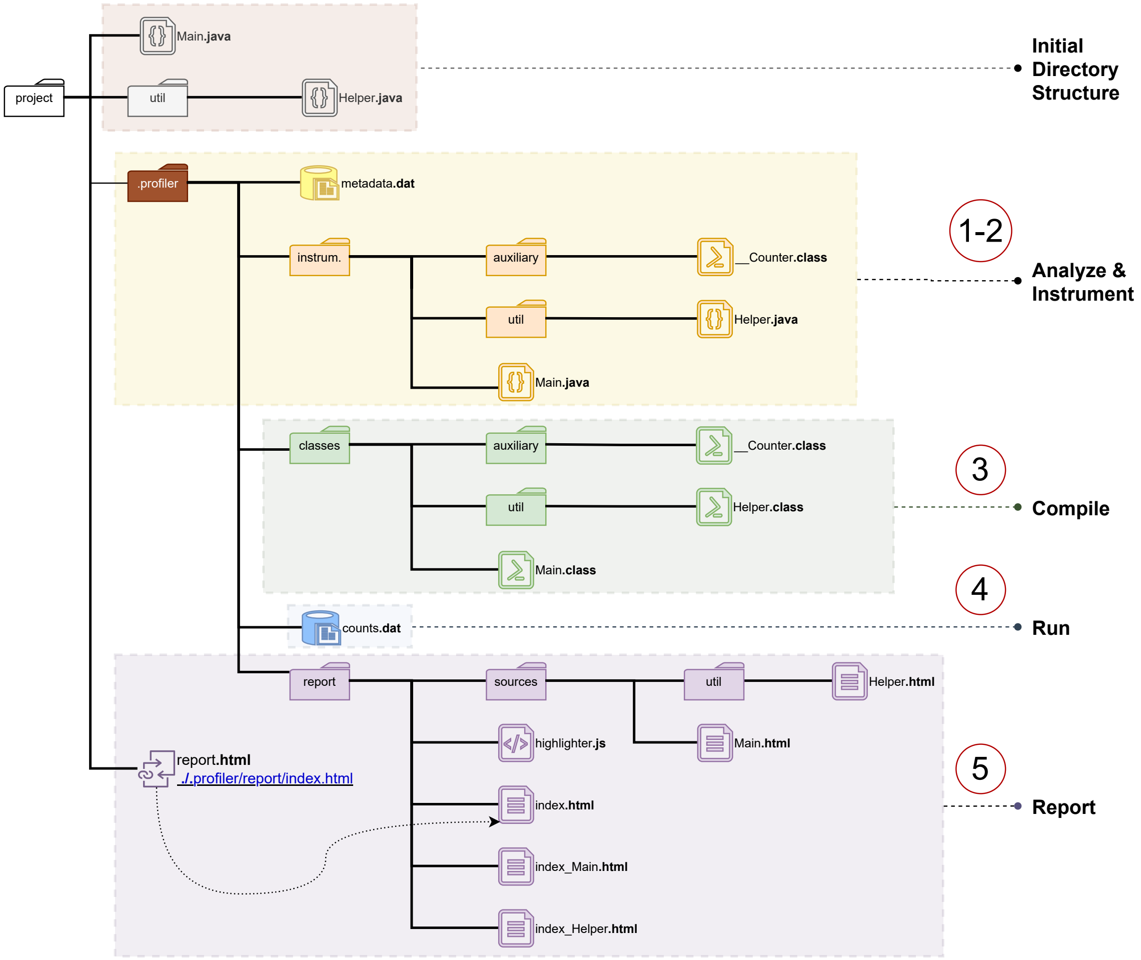


Figure 3.3: Visualization of the output directory structure. Each step is highlighted in a colored box. At the top (in light red), the directory `project` contains the already existing files. The `.profiler` subdirectory and all files in yellow will be added in steps 1 and 2. The green area is added in step 3, the counts are written in step 4 (blue) and all report related files in step 5 (in violet).

The `metadata.dat` and `counts.dat` files are then used, together with the original sources, to generate the HTML report in the `report` folder. In its `sources` subfolder, the annotated code listings of each original Java file can be found and used for detailed inspection. The relative paths of the original package structure are again replicated. Outside the `sources` subfolder are the index files and the `highligther.js` helper script (for dynamic highlighting and mouse-hover event handling). The `index.html` is the main report page, listing all the Java classes of the profiled project. At the same hierarchical level there are additional index files (`index_`*ClassName*`.html`) index files for each Java class. These are the method overviews that can be navigated to from the main index.

Finally (if the system allows it), a symbolic link `report.html` is generated in the current working directory. This link points to `./.profiler/report/index.html` for quickly opening the report without navigating into the (hidden) `.profiler` output directory.

4 Implementation

This section goes a little deeper into details about how the Java profiler was implemented. We elaborate on how the project is structured, which classes are responsible for what and how special cases are handled.

4.1 Steps

Analyze and Instrument

When executing the profiler in the default run mode (other modes can be found in Section 5.2), we always need to specify the main class file of the project. This is required for our logic of automatic compilation and execution of the instrumented code. Specifying the sources directory will find all contained Java files recursively and will instrument them, too.

First, a single `Instrumenter` instance is created and the list of java files is passed to it. The `Instrumenter` creates a separate Coco/R parser for every file to analyze. Using semantic actions in the productions of our ATG, we insert `ParserState` methods into the generated `Parser` code. The `ParserState` is responsible for building the model for every file during parsing. `JavaFile` objects store the full metadata for every Java file.

Finally, the Instrumenter exports all metadata to the file system. This is done by serializing the list of `JavaFile` objects. Therefore, all classes of the `tool.model` package implement the `Serializable` interface. As a first value in the metadata before the Java files, we write the total amount of blocks found (in the entire project). This helps speeding up the initialization of the counters array at run time. In `__Counter.init` we just read the first value and create an array of this size. During the profiling run we do not require any information about parsed files.

Compile and run

After all Java files are instrumented and the metadata is exported, we instantiate a `Profiler` object. It is responsible for compilation, execution and the report generation. To compile the instrumented sources to `.class` files and to output them to the `classes` directory, the command from Listing 4.1 is issued.

```
$ javac -cp .profiler/instrumented \
        -d .profiler/classes \
        .profiler/instrumented/<relPathToMain>
```

Listing 4.1: The command line that is executed to compile instrumented sources and to output the compiled files to the classes directory.

Each `JavaFile` object stores the original file's relative path to the sources directory. This path is used to locate its associated files in the `instrumented` and `classes` directories. Once the class files are compiled, we use a `ProcessBuilder` to execute the command line in Listing 4.2. Here the `".java"` extension is removed from the relative path to reference the class containing the main entry point. Additional arguments to the profiler are appended to this command.

```
$ java -cp .profiler/classes .profiler/classes/<relPathToMain_noExt> ARGS
```

Listing 4.2: The command line to execute the instrumented main file and to pass arguments

For compilation and execution, we do not change the current directory. Relative path arguments and hard-coded path strings in the source code would still work as expected.

At this stage the profiler waits for the program to terminate. Once completed, the counts are automatically written by the `__Counter` shutdown hook. We can now create the report.

Report generation

The report is also generated based on the list of `JavaFile` objects from the metadata. All information about classes, methods, blocks and regions already exist. The only missing data is the number of hits of every block. Every instance of class `Block` has a member `hits`. It is transient (will not be serialized) and is populated right before the report generation with the counter values from `counts.dat`.

First, the list of *top-level* Java classes is written to the `index.html` as a sorted HTML table. Sorting is done, based on the aggregated method invocations. As inner classes are not listed here, their counts are included in the sum of its top-level class. Next, we loop over all `JavaFile` objects. Files without a single detected block are skipped. We then generate the coverage report and a method index for all top-level classes of each file. Method indexes include the methods of inner classes in the sorted table.

The source file coverage report is a table of three columns: region hit counts, line numbers and source code (see Section 5.4 for more details). Highlighting is done by wrapping code in `span` elements and assigning CSS classes to it. Every source code report file imports the `highlighter.js` script. Using jQuery it updates the background color, depending on CSS classes assigned to the `span` elements. Table 4.1 lists the possible span class types and what they represent.

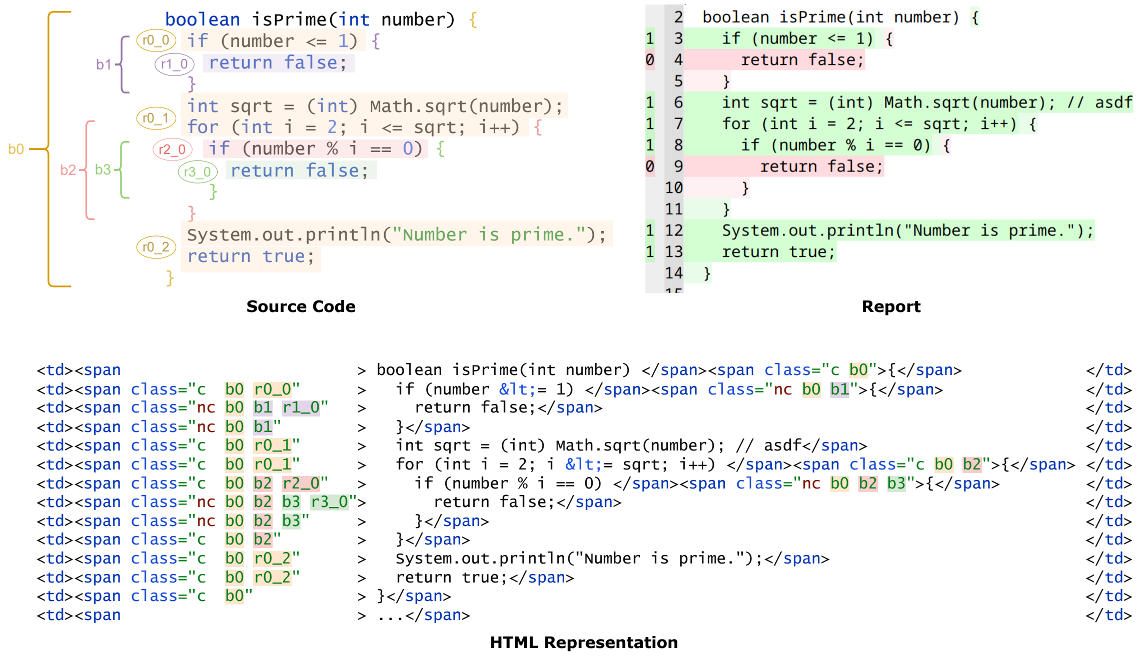
Class	Meaning	Effect
c	covered	Will be highlighted in green
nc	not covered	Will be highlighted in red
b_x	block with id x	All spans inside the x^{th} block have this class
r_{x_y}	y^{th} region in block x	All spans in this region have this class

Table 4.1: List of span element classes in the report and what they are used for

Because we wrap the entire source code in an HTML table (each line is one table row), we need to end the current `span` when a line ends and start it anew in the next line, to produce valid HTML. We extended the sample from Figure 3.2 to show how block tags and region tags are represented in the report files. Refer to Figure 4.1 for this comparison.

Most regions start from the *beginning* of the line (not from the first statement) due to a post-processing step. We eliminate empty spans and spans at the beginning of each line containing only whitespace. This leads to a smaller file size, more consistent highlighting and less visual overhead.

When hovering over any `span` in the source code, a jQuery `mouseenter` event will be fired. We then extract the last class inside the current `span` element and check whether it is a region or a block. If it is a region, the second-last class is also determined, which will always be a block class. We use this class to update the `'background-color'` CSS values of *all* span tags containing the currently selected block or region class. For example, if we



```
boolean isPrime(int number) {
    if (number <= 1) {
        return false;
    }
    int sqrt = (int) Math.sqrt(number);
    for (int i = 2; i <= sqrt; i++) {
        if (number % i == 0) {
            return false;
        }
    }
    System.out.println("Number is prime.");
    return true;
}
```

Source Code

```
  boolean isPrime(int number) {
1   if (number <= 1) {
0     return false;
    }
1   int sqrt = (int) Math.sqrt(number); // asdf
1   for (int i = 2; i <= sqrt; i++) {
1     if (number % i == 0) {
0       return false;
      }
    }
1   System.out.println("Number is prime.");
1   return true;
  }
```

Report

```
<td><span                        > boolean isPrime(int number) </span><span class="c b0">{</span>            </td>
<td><span class="c  b0 r0_0"     >   if (number &lt;= 1) </span><span class="nc b0 b1">{</span>              </td>
<td><span class="nc b0 b1 r1_0"  >     return false;</span>                                                 </td>
<td><span class="nc b0 b1"       >   }</span>                                                               </td>
<td><span class="c  b0 r0_1"     >   int sqrt = (int) Math.sqrt(number); // asdf</span>                      </td>
<td><span class="c  b0 r0_1"     >   for (int i = 2; i &lt;= sqrt; i++) </span><span class="c b0 b2">{</span> </td>
<td><span class="c  b0 b2 r2_0"  >     if (number % i == 0) </span><span class="nc b0 b2 b3">{</span>        </td>
<td><span class="nc b0 b2 b3 r3_0">       return false;</span>                                               </td>
<td><span class="nc b0 b2 b3"    >     }</span>                                                             </td>
<td><span class="c  b0 b2"       >   }</span>                                                               </td>
<td><span class="c  b0 r0_2"     >   System.out.println("Number is prime.");</span>                          </td>
<td><span class="c  b0 r0_2"     >   return true;</span>                                                     </td>
<td><span class="c  b0"          > }</span>                                                                 </td>
<td><span                        > ...</span>                                                               </td>
```

HTML Representation

Figure 4.1: Exemplary demonstration of how blocks and regions are currently tagged inside the HTML code of report source files. On the top-left we have the example from Figure 3.2. To the right of it we see how it would be displayed in the report. On the bottom we see the HTML code for how this is represented in a table.

hover over the `println` statement region in Figure 4.1, all spans containing the `b0` class will be highlighted. Additionally, the region's `'font-weight'` will be set to `bold`.

Whenever we move our mouse away from a span, jQuery will recieve a `mouseleave` event. This is used to reset all colors and font weights to the default values. The source report file will again look like when we first opened it.

The popup for indicating how often a region or block was entered is realized by adding a `title` attribute [25] to every `span` in the code. This will lead to a tooltip opening when hovering over the HTML element.

4.2 Classes

At the time of writing our software repository consisted of more than 90 Java class files (including test classes), distributed over three separate Java modules. Hierarchical diagrams were drafted to create a rough overview over the project. The three Java modules and their connections are shown in Figure 4.2.

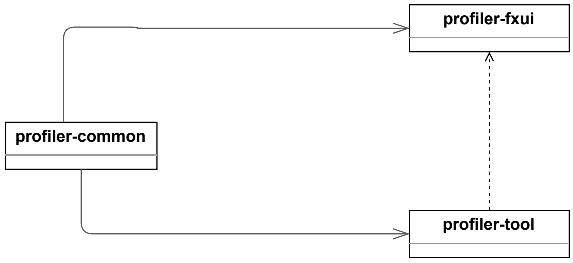
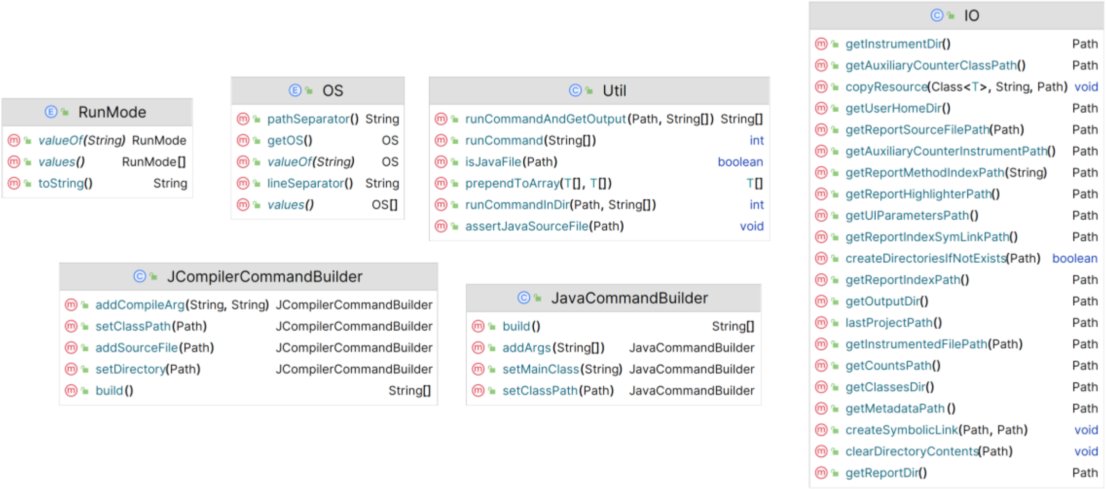


Figure 4.2: The project's Java module dependencies. The `fxui` and `tool` modules both depend on the `common` module. `fxui` only uses the `tool` when executing its main method, but does not depend on any other implementations of it.

The `common` module contains universal utility methods for both `tool` and `fxui`, related to the operating system or filesystem operations. Its class diagram, with available public methods, can be found in Figure 4.4.

The `tool` module contains the most important classes that handle all the instrumentation and profiling logic. Showing all its classes and connections would result in an overloaded picture. Therefore a stripped down overview, including the most important Java classes and how they are used, can be inspected in Figure 4.3.

The class diagram for the `fxui` module is excluded for brevity. The GUI is subject to change and might be fully replaced by alternatives like IDE integrations in the future.



RunMode	
valueOf(String)	RunMode
values()	RunMode[]
toString()	String

OS	
pathSeparator()	String
getOS()	OS
valueOf(String)	OS
lineSeparator()	String
values()	OS[]

Util	
runCommandAndGetOutput(Path, String[])	String[]
runCommand(String[])	int
isJavaFile(Path)	boolean
prependToArray(T[], T[])	T[]
runCommandInDir(Path, String[])	int
assertJavaSourceFile(Path)	void

IO	
getInstrumentDir()	Path
getAuxiliaryCounterClassPath()	Path
copyResource(Class<T>, String, Path)	void
getUserHomeDir()	Path
getReportSourceFilePath(Path)	Path
getAuxiliaryCounterInstrumentPath()	Path
getReportMethodIndexPath(String)	Path
getReportHighlighterPath()	Path
getUIParametersPath()	Path
getReportIndexSymLinkPath()	Path
createDirectoriesIfNotExists(Path)	boolean
getReportIndexPath()	Path
getOutputDir()	Path
lastProjectPath()	Path
getInstrumentedFilePath(Path)	Path
getCountsPath()	Path
getClassesDir()	Path
getMetadataPath ()	Path
createSymbolicLink(Path, Path)	void
clearDirectoryContents(Path)	void
getReportDir()	Path

JCompilerCommandBuilder	
addCompileArg(String, String)	JCompilerCommandBuilder
setClassPath(Path)	JCompilerCommandBuilder
addSourceFile(Path)	JCompilerCommandBuilder
setDirectory(Path)	JCompilerCommandBuilder
build()	String[]

JavaCommandBuilder	
build()	String[]
addArgs(String[])	JavaCommandBuilder
setMainClass(String)	JavaCommandBuilder
setClassPath(Path)	JavaCommandBuilder

Figure 4.3: The class diagram of the `profiler-tool` module. For each Java class, the public methods are listed.

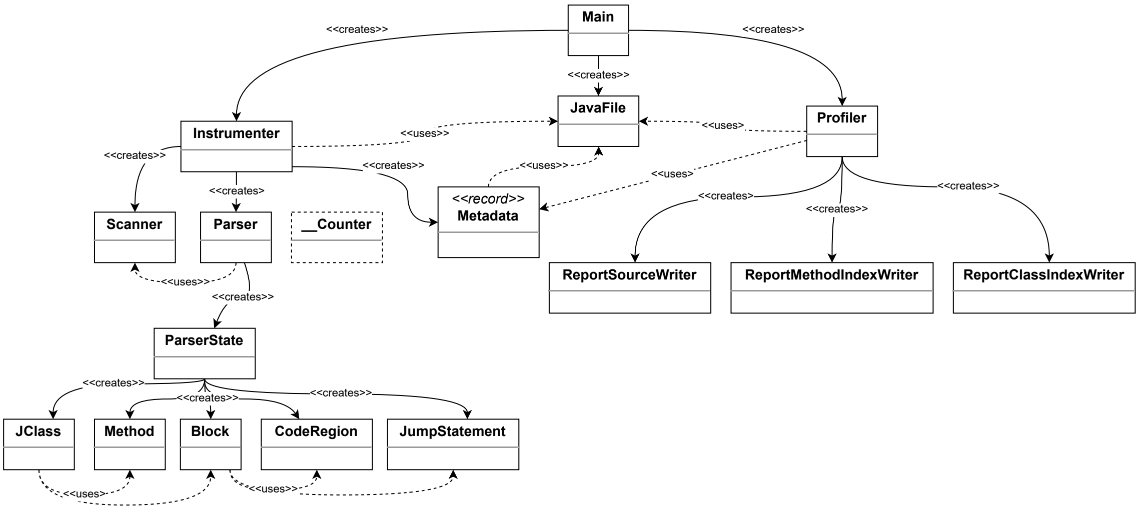


Figure 4.4: The simplified class creation-and-usage diagram of the `profiler-tool` module. Only the most important classes and connections are shown. First, the `Main` class creates `JavaFile` objects and passes them to a new `Instrumenter`. The `Instrumenter` creates a `Scanner` and `Parser` per Java file. Each `Parser` has an associated `ParserState` that builds the object model. The `Instrumenter` then stores this data in the `JavaFile` objects, that will be persisted as the `Metadata` record. Finally, the `Profiler` uses the `Metadata`, together with counts read from the file system, to generate the report. The three report writers share a common `AbstractReportWriter` parent class.

4.3 Special handling of language features

Recent Java versions support many special language constructs that are non-trivial to instrument with counter statements. In this section we elaborate on how we handled the most prominent ones.

Single-statement blocks

One such example are single-statement blocks with omitted braces. As previously mentioned in Section 3.1, we have to add curly braces before our inserted `__Counter.inc` statement and after the original statement ends. In case of nested single-statement blocks, we have to do this multiple times. For this purpose, each block has a boolean member for whether or not it is a single-statement block. In the grammar, all statement productions that support this short-hand use the `BlockOrSingleStatement` NTS instead of `Block`.

The method in Listing 4.3 defines four single-statement blocks. Blocks in lines 3-5 are nested in each other. How this method would look like after instrumentation can be seen in Listing 4.4.

```
1 boolean containsZero(int[][] array) {
2   if (array == null) return false;
3   for (int i = 0; i < array.length; i++)
4     for (int j = 0; j < array[i].length; j++)
5       if (array[i][j] == 0) return true;
6   return false;
7 }
```

Listing 4.3: A method with nested single-statement blocks

```
1 boolean containsZero(int[][] array) {__Counter.inc(261);
2   if (array == null){__Counter.inc(262); return false;}
3   for (int i = 0; i < array.length; i++){__Counter.inc(263);
4     for (int j = 0; j < array[i].length; j++){__Counter.inc(264);
5       if (array[i][j] == 0){__Counter.inc(265); return true;}}}
6   return false;
7 }
```

Listing 4.4: Instrumented version of the method with nested single-statement blocks

Overloaded constructors

Java supports multiple "overloaded" constructors in the same class. If we use `super()` or `this()` invocations, the language enforces that it must be the first statement in the method [26]. The example code in Listing 4.5 shows, how a naive instrumentation could be done. Compilation of this code will fail.

```java
class SmallDog extends Dog {
  public SmallDog(String name , int age) {__Counter.inc(3);
    super(name , age);
    size = Size.SMALL;
    super.speak();
  }
  public SmallDog(String name , int age , Size s) {__Counter.inc(4);
    this(name , age);
    this.size = s;
  }
  ...
}
```

Listing 4.5: Constructor methods that are incorrectly instrumented. A statement preceding `this()` or `super()` is not allowed.

The corrected version can be found in Listing 4.6, where we insert our counter increment statement right *after* the call to other constructors. While it might not be fully correct to only increment a method's counter after it already executed code, it is still reasonable and the best we could come up with.

```java
class SmallDog extends Dog {
  public SmallDog(String name , int age) {
    super(name , age);__Counter.inc(3);
    size = Size.SMALL;
    super.speak();
  }
  public SmallDog(String name , int age , Size s) {
    this(name , age);__Counter.inc(4);
    this.size = s;
  }
  ...
}
```

Listing 4.6: Corrected instrumentation of constructor methods using `incInsertOffset`.

If the method block has the block type `CONSTRUCTOR` and we encounter a `super()` or `this()` invocation, we store the character offset between the beginning of the block and the `";"` after the invocation. Therefore, blocks have an `incInsertOffset` member, that will be considered during instrumentation. This way, the begin position of the block will stay accurate in our metadata.

Anonymous and local classes

Java allows the declaration of classes inside methods (as demonstrated in the unit test sample of Listing 6.1). We can simply declare a named class anywhere inside a block. These local classes [27] are scoped, meaning that they will effectively only be available inside this block. Additionally, there are so-called anonymous classes [28]. They behave similar to local classes, but have no name and are instantiated automatically.

Both of these constructs made it harder to build a consistent object model during source code analysis. The definitions are actually subclasses of the parent class and should not be considered part of the method we currently parse. Also, they can contain further methods and arbitrarily nested (anonymous or local) class definitions. After exiting local class declarations, we need to restore the previous method as the "current" one. For this, the `ParserState` contains a `Stack` of methods, onto which we push the current one when encountering class declarations inside methods.

Switch statements and switch expressions

Switch statements need a lot of special handling due to their abnormal syntax. The switch's curly-brace block itself is not executable and requires no counter statement. Case blocks start right after the `":"` and do not end with a `"}"`. Multiple cases can be combined into one common block, if the previous case contains no statements. A `"break"` can be omitted to achieve a fall-through.

Since the release of Java 14 the language supports so-called switch *expressions*. They behave like the common switch statement, but always return a value. The result can directly be assigned to a variable or returned by a method. Additionally, with the JEP 261 [29], the arrow-case labels were introduced. They can be used as a switch case short-hand without the need for a break keyword or a possible fall-through. Also, the new yield keyword was introduced, to instantly yield a value from the expression, similarly to return in methods. We can use switch statements, switch expressions and standard case labels or arrow-case labels interchangeably. A switch expression can use "case L: yield val;" and a switch statement can use "case L1, L2, L3 -> foo();".

An example of the new switch expressions using arrow-cases can be found in Listing 4.7.

```
StatusCode statusCode = ...;
int sc = switch (statusCode) {
  case OK -> 200;
  case UNAUTHORIZED -> 401;
  case FORBIDDEN -> 403;
  case NOTFOUND -> { yield 404; }
  default -> throw new RuntimeException("invalid status code");
};
```

Listing 4.7: Exemplary usage of a switch expression with arrow-cases. Three cases yield a value implicitly, one has a block using the yield keyword, and the default branch throws an exception.

Similar to instrumenting single statements, this language construct requires adding braces around the value expression. Additionally, we have to insert a yield between our counter statement and the value. If the case already has braces, we do not have to add a yield, as one must already be provided by the programmer. Another exception are brace-less blocks that throw an exception. Although we have to add the braces and our counter statement, we *must not* add a yield before the throw.

In Listing 4.8 we show what must be inserted in order to get correct counts and to be able to compile the program successfully.

```
StatusCode statusCode = ...;
int sc = switch (statusCode) {
  case OK ->{__Couter.inc(5); yield 200;}
  case UNAUTHORIZED ->{__Couter.inc(6); yield 401;}
  case FORBIDDEN ->{__Couter.inc(7); yield 403;}
  case NOTFOUND -> {__Couter.inc(8); yield 404; }
  default ->{__Couter.inc(9); throw new RuntimeException();}
};
```

Listing 4.8: Instrumentation of switch expressions with arrow-case blocks and one throws-branch

To handle all variations of instrumentation correctly, we introduced the block types `SWITCH_STMT`, `SWITCH_EXPR` and the case block types `COLON_CASE` and `ARROW_CASE`. Every block has a pointer to its parent. During instrumentation, we exclude switch blocks themselves, and for case blocks we determine the surrounding switch block type. Only for switch expressions we add a `yield`. To handle the special case of `throw` branches we need to check whether the parent block type matches `SWITCH_EXPR`, the block satisfies `isSingleStatement` and ends with a `throw` statement.

Lambda expressions

Since Java Version 8, lambda expressions [30] are supported. They are similar to anonymous classes, but can be more concise, expressing instances of single-method classes (usually interfaces). They start with a parameter list followed by an arrow `"->"` and a method block. They are trivial to instrument if the block has braces. If the right-hand side is a brace-less expression, instrumentation can be tricky.

We first attempted to handle them like the singe-statement switch expression arrow-case blocks. This would mean that we have to add the curly braces and insert a `return` before the value expression (and a `";"` after it). The problem with this approach is that not every lambda expression returns a value. The compiler infers the type of a lambda by context information and determines the return type. Having only access to the source code it is practically impossible to know, whether or not we should add a `return` before the contents of a brace-less lambda block.

A demonstration of correctly instrumented lambdas can be found in Listing 4.9.

```java
Function<Integer, Double> divideBy3;
divideBy3 = num ->{__Counter.inc(0); return num / 3.0;};
int[] array = new int[]{1, 2, 3, 4, 5, 6};
Arrays.stream(array)
  .map(x ->{__Counter.inc(1); return x*2;})
  .peek(x ->{__Counter.inc(2); System.out.println(x);})
  .filter(x ->{__Counter.inc(3); return (x > 5);})
  .reduce((acc, x) ->{__Counter.inc(4); return (acc) + (x);})
  .ifPresent(x ->{__Counter.inc(5); ...;});
```

Listing 4.9: Correct source code instrumentation of brace-less lambdas (hard to implement)

Notice that in line 6, where we simply print out `x`, we *must not* insert a `return`. It would result in a compilation error. Because we cannot infer the return type of an arbitrary lambda, we had to find another way.

Our final approach[1] was to wrap every lambda inside another lambda, which is either a `Runnable` or a generic `Supplier` with a single type parameter. We pass this wrapped lambda as an argument to a special `incLambda` method of our `__Counter` class. There are two implementation variants shown in Listing 4.10. By overload resolution, the compiler will automatically assign the correct variant when compiling our instrumented copies.

```java
public static void incLambda(int n, Runnable method) {
  __Counter.inc(n);
  method.run();
}
public static <T> T incLambda(int n, Supplier<T> function) {
  __Counter.inc(n);
  return function.get();
}
```

Listing 4.10: The `__Counter` methods to handle any wrapped lambda as an argument. The compiler will choose the correct variant by overload resolution. First we increment the block counter, then we execte `run()` for `Runnable`s or return the value of `get()` for `Supplier`s.

Using this approach, we neither have to add `return` keywords nor curly-braces. Based on the example from above, we show the new way of instrumentation in Listing 4.11.

```java
Function<Integer, Double> divideBy3;
divideBy3 = num ->__Couter.incLambda(0, () -> num / 3.0);
int[] array = new int[]{1, 2, 3, 4, 5, 6};
Arrays.stream(array)
  .map(x ->__Couter.incLambda(1, () -> x*2))
  .peek(x ->__Couter.incLambda(2, () -> System.out.println(x)))
  .filter(x ->__Couter.incLambda(3, () -> (x>5)))
  .reduce( ... )
  .ifPresent( ... );
```

Listing 4.11: New variant of instrumenting brace-less lambdas using wrapping and `incLambda`

[1] Thanks for the hint by Julian Garn

Text blocks

Since the release of Java 15 it is possible to define multi-line string literals (see JEP 378 [31]). They are called text blocks and are enclosed inside triple quotes. While strings are not a target of instrumentation, we struggled to define a valid scanner definition to correctly parse both standard string literals and text blocks as the same `string` token type. Missing a single escape character will lead to parsing errors and misinterpreted tokens.

In Listing 4.12 an example of a multi-line text blocks can be found. Using correct escaping, these text blocks can contain nested text blocks inside them. Also, three sequential `"""` do not necessary terminate the token if one of them is escaped. We also have to make sure that `""` is not recognized as an empty string token if a third semicolon follows.

```
String emptyString = "";
String usualString = "Hello\n\"World\"";
String nestedTextBlocks = """
  multiline-text
  can contain ""strings"", \"escaped strings\"
  and nested ""\"
    Text blocks!
    within ""\\\\"
      "nested(!) text blocks"
    "\\\\""
    does not end on \"""
""";
```

Listing 4.12: Demonstration of different possible string literals, including a multi-line text block that contains nested text blocks

To correctly parse all possible types of string literals we tried many different scanner definitions. The final definition for our scanner automaton can be found in the appendix as Listing 1.

4.4 Control flow breaks

The last major difficulty we faced were statements that break the control flow of the program. In addition to conditional statements (`if-then-else` and `switch`), there are loop statements (`for`, `while`, `do-while`) and control flow breaks (`break`, `continue`, `return`).

Other examples are the `yield` and the `throw` statement. Conditional statements and loop statements have nested blocks and don't need any special handling. What we had to handle, though, were control flow breaks, `yield` and `throw`. They appear as the last statement in blocks and lead to skipping the execution of subsequent parts of a program (not necessarily contained in the same block).

Due to our instrumentation approach of inserting a single counter statement at the *beginning* of every block, the data might not reflect the true counts of every statement, if control flow breaks are used. To demonstrate the problem we refer to Listing 4.13.

```
1 275      | static int fib(int n) {
2 275      |   if (n <= 1)
3 275  142 |     return n;
4 275      |   return fib(n - 1) + fib(n - 2); // 275-142 = 133
5          | }
```

Listing 4.13: Fibonacci example with counter values on the left side. We know the counter values of the blocks, but the execution count of line 4 is not inherently clear.

We have two counters for this method. One for the method block and one for the if-statement block. Line 4 shares the same counter as line 2. The only way to know how often line 4 was executed, is to do the math: Since line 3 was executed 142 times, line 4 was actually reached only $275 - 142 = 133$ times. This is trivial in this small example, but once a statement depends on many different control flow break statements, deeper inside a nested hierarchy, the calculation is hard to do manually. Moreover, `break` and `continue` can optionally jump to a *label* adding even more complexity.

We came up with two main approaches for showing exact execution counts for each code line.

Variant 1: Insert additional counters

We determine the continuation point in an outer block after a control flow break and insert an additional counter there. By incrementing this counter only when execution actually gets to this point, we know exactly how often the following statements were executed. We sketch this approach in Listing 4.14.

```
for ( ... ) {__Counter.inc(23);
  if ( ... ) {__Counter.inc(24);
    break;
  } else if ( ... ) {__Counter.inc(25);
    continue;
  }__Counter.inc(26);
  x += 5;
  if ( ... ) {__Counter.inc(27);
    return;
  }__Counter.inc(28);
  System.out.println(...);
}
```

Listing 4.14: First approach of handling control flow breaks by inserting additional counter statements. The additional counters are highlighted in darker gray.

While this is a valid and reliable approach, it requires the insertion of many more counters. This will further slow down the instrumented program during execution by causing more counter increment overhead. We also lose the unique mapping between a block's id and the counter index. Currently we have exactly as many counters as there are blocks and the counters array has the size of the number of blocks. Mapping back counter values to block IDs would be ambiguous with this approach. We could start a new block at these continuation points, but it would distort our metadata and create difficulties with visualization in the report (because we highlight the whole block on hover).

Variant 2: Regions and calculated counts

As shown in the Fibonacci sample of Listing 4.13, we actually know how often statements are executed by how often blocks with a control flow break were entered. An experienced programmer can quickly infer the true count by looking at the counter values of other blocks. We tried to map this knowledge into an algorithm.

Refer to the modified Listing 4.15 with no additional counters. The counters for continuation points are calculated using only other block counters and the model structure itself. This way, we insert less counters and thus avoid additional run-time overhead. Furthermore, the calculation of the effective counts is done only during report generation. This approach induces no slowdowns additional for counter increments or counter outputs to the file system. It is, however, harder to get right than the first variant.

```
for ( ... ) {__Counter.inc(23);
  if ( ... ) {__Counter.inc(24);
    break;
  } else if ( ... ) {__Counter.inc(25);
    continue;
  }
  x += 5; // c23 - c24 - c25
  if ( ... ) {__Counter.inc(26);
    return;
  }
  System.out.println(...); // c23 - c24 - c25 - c26
}
```

Listing 4.15: Second approach of handling control flow breaks by calculating the effective region counts from other block counts.

To implement this logic, we introduced the CodeRegions into the object model. A region stores its parent block for the base hit-count and a list of pointers to the dependent control flow break blocks. The final calculation can be done using the equation:

$$h_{region} = h_b - \sum_{i=0}^{D} h^i_{dep} \tag{4.1}$$

Where h denotes hit-counts, b is the region's parent block, D is the number of dependent block and h^i_{dep} the hit-count of i^{th} of these dependent control break blocks. If a region has no dependent blocks, the count will match the parent block count exactly. If done right, it can never happen that a region count is less than zero. An inner block, ending with a control flow break, cannot be entered more often than the surrounding block.

The ParserState helper class and the ControlBreak model class contain special logic to assign dependent blocks during parsing. We always have a curBlock pointer to the innermost block and an optional curCodeRegion member in ParserState. When we encounter a control flow break we save it to the controlBreak member of the current block. Afterwards, we register the block in its parent blocks. Every type of ControlBreak has its own stopping criterion for ending the propagation. These are listed in Table 4.2.

As soon as all parent blocks have a reference to the inner control break block, we continue parsing. When we continue in an outer block that does not immediately end (another

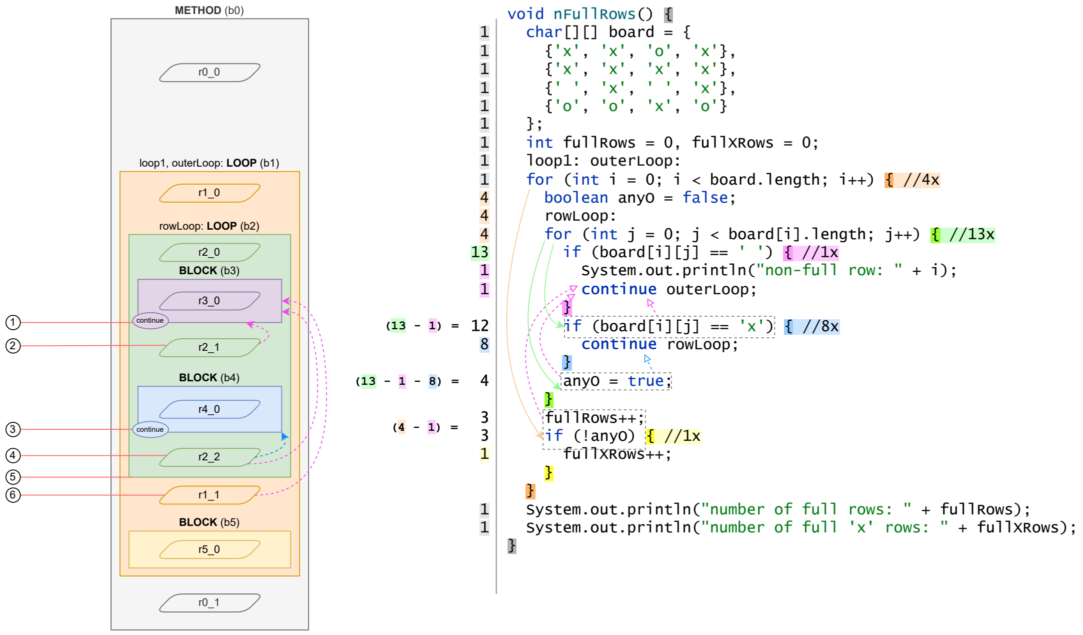
Control break type	Propagate until block type
`break`	innermost `LOOP` or switch case
`continue`	innermost `LOOP`
`return`	`METHOD` or `LAMBDA`
`yield`	innermost switch *expression*
`throw`	until `METHOD` or `TRY` block
`break/continue` with label	until a block matching *this* label

Table 4.2: Until which type of block an inner control break block gets registered.

statement follows), we start a new `CodeRegion`. At this point the region inherits all *currently* registered inner blocks as dependent control break blocks.

Another difficulty we were facing are labels. Blocks can be labeled with an identifier followed by a colon. A block can have multiple labels. The statements `break` and `continue` can have a label target, ignoring the innermost applicable loop or `switch`. Thus, we have to store a possible (list of) labels for each block. When encountering a control flow break with a label, we register this break in all parent blocks until reaching one with a matching label, independent of the block type.

A demonstration of how inner control break blocks are registered and propagated to regions, and how the final hit-counts will be calculated can be found in Figure 4.5.



```
                       void nFullRows() {
                   1     char[][] board = {
                   1       {'x', 'x', 'o', 'x'},
                   1       {'x', 'x', 'x', 'x'},
                   1       {' ', 'x', ' ', 'x'},
                   1       {'o', 'o', 'x', 'o'}
                   1     };
                   1     int fullRows = 0, fullXRows = 0;
                   1     loop1: outerLoop:
                   1     for (int i = 0; i < board.length; i++) { //4x
                   4       boolean anyO = false;
                   4       rowLoop:
                   4       for (int j = 0; j < board[i].length; j++) { //13x
                  13         if (board[i][j] == ' ') { //1x
                   1           System.out.println("non-full row: " + i);
                   1           continue outerLoop;
                             }
       (13 - 1) =     12         if (board[i][j] == 'x') { //8x
                   8           continue rowLoop;
                             }
   (13 - 1 - 8) =      4         anyO = true;
                           }
        (4 - 1) =      3       fullRows++;
                   3       if (!anyO) { //1x
                   1         fullXRows++;
                           }
                         }
                   1     System.out.println("number of full rows: " + fullRows);
                   1     System.out.println("number of full 'x' rows: " + fullXRows);
                       }
```

Figure 4.5: A program with control flow breaks and calculated region hit-counts

The blocks are color-coded. Region counts on the left of the program share the block's background color if no calculation is necessary. Otherwise the calculation is provided. On the left is a sketch of the internal object model for this method. The boxes are blocks and the parallelograms are regions. Blocks ending with a control flow break are marked with a circle. Dashed arrows show dependent blocks. Important moments during file parsing are marked with a red line and a number:

(1) At the moment of encountering "`continue outerLoop;`" block b3 is registered in parent blocks b2 and b1 as an inner control break block.

(2) When continuing with block b2 and starting a new code region r2_1, it inherits b3 as a dependent control break block.

(3) When encountering "`continue rowLoop;`" we register b4 only in parent b2 (in addition to b3; additive). This has no more effect on region r2_1.

(4) We continue with block b2 and start a third region. This region inherits b3 and b4 as dependent blocks.

(5) Block b2 goes out of scope as we return to block b1 (which still has b3 registered as an inner control break block).

(6) The second region r1_1 of block b1 inherits b3 as a dependent block.

5 Usage

This section contains information about how to download, build and use our Java profiler. It further describes how the command-line tool can be executed and which optional arguments are available. Finally, we introduce the graphical interface to run the tool and explain navigation and usage of the generated report.

5.1 Installation

For both executing the tool and building the software from source, a Java Development Kit (JDK) of Version 17 or higher is needed. A basic Java Runtime Environment (JRE) is not sufficient due to the fact that we leverage the `javac` utility to compile instrumented files into `.class` files. The Java compiler utility is absent in a JRE.

Download

Every release in the "Releases" section [1] of the profiler's GitHub page [32] contains:

- the pre-built `.jar` file of the command-line profiler
- a ZIP-file containing the JavaFX runner for each of the three major desktop operating systems (Windows, Linux, macOS)

Both the tool and the UI are "portable", meaning that they can be placed anyway in the local file system and can be executed from there. No prior setup step is required.

[1] `https://github.com/matwoess/java-profiler/releases`

Build from sources

Alternatively, the project can also be cloned or downloaded and compiled directly from its source files. Cloning the public repository can be done using the `git`[2] utility. After retrieving the project source files, we first need to generate the missing `Scanner.java` and `Parser.java` files using Coco/R. These two files are purposely excluded from the repository because they are defined by the ATG and `.frame` files and are updated implicitly with every change to the grammar. To automate this step, the project contains a bash and a PowerShell script inside the `scripts` folder. These scripts will download the Coco/R library (if it does not exist locally yet) and place it in the `lib` subfolder. It then uses the `Coco.jar` executable to generate the missing scanner and parser files. Once this step is completed, we can execute the Gradle[3] wrapper script `gradlew` in the project root to compile, package or run the project.

Using the commands from Listing 5.1, all those steps can be executed on a Linux or macOS system. For Windows, the script paths should be altered to used backslashes and a `.bat` extension for the `gradlew` command.

```
$ git clone https://github.com/matwoess/java-profiler.git
$ cd java-profiler
$ scripts/generate-parser.sh
$ gradlew <task>
```

Listing 5.1: Commands to clone and build the profiler on Linux or macOS. On Windows, the `generate-parser.sh` command should be replaced by `generate-parser.ps1`.

5.2 Command-line usage

When executing the `tool.Main` class of our profiler, we have a few optional and required arguments. Depending on the so-called "Run mode", different configurations are possible. The most important argument is the path to the file containing the main entry point of the project. When specified, it allows the profiler to execute all steps including running the program and generating the report autonomously.

[2] https://git-scm.com/

[3] https://gradle.org/

Using the command from Listing 5.2, we can use the `java` binary to execute the tool. For this command and the program itself to execute successfully, the `java` and `javac` binaries should be included in the system `PATH` variable. Installing a JDK on any platform should handle this automatically.

```
$ java -jar profiler.jar ARGS
```

Listing 5.2: Command line to run the profiler tool JAR

It is recommended to place the tool JAR in a central directory and to include a custom script to execute `java -jar path/to/profiler.jar "$@"` automatically, passing any additional arguments on to the executable. If this script is included in the system path, it can be executed from any directory using the ''`profile ARGS`'' short-form.

The complete help text when issuing `profile --help` is shown in Listing 5.3.

```
Usage: profiler [options] <main file> [program args]
Or   : profiler [options] <run mode>
Options:
  -h, --help                       display this message and quit
  -s, --synchronized               instrument using synchronized counters increments
  -v, --verbose                    output verbose info about instrumentation of files
  -d, --sources-directory <dir>    directory with additional java files to instrument
Run mode (exclusive):
  -i, --instrument-only <file|dir> only instrument a single file or directory and exit
  -r, --generate-report            only generate the report from metadata and counts
Main file:
 The path to the main Java file. It will be compiled and executed after instrumentation.
 (Must not be specified for the generate-report run mode)
Program args:
 Will be passed to the main method if given
```

Listing 5.3: The `--help` output of the command-line tool

The three run modes are: default, instrument-only and generate-report. In the default mode, all 5 workflow steps will be executed sequentially. In the instrument-only mode, all given Java files will only be analyzed and instrumented. This can be used for cases when the compilation requires special handling. The report-only mode requires the existence of the `metadata.dat` and `counts.dat` files inside an existing output directory. It will use these to (re-)generate the HTML report.

These modes can be used in the following three scenarios:

1. Instrument a Java file, compile it, execute it and generate the report.

```
$ profile Main.java arg1 arg2
```

2. Instrument *all* Java files in a directory, compile the main file, execute it and generate the report.

   ```
   $ profile -d src/ src/at/jku/ssw/Main.java
   ```

3. Only instrument the sources, *custom*-compile them (with additional arguments or a build tool) followed by executing the compiled classes manually and then using the `-r` run mode to only generate the report.

   ```
   $ profile -i src
   $ javac -cp .p/instr -customArg .p/instr/Main.java
   $ java -cp .p/instr Main
   $ profile -r
   ```

Passing the `--synchronized` option will cause all counters to be incremented atomically. This can be useful for ensuring exact counts for multi-threaded programs if a few methods or blocks are accessed very frequently by each thread simultaneously.

The `--verbose` option will result in detailed console output of which class/method/block-/region is entered and exited during the parsing step of Java files. This option should mainly be used for debugging purposes.

5.3 Graphical interface usage

For programming beginners, for users that are less familiar with the command line, and for general convenience, a JavaFX [21] application was created. It can be used to easily configure the parameters and options, execute the tool in a native terminal and quickly open the report in the browser.

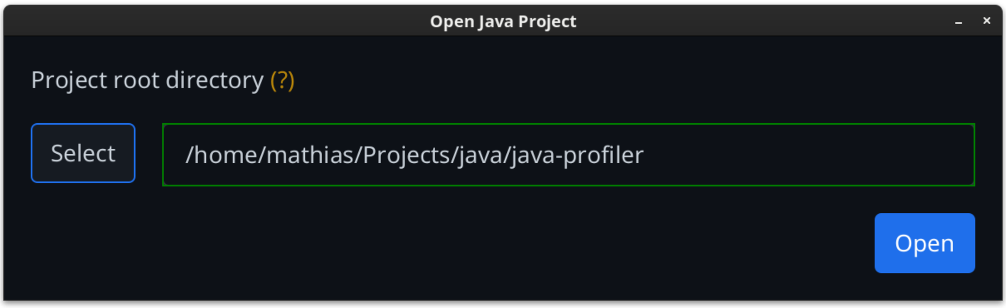
The `bin` folder of the JavaFX application ZIP file contains two executable script files. The `profiler-fxui` script for Linux and macOS and the `profiler-fxui.bat` script for Windows systems. These are automatically generated by the `distZip` Gradle task of the Distribution Plugin[4] and start the JavaFX UI. It configures all the necessary environment

[4] `https://docs.gradle.org/current/userguide/distribution_plugin.html`

variables such as the class-path and module-path and launches the `fxui/fxui.Launcher` module. Double clicking this script should open the project selection dialog.

Project selection dialog

Before displaying the main application window to configure the tool arguments, we first need to open a Java project. Figure 5.1 shows an example state of this dialog.



/home/mathias/Projects/java/java-profiler

Figure 5.1: The project selection dialog of the JavaFX application

Golden " (?) " labels throughout the application can be hovered over for detailed information about what the associated elements are for.

Clicking Select will show the system's native dialog to choose a folder. Alternatively the path can be entered manually. If the entered path is an existing folder, the border of the text field will be green. When an invalid path is specified, the border will turn red, indicating a misconfiguration. In case the text field is empty, it has no border. Only with a valid path selected or entered the Open button is enabled.

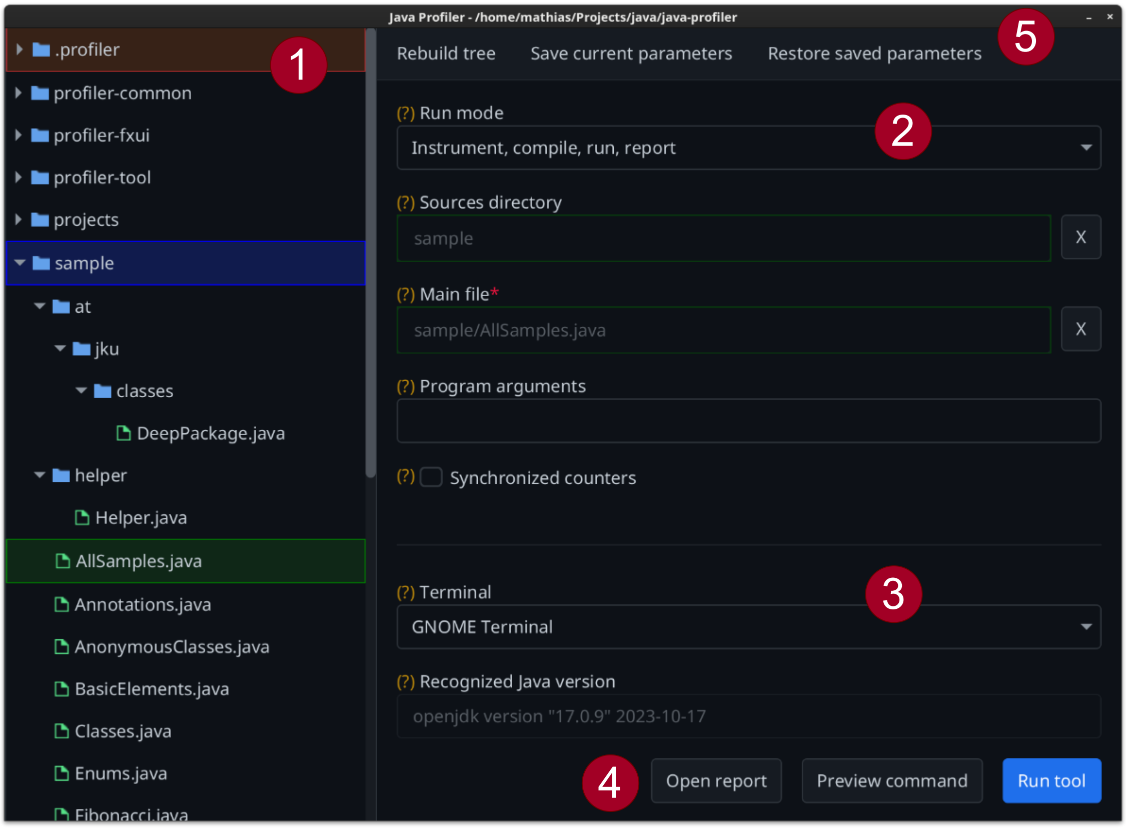
The native file picker window will start with the currently entered valid path in the text field for convenience. If the path string is empty or invalid, the default starting point for folder selection is the user's home directory (fetched by `System.getProperty("user.home")`).

Upon clicking the Open button, we are forwarded to the main application window. The current valid path is saved to a text file `lastProjectRootDirectory.txt` in the *application* directory (not the project directory). The next time we execute the application, the string of this path is pre-filled to quickly reopen the same project without relying on the folder picker dialog again.

If this dialog is closed, the application terminates immediately without displaying the main view.

The main application window

After selecting the project directory in the previous step, the main window is displayed. Figure 5.2 shows how this main view might look like, on the example of our project code itself. ① is the file tree view, ② are the selected parameters, ⑤ is a toolbar to save and restore parameters or rebuild the tree. Area ③ shows the selected native terminal application and the recognized Java version. The three buttons ④ can be used to open the report, to preview the run command and to execute the tool.



(?) Run mode
Instrument, compile, run, report

(?) Sources directory
sample

(?) Main file*
sample/AllSamples.java

(?) Program arguments

(?) □ Synchronized counters

(?) Terminal
GNOME Terminal

(?) Recognized Java version
openjdk version "17.0.9" 2023-10-17

Figure 5.2: The main windows of the JavaFX application

① The file system tree on the left part of the dialog lists all Java files and all directories that contain at least one Java file. The tree model is built recursively starting from the project root directory. The root itself is excluded, only showing its contents. Folders that do not contain any files ending with the `.java` extension are filtered out of the tree view. Directories have a blue icon, Java files a green one.

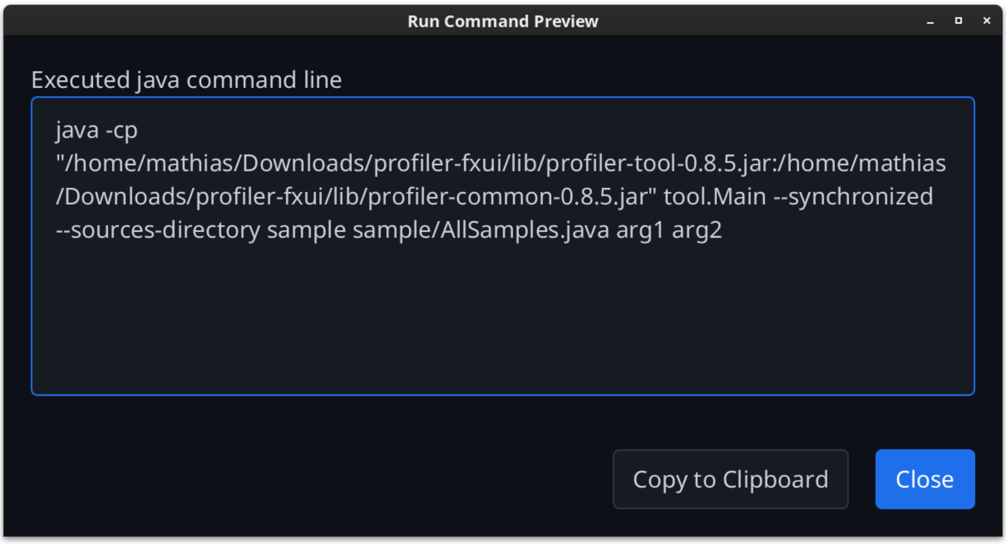
The main purpose of the tree view is to select the main Java source file and the sources directory. This can be done by selecting an item and pressing the Return key. Alternatively, a context menu can be opened on each item in the tree by right-clicking it. The only available menu entry will dynamically be either Select as sources root or Select as main file, depending on the type of the selected path.

If a folder was selected, it will be highlighted with a blue background color. The selected Java main file will be shown with a green background. The hidden `.profiler` output directory is highlighted in brown to separate it from the rest of the project folders.

② On the right side of the window, there is the main view. It contains the currently configured arguments and options for the tool when executing. The first property is the run mode. By default, it is configured to execute all steps. The other two run modes "Instrument only" and "Generate report only" can be selected by the drop-down. The next two fields are the sources directory and the main file. They are read-only, as they should be selected using the tree view. The X buttons next to them clear their value. The "Program arguments" field is a string of space-separated values that will be passed on to the executed program. Finally, the "Synchronized counters" checkbox will pass the `--synchronized` option to the profiler. The `--verbose` option is not available via the user interface. Depending on the "Run mode", unavailable options will be hidden dynamically.

③ The lower section of the main view contains two more fields. The "Terminal" drop-down can be used to specify the terminal application to be launched when pressing the Run tool button. The available options depend on the current operating system. The second field is "Recognized Java version". It is read-only and simply shows the first output line of the `java -version` command. It is there to inform the user about which version of the `java` and `javac` binaries will be used by the tool, in case there are multiple versions installed on the system.

④ At the bottom-right of the window there are three buttons. The right-most Run tool button is used to execute the command-line tool with all currently configured parameters. It is disabled until the current run mode's required parameters are set. The middle Preview command button will open a popup dialog as shown in Figure 5.3. There, the text field shows the exact command that will be executed in the configured terminal application. This preview can be used to get familiar the command-line usage of the profiler. The Copy to clipboard button can be used to quickly paste the full command into a terminal or append it to a script file. Finally, the left-most button Open report in Figure 5.2 in will open the `.profiler/report/index.html` file in the system's default application for `.html` files (usually a browser). The button will only be visible if this file exists.



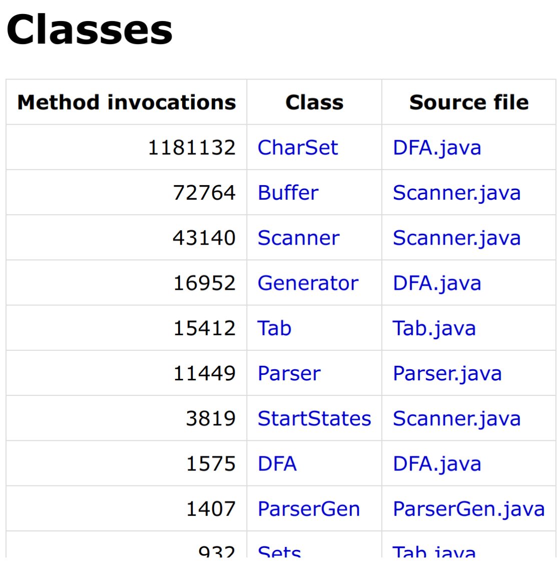
```
java -cp
"/home/mathias/Downloads/profiler-fxui/lib/profiler-tool-0.8.5.jar:/home/mathias
/Downloads/profiler-fxui/lib/profiler-common-0.8.5.jar" tool.Main --synchronized
--sources-directory sample sample/AllSamples.java arg1 arg2
```

Figure 5.3: The popup dialog previewing the profiler execution command. It is read-only and has two buttons for copying the command to the system clipboard and closing the dialog.

⑤ At the top of the main view is a toolbar with up to three buttons. Clicking the Rebuild tree button will recalculate the project's directory structure and refresh the tree view. This is currently the only way that newly created or deleted files can be added or removed from the tree. When pressing the Save current parameters button, all configured field values will be serialized into a `parameters.dat` file in the `.profiler` output directory. Only if this file exists, the Restore saved parameters item will be available and can be used to read, restore and overwrite all configured values.

5.4 Report

After the profiler generated the HTML report, all files are located in the `.profiler/report/` directory. The `index.html` is the main page of a report and presents an overview of all *top-level* Java classes in a project. Opening it in a browser will show a table as depicted in Figure 5.4.



Classes

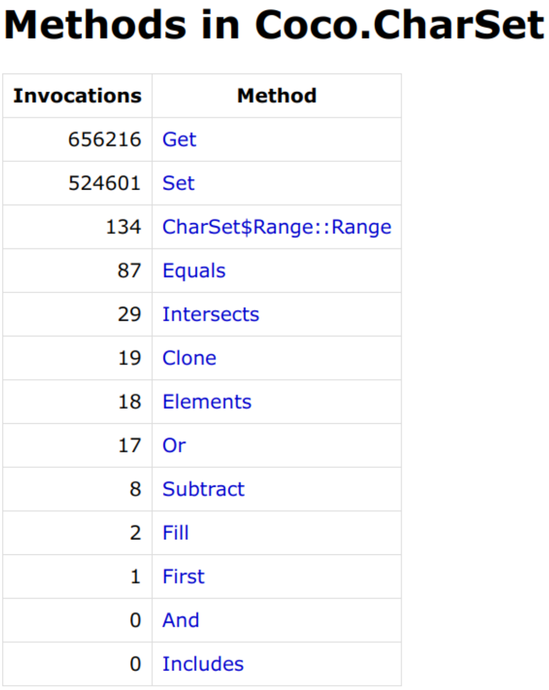
Method invocations	Class	Source file
1181132	CharSet	DFA.java
72764	Buffer	Scanner.java
43140	Scanner	Scanner.java
16952	Generator	DFA.java
15412	Tab	Tab.java
11449	Parser	Parser.java
3819	StartStates	Scanner.java
1575	DFA	DFA.java
1407	ParserGen	ParserGen.java
932	Sets	Tab.java

Figure 5.4: Partial classes overview table of a report generated by profiling Coco/R.

Every row in the table represents one top-level Java class of the profiled project. Each row has three columns.

- "Method invocations" - the aggregated count of hits, summed over all methods and constructors in this class. The hit count of sub-blocks inside each method block have no influence on this.
- "Class" - this column contains the class name, which is a hyperlink to the method index of this class. The class name is not fully qualified to avoid visual overhead.
- "Source file" - the file name that the class was located in. It is also displayed as a link to the annotated report source file. If multiple classes are defined in one source file, this column does not necessarily have unique values (like the `Buffer`, `Scanner` and `StartStates` classes all being located in the `Scanner.java` source file).

When navigating into one of the classes, we arrive at this class's method index. An example of such an index can be seen in Figure 5.5.



Methods in Coco.CharSet

Invocations	Method
656216	Get
524601	Set
134	CharSet$Range::Range
87	Equals
29	Intersects
19	Clone
18	Elements
17	Or
8	Subtract
2	Fill
1	First
0	And
0	Includes

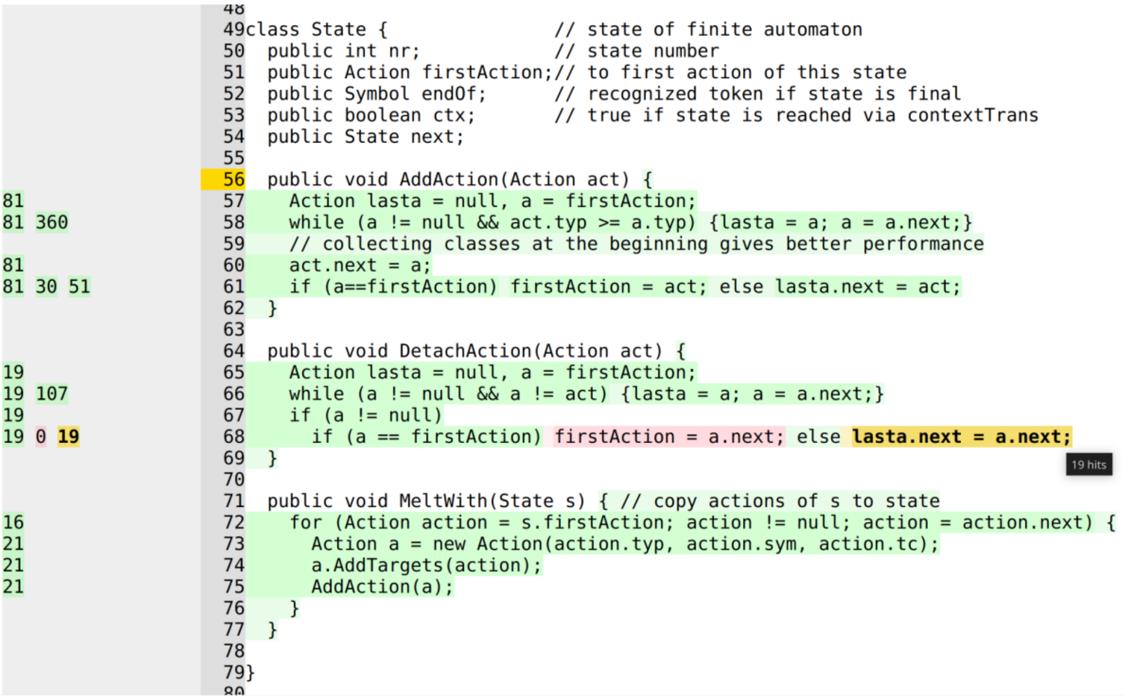
Figure 5.5: The method index of the `Coco.CharSet` class from a report, generated by profiling Coco/R.

It lists all methods and constructors of a top-level Java class and its inner classes, sorted by how often they were called in descending order. The table has two columns.

- "Invocations" - is the total amount of executions of each method.
- "Method" - the name of the method. If the class is an inner or anonymous class it will be shown with the class file naming prefix [33]. In the example form Figure 5.5 `CharSet` has an inner class `Range` with a constructor method `Range`, which is therefore listed as `CharSet$Range::Range`. Anonymous (inner) classes are numbered starting from 1. A method `foo` inside the first anonymous class in `CharSet` would be listed as `CharSet$1::foo`.

Clicking the link of a method name will forward us to the report detail-view of the source file. Additionally, the browser will jump to the line number of the method's definition.

For every Java source file of the project, containing at least one top-level class, an annotated HTML version will be created. It can be opened directly from the file system or by clicking its respective link in the classes or methods index.



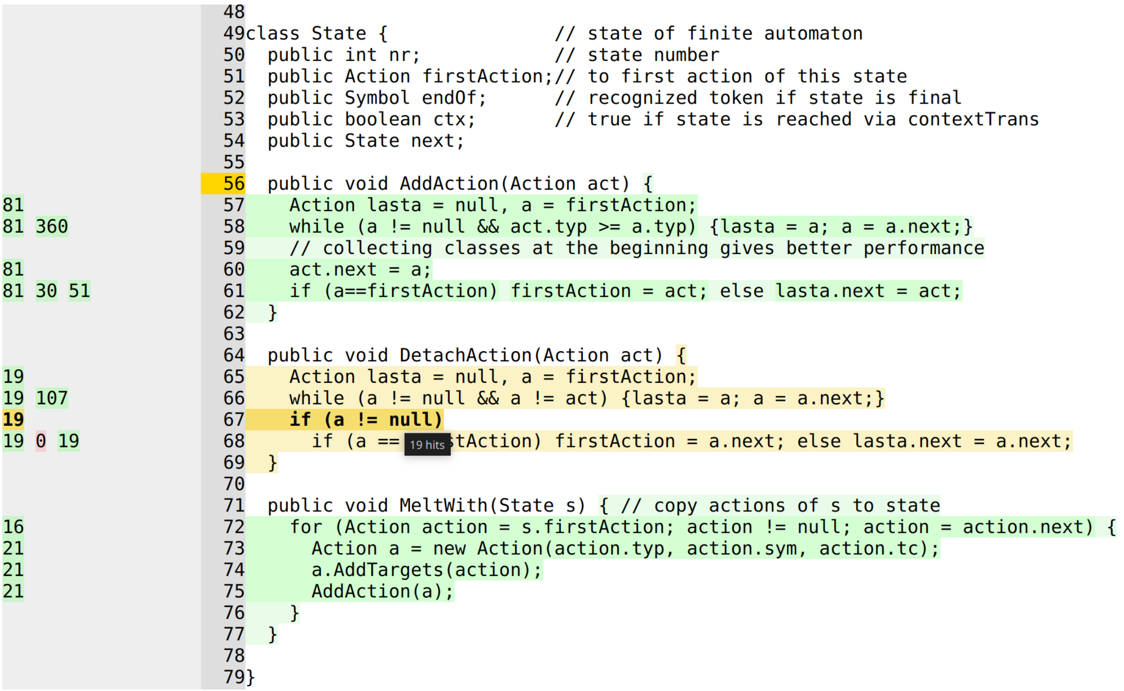
```
           48
           49class State {                 // state of finite automaton
           50  public int nr;              // state number
           51  public Action firstAction;// to first action of this state
           52  public Symbol endOf;        // recognized token if state is final
           53  public boolean ctx;         // true if state is reached via contextTrans
           54  public State next;
           55
           56  public void AddAction(Action act) {
81         57    Action lasta = null, a = firstAction;
81 360     58    while (a != null && act.typ >= a.typ) {lasta = a; a = a.next;}
           59    // collecting classes at the beginning gives better performance
81         60    act.next = a;
81 30 51   61    if (a==firstAction) firstAction = act; else lasta.next = act;
           62  }
           63
           64  public void DetachAction(Action act) {
19         65    Action lasta = null, a = firstAction;
19 107     66    while (a != null && a != act) {lasta = a; a = a.next;}
19         67    if (a != null)
19 0 19    68      if (a == firstAction) firstAction = a.next; else lasta.next = a.next;
           69  }
           70
           71  public void MeltWith(State s) { // copy actions of s to state
16         72    for (Action action = s.firstAction; action != null; action = action.next) {
21         73      Action a = new Action(action.typ, action.sym, action.tc);
21         74      a.AddTargets(action);
21         75      AddAction(a);
           76    }
           77  }
           78
           79}
           80
```

Figure 5.6: Partial screenshot of the annotated `DFA.java` source file from a Coco/R report

Figure 5.6 shows the line numbers 49 through 79 containing the definition of class `State`, its five member variables and its three methods. Left of the line numbers are hit counts for regions, to the right is the source code with regions and blocks color-highlighted. One code region is hovered upon in line number 68, showing a popup with its hit-count. The line number column separates the region hit-counts and the source code. Line 56 is marked, because we jumped directly to this line using a method index link. The file metadata and the counter values are used to color important parts in the code for immediate insight.

Blocks that were hit at least once during execution are highlighted with a light-green background color. Covered `CodeRegions` are then colored with a darker green. Not covered blocks and regions are shown with a light-red and darker red background.

Hovering over any block or region in the source code will show the surrounding block (and all its contents) in orange. If a region is selected, it will be displayed in a dark-orange background and bold font. Additionally, the hit-count on the left side of the line numbers is highlighted together with the corresponding region. This behavior can be better observed in Figure 5.7, where we hover over the if-statement in line 67. This leads to the whole method block being highlighted to show where the main counter of this statements comes from.



```
          48
          49class State {              // state of finite automaton
          50  public int nr;           // state number
          51  public Action firstAction;// to first action of this state
          52  public Symbol endOf;     // recognized token if state is final
          53  public boolean ctx;      // true if state is reached via contextTrans
          54  public State next;
          55
          56  public void AddAction(Action act) {
81        57    Action lasta = null, a = firstAction;
81 360    58    while (a != null && act.typ >= a.typ) {lasta = a; a = a.next;}
          59    // collecting classes at the beginning gives better performance
81        60    act.next = a;
81 30 51  61    if (a==firstAction) firstAction = act; else lasta.next = act;
          62  }
          63
          64  public void DetachAction(Action act) {
19        65    Action lasta = null, a = firstAction;
19 107    66    while (a != null && a != act) {lasta = a; a = a.next;}
19        67    if (a != null)
19 0 19   68      if (a == 19 hits tAction) firstAction = a.next; else lasta.next = a.next;
          69  }
          70
          71  public void MeltWith(State s) { // copy actions of s to state
16        72    for (Action action = s.firstAction; action != null; action = action.next) {
21        73      Action a = new Action(action.typ, action.sym, action.tc);
21        74      a.AddTargets(action);
21        75      AddAction(a);
          76    }
          77  }
          78
          79}
```

Figure 5.7: Partial screenshot of the annotated `DFA.java` source file from a Coco/R report with different highlighting.

Figure 5.7 shows the same content as in Figure 5.6, but contains different highlighting. The `if (a != null)` statement is hovered over, which is a region located directly inside the `DetachAction` method block. Both the method block and the code region are highlighted. A popup shows that it was executed 19 times. Hovering over a hit-count in the leftmost column will also highlight the associated source code region. The block will not be highlighted in this case and no unnecessary popup will be shown.

When leaving highlighted areas, all colors and effects are reset to the previous ones.

6 Evaluation

We attempted to create a simple, efficient and exact profiler, supporting the instrumentation of modern Java language features. We were striving for a robust and minimal fuzzy grammar that supports large and complex projects. In this section, we describe how we evaluated whether we achieved this goal.

6.1 Unit tests

We started out by supporting only basic `"class"` definitions (no interfaces, enums or records) and tried to find method blocks, ignoring all tokens between them. Everything else was skipped by Coco/R's `ANY` token. Inside methods only a few basic statement types such as `if/for/while/try` were recognized.

Starting from such a minimal program we systematically attempted to correctly parse bigger and bigger sample files. After correcting an error or adding support for a new language construct, unittests were added or updated. This way, we made sure that after extending the ATG, all previously tested forms were still recognized correctly.

Our test suite now contains more than 100 jUnit 5 test cases for the parsing and metadata collection step alone. Each such test consists of the following elements.

- A string definition of a sample Java class using mixed or specific language constructs
- The declaration of expected metadata, parsed from this string
- Parsing of a temporary file with the above content and comparison of equality

The test suite contains a builder class using a DSL-like syntax to define the expected metadata. One exemplary unit test case for *local classes* is shown in Listing 6.1.

```java
@Test
public void testInterfaceInIfBlockAndAnonymousInstantiation() {
  String fileContent = """
      public class Main {
        public static void main(String[] args) {
          if ((2 + 6) % 2 == 0) {
            interface IGreeter {
              default void greet() {
                System.out.println(\"Hello there.\");
              }
              void greetPerson(Object person);
            }
            IGreeter greeter = new IGreeter() {
              @Override
              public void greetPerson(Object person) {
                System.out.println(\"Hello \" + person.toString() + \".\");
              }
            };
            greeter.greetPerson(IGreeter.class);
          }
          if (true) return;
        }
      }
      """;
  JavaFile expected = jFile(
      jClass("Main",
          jClass(LOCAL, "IGreeter",
              jMethod("greet", 5, 7, 147, 204),
              jMethod("greetPerson")
          ),
          jClass(ANONYMOUS, null,
              jMethod("greetPerson", 12, 14, 361, 438)
          ),
          jMethod("main", 2, 19, 61, 522,
              jBlock(BLOCK, 3, 17, 89, 496),
              jSsBlock(BLOCK, 18, 18, 510, 518).withJump(RETURN)
          )
      )
  );
  TestUtils.assertResultEquals(expected, parseJavaFile(fileContent));
}
```

Listing 6.1: Demo unit test for a small Java class containing both a local and an anonymous class

In this unit test example we define a `jFile` containing a `jClass` called `"Main"` with two inner classes and a `"main"` method. The first inner class is a *local* interface `"IGreeter"` that is defined only inside an `if` block. The second is an anonymous, nameless class implementing this interface and overriding its abstract method. The interface defines a method `greet` with a default implementation from line number 5 to line number 7, starting at character position 147 and ending at character position 204. It also contains an abstract `greetPerson` method having no method block. The `Main.main` method's block spans from line 2 to line 19 and contains two inner `if` blocks. The second one is a single statement, declared by `"jSsBlock"` and ending with a `return` statement. We have to define this in the expected model by using the `".withControlBreak(RETURN)"` builder method.

In addition to the unit tests for parsing our repository includes a `sample` directory[1] with demo classes using all kinds of Java language features. Each sample file has its own test cases that serve as a kind of run configuration.

Finally, combinations of run mode, options and arguments for the profiler command-line tool are tested by the `MainTest` test class.

6.2 Larger test cases

In addition to our samples and code snippet test cases, we also verified our profiler against several larger Java projects. To ensure robustness and completeness of our fuzzy grammar, five projects were added gradually to our unit test suite.

Table 6.1 contains some statistics about how large or complex their respective code repositories are, based on numbers reported by the IntelliJ Statistic[2] plugin and summary output of our profiling tool. Coco/R was the first simpler project we tackled. Then zip4j was added as a larger test case. Finally, dacapobench, JaCoCo and jUnit 5 were added to fully test the stability of our profiler's parsing and instrumentation capabilities.

More projects will likely be added in the future, but successfully instrumenting the jUnit 5 codebase gave us confidence in the robustness of our tool.

[1] `https://github.com/matwoess/java-profiler/tree/main/sample`

[2] `https://plugins.jetbrains.com/plugin/4509-statistic`

	Reported by Statistic plugin			Found by profiler		
Project	Java files	Total lines	LoC	Classes	Methods	Blocks
Coco/R	7	4 722	3 891	31	246	1 358
zip4j	98	12 015	8 157	119	917	1 779
dacapobench	109	15 716	10 134	134	779	2 210
JaCoCo	646	67 411	39 727	882	3 993	5 730
jUnit 5	1 394	165 997	97 152	2 741	11 142	15 600

Table 6.1: This table shows statistics about the initial five larger test case projects. The left-side columns contain the number of `.java` files, their total lines and their lines of code (reported by the IntelliJ Statistic plugin). The right-side columns list the number of classes, method and blocks found by our profiler.

Coco/R [3] was the first milestone of our project. To successfully instrument, compile, run and generate a report for it was the first step towards a working grammar. The source code exhibits heavy usage of single-statement blocks, which was the first special case we had to consider. It often has multiple code blocks in one source code line, which was a challenge for report building and visualization. Additionally, the project was a fitting choice for a first larger test case due to the non-existence of any lambda expressions or other Java 8+ features.

zip4j [4] was the next larger repository we had chosen. It contains twice the amount of lines of code compared to Coco/R while still targeting Java 1.7. It is a library without a main entry point, so, we had to add a Main class. Its `main` method simply calls the zipping function for a folder into a temporary ZIP file. This project also helped us to solidify the pathing of our tool due to its deep maven directory structure.

dacapobench [5] is a popular benchmark suite. It uses the `ant` build tool to compile and is thus non-trivial to support with our instrumentation and compilation process. For validating the robustness of our parser and instrumenter, we included it as an instrument-only project. The minimum version for building the suite is Java 11.

[3] `https://github.com/SSW-CocoR/CocoR-Java`

[4] `https://github.com/srikanth-lingala/zip4j`

[5] `https://github.com/dacapobench/dacapobench`

JaCoCo [6] is a Java Code Coverage library and is another instrument-only test project. Its repository contains significantly more Java files, classes, methods and blocks than any previous test project. It features modern code, is actively maintained, is built with Java 17 syntax and supports code coverage for all modern language features. Successful instrumentation meant that our parser can handle features like switch expressions, records, annotations and lambda expressions.

jUnit 5 [7] is our biggest, most advanced instrumentation test case. At the time of writing our profiler found and successfully instrumented a total of 15,600 blocks in 2,741 Java classes inside this repository. This popular test suite is also actively maintained, built only by the latest Java 21 LTS release and should report an error first, if new features are unsupported.

6.3 Runtime impact

To get an idea of the run-time overhead caused by our counter logic in an instrumented program, we applied our profiler on several benchmarks of the DaCapo benchmark suite [34, 35] of Version 23.11-chopin (git revision `fd292e92`). We compared the execution time of seven hand-picked benchmarks with the following configurations:

- "orig" - The original unmodified benchmark project without instrumentation
- "instr" - A version with counter increment statements added to every code block
- "synclock" - The benchmark with synchronized counters (using the first attempt with `synchronized` methods and locking the `__Counter` class on each increment)
- "sync" - The same synchronized counters, but using the newer version of `__Counter` leveraging the `AtomicLongArray`

By default, when attempting to compile a benchmark, DaCapo deletes all pre-existing (instrumented) source files and build folders and newly extracts the source code from an automatically downloaded archive or version control.

[6] `https://github.com/jacoco/jacoco`

[7] `https://github.com/junit-team/junit5/`

		Warmup							Benchmarking				
Benchmark	Variant	1	2	3	4...27	28	29	30	31	32	33	34	35
avrora	orig	5 084	4 750	4 931	...	6 397	5 334	5 462	5 406	5 402	5 357	5 356	5 340
	instr	7 940	8 346	9 102	...	8 639	8 618	8 640	8 561	8 597	8 652	8 629	8 661
	sync	12 915	13 687	14 847	...	14 975	14 982	15 151	15 167	15 136	15 132	15 205	15 152
	synclock	94 068	92 935	92 659	...	96 226	95 538	95 226	96 034	95 523	96 024	97 763	95 624
fop	orig	3 411	1 414	1 304	...	747	741	700	670	720	769	759	666
	instr	3 841	1 754	1 678	...	914	993	937	934	993	896	988	886
	sync	5 450	3 029	2 878	...	1 800	1 745	1 730	1 751	1 709	1 767	1 843	1 748
	synclock	5 783	3 409	3 196	...	2 526	3 609	2 735	2 170	2 291	2 166	2 195	2 116
graphchi	orig	8 446	7 886	8 026	...	8 406	8 331	8 347	8 473	8 612	8 495	8 421	8 298
	instr	12 777	11 421	11 501	...	12 090	11 657	11 631	11 693	11 685	11 661	11 663	11 690
	sync	44 879	45 191	45 339	...	44 017	44 155	44 167	44 040	44 762	49 689	42 842	42 613
	synclock	171 544	168 820	169 933	...	164 234	165 190	165 397	164 489	166 175	166 632	165 512	165 423
h2	orig	16 895	11 631	10 277	...	10 205	10 316	10 325	10 227	10 244	10 413	10 429	10 382
	instr	13 736	11 686	10 167	...	10 521	10 711	10 485	10 358	10 521	10 318	10 413	10 429
	sync	13 574	11 212	11 274	...	10 275	10 394	10 302	10 222	10 409	10 385	11 564	11 005
	synclock	14 287	11 828	11 711	...	10 422	10 319	10 391	10 494	10 426	10 574	10 451	10 678
pmd	orig	8 882	5 662	5 374	...	3 485	3 491	3 494	3 928	3 518	3 496	3 502	3 501
	instr	11 078	9 547	8 836	...	7 388	7 439	7 479	7 418	7 414	7 527	7 474	7 394
	sync	10 965	9 173	9 186	...	7 593	7 609	7 583	7 608	7 577	7 666	8 170	7 912
	synclock	11 307	10 684	9 533	...	8 047	8 091	8 067	8 109	8 084	8 101	8 023	8 058
sunflow	orig	16 214	19 959	19 489	...	19 455	18 862	18 972	18 614	19 781	20 209	18 200	19 472
	instr	137 198	141 507	141 026	...	139 389	142 614	133 795	137 091	139 763	140 613	141 095	141 018
	sync	195 899	200 552	200 198	...	204 136	203 941	200 405	206 141	203 225	202 720	208 567	208 113
	synclock	1 986 732	2 264 628	2 268 873	...	2 289 126	2 247 669	2 229 021	2 232 473	2 248 488	2 249 519	2 250 110	2 246 401
xalan	orig	5 408	3 374	3 114	...	3 318	3 291	3 425	3 172	3 368	3 327	3 352	3 409
	instr	9 107	7 742	8 502	...	8 345	8 348	9 011	8 443	8 457	8 421	8 349	8 403
	sync	13 514	12 483	12 584	...	13 032	12 987	12 954	13 191	13 013	13 033	12 964	13 033
	synclock	159 638	167 489	165 771	...	171 297	172 138	170 682	170 935	171 086	169 206	167 826	166 837

Table 6.2: Run time in milliseconds for 35 runs of 7 programs from the DaCapo benchmark suite in different configurations. Values for the *warmup* runs 4 through 27 are excluded.

To instrument the programs, we had to modify the build configurations of DaCapo and the projects themselves (to include our `__Counter` class).

Each configuration was executed 35 times in a test environment. We used a Lenovo ThinkPad X1 Yoga 3rd Gen laptop (as opposed to a dedicated server) running a 64-bit Linux 6.7.4-arch1-1 operating system, on AC, using the "Performance" power mode and OpenJDK (build 17.0.10+7). The laptop features an Intel® Core™ i7-8550U CPU with 4 cores and 8 hyper-threads. We used the default workloads for each benchmark. Any multi-threaded benchmark was executed using 4 threads to drive the workload. The reported run time of benchmark execution times (in milliseconds) for each configuration can be found in Table 6.2. Additional information about the chosen set of benchmarks are listed in Table 6.3.

We immediately observe that the "synclock" variant entails a significant run-time overhead, especially for highly concurrent programs. This has led us to the implementation of the alternative "sync" variant. In the case of `sunflow` (a CPU ray-tracing program) the run time has increased from an average of 19.2 seconds to 37 minutes. This is a slowdown of 117 times. For the following visualization we excluded the "synclock" variant entirely.

According to Kalibera et al. [36] (Table 3), all the benchmarks we chose should reach a stable state after about two to six warmup runs. For visualization of the run-time overhead, we ignore the first thirty runs and averaged the run time of the final five.

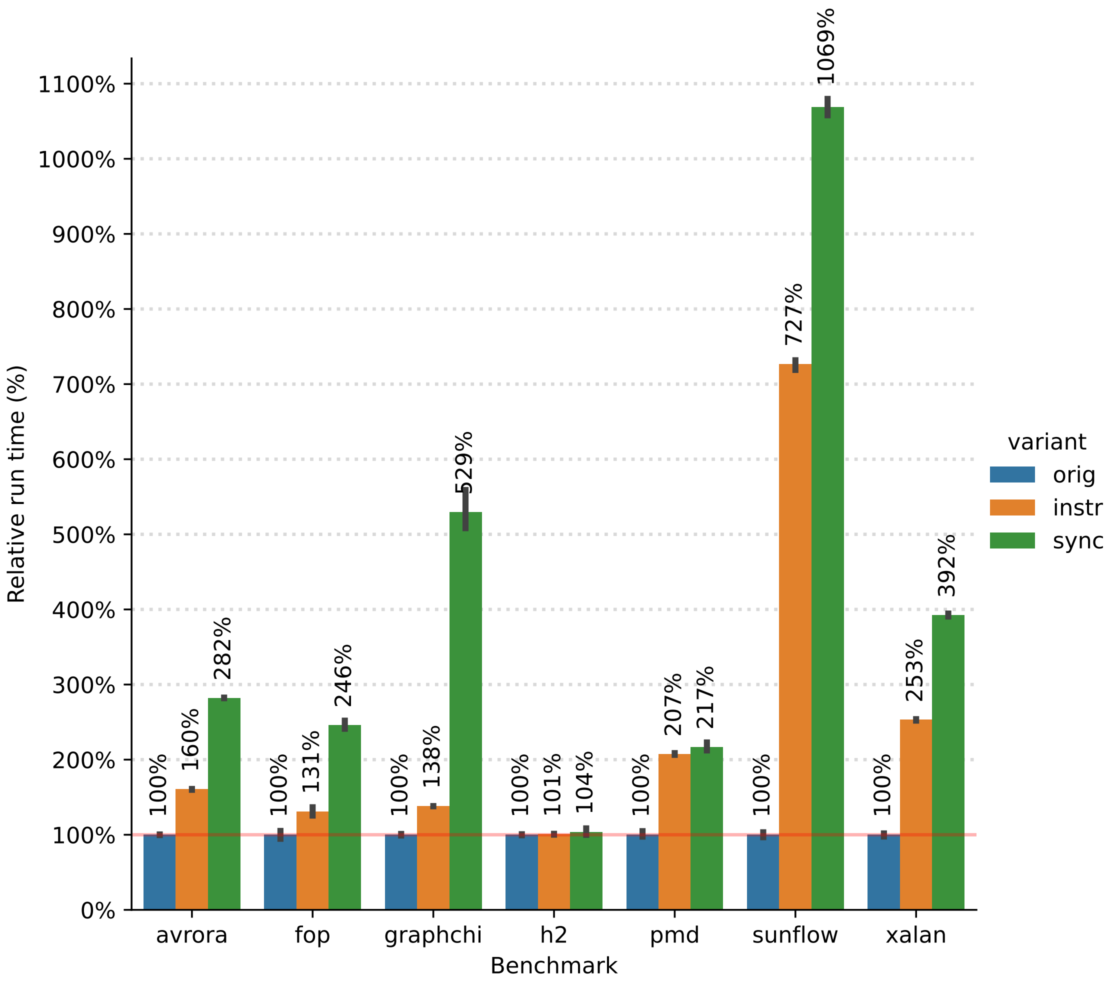


Figure 6.1: Relative run-time overhead of our profiler applied to DaCapo benchmarks, using synchronized and unsychronized counter variants

In Figure 6.1 we visualize the run-time overhead of instrumented programs and those using synchronized counters (with the `AtomicLongArray`). Most benchmarks show only a relatively small slowdown to less than 200% run time. When using the synchronized variant, we observe a more significant overhead for getting exact counts of every block.

Benchmark	Multi-threaded	Description
avrora	internally	simulates programs on a microcontroller grid
fop	no	generates PDFs out of XSL-FO files
graphchi	yes	a disk-based graph computation engine
h2	yes	benchmarks an in-memory banking database
pmd	internally	analyzes Java classes for source code problems
sunflow	yes	renders a set of images using ray tracing
xalan	yes	transforms XML documents into HTML

Table 6.3: Description and multi-threading level of each benchmark in the chosen set. "Internally" means, that the benchmark is driven by a single external thread, but is multi-threaded internally [37].

We can explain the non-existent overhead for `h2` by having instrumented only the program itself and not its `derby` database driver library. The `sunflow` benchmark is the opposite extreme. While basic instrumentation induces an overhead of more than 700% run time, the synchronized counters result in 10-fold execution time. This is still way better than the "synclock" variant, but must be considered when making use of our profiler for heavily parallel programs.

We conclude, that our profiler can effectively be used on almost all real-world applications to gather exact coverage data and find hot-spots in a program with a reasonably small overhead. Especially for multi-threaded programs, however, we do not recommend to use the profiling configuration for production environments.

6.4 Limitations

Our approach and architecture led to a few temporary and a few general limitations. They should be considered when making use of this open-source profiler.

General limitations

- Only the project itself is instrumented (no library classes without source code).

While online bytecode profilers often instrument classes of the JDK or external libraries on its first usage, our profiler focuses only on the source code and classes of the project itself.

- Run-time exceptions inside and outside of `try` blocks cannot be considered for the resulting coverage data.

  If a statement fails without explicitly throwing an Exception inside the project's code itself, the hit-count of left-out statements is not subtracted and the whole current block or region is shown as equally covered.

Temporary limitations

- The current "method invocations" metric does not reflect how many statements were executed inside the method.

  For example, when a method was only executed once but contains loops with high repetitions, it will be sorted at the bottom of the report table and will contribute little to the class' aggregated count. Further metrics are needed.

- Custom build tools (like `ant`, `maven` and `gradle`) are not supported yet.

  The profiler only creates instrumented copies of `.java` files and can automatically compile the instrumented main class. To compile complex projects, copying and usage of additional build tool files must be done manually.

- Imperfect grammar.

  While we can successfully parse and instrument large projects, we do not claim to find every possible code block. The fuzzy approach leads to some special structures being currently skipped.

- No full Java 21 feature support yet. Switch case pattern matching can currently cause parser errors.

7 Conclusion and Outlook

The profiler we created leverages a small and simple grammar and a compiler generator to efficiently parse, instrument and profile Java projects of any size. It is easy to use, runs on all desktop platforms and creates basic but intuitive summary reports.

The growing feature set of the Java language created many challenges for correct instrumentation. Our choice of designing the ATG in a fuzzy and minimal way resulted in reduced complexity and better maintainability, while still being robust against large projects. Many special cases had to be considered to achieve correct and full instrumentation of every block type.

Our tool is free, requires no installation step and can be used without a steep learning curve. Due to its open-source nature and permissive license we hope for quick adoption and collaboration efforts. Feedback on our GitHub page is encouraged and appreciated.

Future work and ideas

While our Java profiler now has all the features of our initial design, we plan to develop it further and extend its capabilities. Some of the ideas we had are listed below.

- Creating even more test cases including parsing and instrumentation tests for large projects. The Qualitas corpus [38] provides a curated collection of Java software systems[1], of which several more candidates could be included in our unit test suite to ensure wide compatibility of our parser.
- Improving the report. We only have a simple metric of method invocations determining the sort order for report index tables. Additional statistics could be added to diversify the exploration possibilities. The sort order should be customizable.

[1] http://qualitascorpus.com/docs/catalogue/20130901/index.html

- Track time in counters. The only thing we profile is the hit-count of blocks. This gives us no information of how long statements and methods actually were executing. By inserting an additional statement at the *end* of blocks, we could calculate the time difference between entering and exiting blocks. Averaging the results should give insight into their mean execution time. Alternatively, we could only log the current system time when incrementing a counter and do more complex calculations during report generation.

- Java (keyword) syntax highlighting in report source files. Currently the displayed code is just the monospace text. Adding highlighting might lead to better readability.

- Merge counter results of multiple runs into one report. Currently, every time we run the instrumented and compiled program the counter data are overwriting the previous results. If we collect multiple `counts<suffix>.dat` files in a subfolder, we could sum up all counter values and generate a report from the merged results. This could be especially beneficial for running a unit test suite on the profiled project. Of course, this only work if the program wasn't modified in between runs (should be validated by a checksum).

- Support for Java 21 features. Specifically switch case pattern matching should be reflected in the grammar.

- Diverse improvements for the JavaFX tool-runner UI. For instance, a better auto-refreshing file tree and more theming options such as a light or dark theme depending on current system settings.

- Integrations of our profiler for IDEs and build tools to automate the profiling process and improve user experience. For example, a special "Profile" button in an IDE or a profiling-task from a plugin for build tools (like Gradle) could be implemented.

- Creating a public repository on Maven Central[2] to allow for declarative integration and automated fetching of our profiler as a dependency.

[2] `https://mvnrepository.com/repos/central`

Bibliography

[1] Qurrat Ul Ain et al. "Analysis of hotspot methods in JVM for best-effort run-time parallelization". In: *Proceedings of the 9th International Conference on E-Education, E-Business, E-Management and E-Learning*. 2018, pp. 60–65 (cit. on p. 1).

[2] *The JIT compiler - IBM Documentation*. Mar. 29, 2023. URL: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler (visited on 12/27/2023) (cit. on p. 1).

[3] Dávid Tengeri et al. "Negative effects of bytecode instrumentation on Java source code coverage". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 225–235 (cit. on pp. 1, 2, 3).

[4] Todd Mytkowicz et al. "Evaluating the accuracy of Java profilers". In: *ACM Sigplan Notices* 45.6 (2010), pp. 187–197 (cit. on p. 2).

[5] John Whaley. "A portable sampling-based profiler for Java virtual machines". In: *Proceedings of the ACM 2000 conference on Java Grande*. 2000, pp. 78–87 (cit. on p. 2).

[6] Hanspeter Mössenböck, Albrecht Wöss, and Markus Löberbauer. *Der Compilergenerator Coco/R*. na, 2003 (cit. on pp. 2, 10, 22).

[7] *Chapter 4. The class File Format*. Dec. 27, 2023. URL: https://docs.oracle.com/en/java/javase/17/docs/specs/man/java.html (visited on 12/27/2023) (cit. on p. 3).

[8] Walter Binder, Jarle Hulaas, and Philippe Moret. "Advanced Java bytecode instrumentation". In: *Proceedings of the 5th international symposium on Principles and practice of programming in Java*. 2007, pp. 135–144 (cit. on pp. 3, 4).

[9] *Prof-It for C#*. Sept. 9, 2010. URL: http://dotnet.jku.at/projects/Prof-It/Default.aspx (visited on 01/17/2024) (cit. on p. 5).

[10] *JEP 328: Flight Recorder*. Sept. 9, 2018. URL: https://openjdk.org/jeps/328 (visited on 01/18/2024) (cit. on p. 5).

[11] *About Java Flight Recorder*. Jan. 18, 2024. URL: `https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm` (visited on 01/18/2024) (cit. on p. 5).

[12] *JDK Mission Control*. Jan. 18, 2024. URL: `https://www.oracle.com/java/technologies/jdk-mission-control.html` (visited on 01/18/2024) (cit. on p. 6).

[13] *VisualVM: Home*. Feb. 8, 2024. URL: `https://visualvm.github.io/` (visited on 02/08/2024) (cit. on p. 7).

[14] *Cobertura*. Jan. 19, 2024. URL: `https://cobertura.github.io/cobertura/` (visited on 01/19/2024) (cit. on p. 7).

[15] *EclEmma - JaCoCo Java Code Coverage Library*. Oct. 16, 2023. URL: `https://www.eclemma.org/jacoco/` (visited on 01/19/2024) (cit. on p. 7).

[16] Thomas Mccabe. "Cyclomatic complexity and the year 2000". In: *IEEE Software* 13.3 (1996), pp. 115–117 (cit. on p. 7).

[17] *Profiling Tools and IntelliJ IDEA Ultimate | The IntelliJ IDEA Blog*. Mar. 6, 2020. URL: `https://blog.jetbrains.com/idea/2020/03/profiling-tools-and-intellij-idea-ultimate/` (visited on 01/19/2024) (cit. on p. 8).

[18] *The Java® Language Specification - Java SE 17 Edition*. Aug. 9, 2021. URL: `https://docs.oracle.com/javase/specs/jls/se17/html/index.html` (visited on 12/27/2023) (cit. on pp. 9, 22).

[19] *Java Class File Naming Conventions | Baeldung*. Jan. 5, 2024. URL: `https://www.baeldung.com/java-class-file-naming` (visited on 01/08/2024) (cit. on p. 10).

[20] Niklaus Wirth. "What can we do about the unnecessary diversity of notation for syntactic definitions?" In: *Communications of the ACM* 20.11 (1977), pp. 822–823 (cit. on p. 10).

[21] *JavaFX*. Jan. 6, 2024. URL: `https://openjfx.io/index.html` (visited on 01/06/2024) (cit. on pp. 12, 47).

[22] *The javac Command - Directory Hierarchies*. Dec. 23, 2023. URL: `https://docs.oracle.com/en/java/javase/17/docs/specs/man/javac.html#directory-hierarchies` (visited on 12/23/2023) (cit. on p. 17).

[23] *The java Command*. Dec. 23, 2023. URL: `https://docs.oracle.com/en/java/javase/17/docs/specs/man/java.html` (visited on 12/23/2023) (cit. on p. 17).

[24] *Runtime (Java SE 17 & JDK 17) - addShutdownHook(java.lang.Thread)*. Dec. 27, 2023. URL: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Runtime.html#addShutdownHook(java.lang.Thread)` (visited on 12/27/2023) (cit. on p. 20).

[25] *HTML Global title Attribute*. Jan. 25, 2024. URL: `https://www.w3schools.com/tags/att_global_title.asp` (visited on 01/25/2024) (cit. on p. 29).

[26] *Constructor Specification in Java | Baeldung - Rules of Constructor Invocation*. Jan. 8, 2024. URL: `https://www.baeldung.com/java-constructor-specification#rules-of-constructor-invocation` (visited on 01/26/2024) (cit. on p. 33).

[27] *Local Classes (The Java™ Tutorials > Learning the Java Language > Classes and Objects)*. Jan. 26, 2024. URL: `https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html` (visited on 01/26/2024) (cit. on p. 34).

[28] *Anonymous Classes (The Java™ Tutorials > Learning the Java Language > Classes and Objects)*. Jan. 26, 2024. URL: `https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html` (visited on 01/26/2024) (cit. on p. 34).

[29] *JEP 361: Switch Expressions*. Mar. 11, 2022. URL: `https://openjdk.org/jeps/361` (visited on 01/26/2024) (cit. on p. 34).

[30] *Lambda Expressions (The Java™ Tutorials > Learning the Java Language > Classes and Objects)*. Jan. 27, 2024. URL: `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html` (visited on 01/27/2024) (cit. on p. 36).

[31] *JEP 378: Text Blocks*. July 30, 2020. URL: `https://openjdk.org/jeps/378` (visited on 01/28/2024) (cit. on p. 38).

[32] *matwoess/java-profiler: A command-line profiler for Java programs that generates HTML reports. Features an optional JavaFX tool-runner GUI.* Jan. 5, 2024. URL: `https://github.com/matwoess/java-profiler` (visited on 01/05/2024) (cit. on p. 44).

[33] *Java Documentation - Get Started*. Jan. 19, 2024. URL: `https://docs.oracle.com/en/java/` (visited on 01/19/2024) (cit. on p. 53).

[34] S. M. Blackburn et al. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190. DOI: `http://doi.acm.org/10.1145/1167473.1167488` (cit. on p. 60).

[35] S. M. Blackburn et al. *The DaCapo Benchmarks: Java Benchmarking Development and Analysis (Extended Version)*. Tech. rep. TR-CS-06-01. http://www.dacapobench.org. 2006 (cit. on p. 60).

[36] Tomas Kalibera and Richard Jones. "Rigorous benchmarking in reasonable time". In: *Proceedings of the 2013 international symposium on memory management*. 2013, pp. 63–74 (cit. on p. 62).

[37] *DaCapo Benchmarks Home Page*. Jan. 11, 2024. URL: `https://dacapobench.sourceforge.net/` (visited on 01/11/2024) (cit. on p. 63).

[38] Ewan Tempero et al. "Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies". In: *2010 Asia Pacific Software Engineering Conference (APSEC2010)*. Dec. 2010, pp. 336–345. DOI: `http://dx.doi.org/10.1109/APSEC.2010.46` (cit. on p. 65).

Appendix

```
string =
    quote (
        quote quote lf  // """\n -> multi-line "text block" string
            {   noQuote
                | quote noQuote
                | quote quote noQuote
                | [quote] [quote] bslash quote
            }
            quote quote quote
        | {            // common string literal
            (   noQuoteNoBSlash
                | bslash (
                    escapableChar
                    | ’u’ {’u’} hexDigit hexDigit hexDigit hexDigit
                    | (octalDigit [octalDigit] | zeroToThree octalDigit octalDigit)
                )
            )
        }
        quote
    ).
```

Listing 1: Coco/R EBNF scanner specification to correctly parse all types of strings, including multi-line text blocks.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, Februar 2024
Ort, Datum

Mathias Wöß
Mathias Wöß, BSc