Java Project Analyser

Course Project for Java EE, Tongji University

Team Leader: **Zhang Yao**
zhang@cinea.cc

Tongji University

Table of Contents

Overview

Team Members

Name	Matr. No.	Contact number	Email address
Zhang Yao	2152955	+86 13518772062	zhang@cinea.cc
Tang Shuhan	2153877	+86 15300862015	2405353202@qq.com
Wang Shuyu	2151894	+86 17709584390	1491456253@qq.com
Tang Kexian	2152240	+86 18707016886	1135530278@qq.com

No specific order in the table.

Description

This project offers analysis functionality for Java projects. Users can conveniently obtain the call relationship of specified methods in a project using the software provided by this project. This includes a list of methods that a given method calls, as well as a list of methods that call the given method. The software can also trace call chains to a specified depth. Additionally, it provides tracking for the sources of function arguments within a method, assisting users in locating the origins of these arguments. If the arguments are provided in the form of literals, the software will also directly provide their values.

Features

Here is a feature list which is implemented by our team:

- REPL-style command-line interaction
- Import and organize Java projects according to package structure
- Import large-scale Java projects within an acceptable timeframe
- Check for the presence of syntax errors when importing projects
- Parallelize reading, importing, and analyzing projects
- Provide user-friendly output for time-consuming operations
- List all files, classes and interfaces, and methods in the project
- List all instances of a specific class within the project
- Distinguish different function overloads and prompt users to make a selection in cases where overloads exist.
- Find all other methods called by a specified method
- Identify all instances where a specific method is called within the project
- Trace all methods that could potentially call a specified method
- Locate and trace all possible argument sources for a specified method and provide values for literal argument sources.

Usage

Requirement

A Java environment that is **greater than or equal to 17 is required**. We use **Amazon Corretto 17 LTS** in our development and test. But other JDK distribution is also compatiable. Learn more about Amazon Corretto: https://aws.amazon.com/corretto

Compile our project

We use **Maven** to manage our project and its dependencies. A **Maven Wrapper** instance is also included in our project repository. You can just using it to compile our project in most cases. Learn more about Maven Wrapper: https://maven.apache.org/wrapper/

We suggest checking your JDK version before compiling our project:

```
${JAVA_HOME}/bin/java -version    # Never continue if not 17 or higher.
```

To start the compilation process, please execute the following command:

```
./mvnw -B dependency:go-offline
./mvnw -B package
```

The first command will try to download all dependencies our project used from the Maven Central Repository. You may use Aliyun Maven Central Repository in China to accelerate the downloading process. Learn more about Aliyun Maven Central Repository: https://developer.aliyun.com/mvn/guide.

After finishing the compilation, you may find the artifact in `target` directory, which looks like `java-project-analyser-cli-0.0.1-SNAPSHOT.jar`. You may move the output to a convenient location and change its name as needed. Assuming we have renamed it to `analyser.jar` in convience in the following parts.

Simple usage

Just use `java -jar` command to launch our project and use it:

```
java -jar ./analyser.jar
```

Our project provides a REPL user interface, in which you can easily input commands, obtain results, and reuse the previous state instead of having to restart the program after each command execution. Our project utilizes the **Spring Shell** framework to provide you with an exceptional interactive command-line experience. Learn more about Spring Shell: https://spring.io/projects/spring-shell.

Enter `help` to get a list of functionalities provided by our project.

```
shell:>help
```

Open a project

> **Important**
>
> Our project does not support libraries that generate and process code at compile-time, such as **Lombok**.
>
> If you want to analyse a project which uses Lombok, consider **Delombok** it manually before analysing. Learn more about the Delombok feature: https://projectlombok.org/features/delombok

Use open command to open, load and analyse a Java project:

```
shell:>open /usr/src/sample-project/src/main/java
```

Tips:

- The path should contain the root package, but not the root package self.
- On Windows, please double the backslashes in the path or change them to forward slashes. For example, both `C:/commons-lang/src/main/java` and `C:\\commons-lang\\src\\main\\java` are correct.
- The process of opening the project should be completed within one second, and it should not exceed ten seconds at most. In our test, open a huge project with 200 classes and 3000 methods usually cost about 2 seconds.

Example output:

```
shell:>open D:\\IdeaProjects\\java-project-analyser-sample\\src\\main\\java
Resolving 100% [============================] 1/1 (0:00:00 / 0:00:00)
Indexing 100% [============================] 3/3 (0:00:00 / 0:00:00) \

Open project success:
                      1 packages found.
                      1 files found.
                      1 classes and interfaces found.
                      3 methods found.
                      213ms used.
```

Query method invocation relationships

Use `func` command to query the method invocation relationships of a specified method:

```
shell:>func <class reference> <method name> <optional: searching deepth>
```

For example:

```
shell:>func main.Test sayHello
```

Example output:

```
shell:> func main.Test sayHello
Please wait.
******************* Invokes ******************
  -> java.io.PrintStream.println(java.lang.String)
**********************************************

****************** Invoked by ******************
  <- main.Test.introduction(java.lang.String)
      <- main.Test.main(java.lang.String[])
  <- main.Test.main(java.lang.String[])
**********************************************


Time cost:
    Get Invokes: 3 ms
    Get Invoked: 1 ms
shell:>
```

Some times you may want to specify a method from its overloads. The program will distinguish this situation and ask you to choose the method you need:

```
shell:>func org.apache.commons.lang3.StringUtils split
***************
 1) org.apache.commons.lang3.StringUtils.split(String)
 2) org.apache.commons.lang3.StringUtils.split(String, String)
 3) org.apache.commons.lang3.StringUtils.split(String, String, int)
 4) org.apache.commons.lang3.StringUtils.split(String, char)
***************
Please choose one:
```

Query method parameter sources

Use `param` command to query the parameter sources of a specified method:

```
shell:>param <class reference> <method name> <optional: searching deepth>
```

For example:

```
shell:>func main.Test sayHello
```

Example output:

```
shell:>param main.Test sayHello
Please wait.
******************Parameter Origin******************
  String name:
        name1: (main.Test) introduction
         <- "Odie": (main.Test) main
```

```
        "Garfield": (main.Test) introduction
        "Jon": (main.Test) sayHello
*******************************************************

Time cost: 4 ms
shell:>
```

Automaticly fixing syntax errors

to be implemented

Design

System architecture and components

Our system is built on top of the Spring Boot framework. We utilize the **Spring Framework** to organize and manage the lifecycle of Beans and dependency injection. We employ Spring Shell to create an easy-to-use interactive command-line interface.

Specifically, our system is mainly divided into the following parts:

- **Service interfaces and their implementations**, which are responsible for providing the core functions of the project, such as importing the project, parsing the method call relationships, and tracing back the sources of parameters.
- **Model classes**, which are the data structures used by the core business of the project. They abstract and model business information and store this information.
- **Helper classes**, which are tools used during the execution of the project's core business. They operate without any state, provide only static services, and are not instantiated into class instances.
- DTOs (**Data Transfer Objects**), which are the data structures used for interaction between the core business classes and helper classes. These data structures don't store business-related models but are responsible solely for pure information transfer.

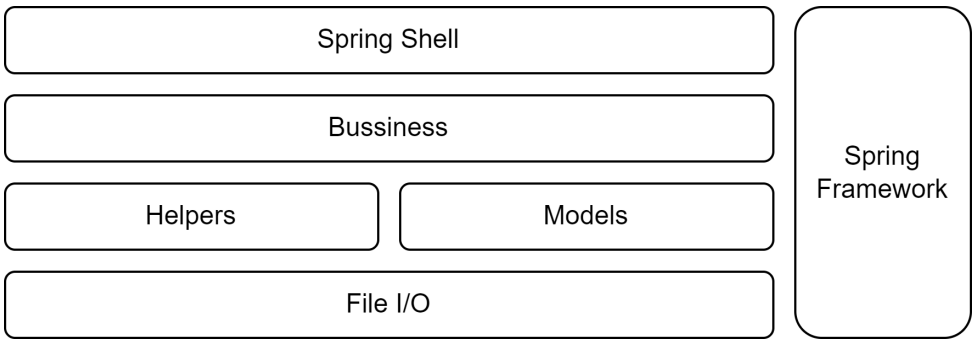

Figure 1: System architecture

General workflow

We have the control and data flow as below:

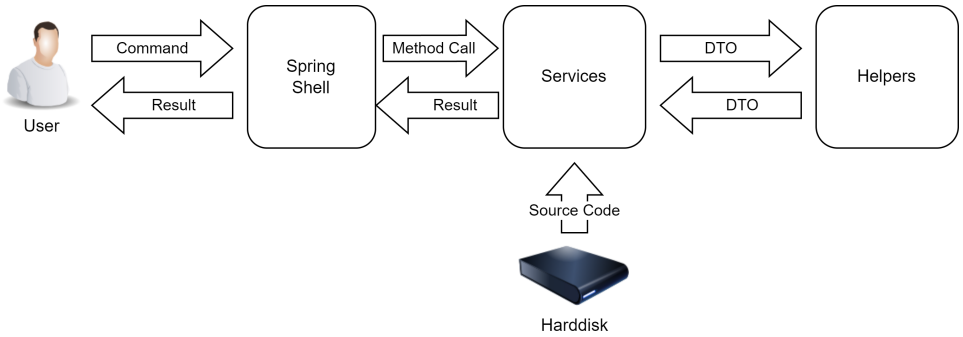

Figure 2: System control and data flow

Project loading workflow

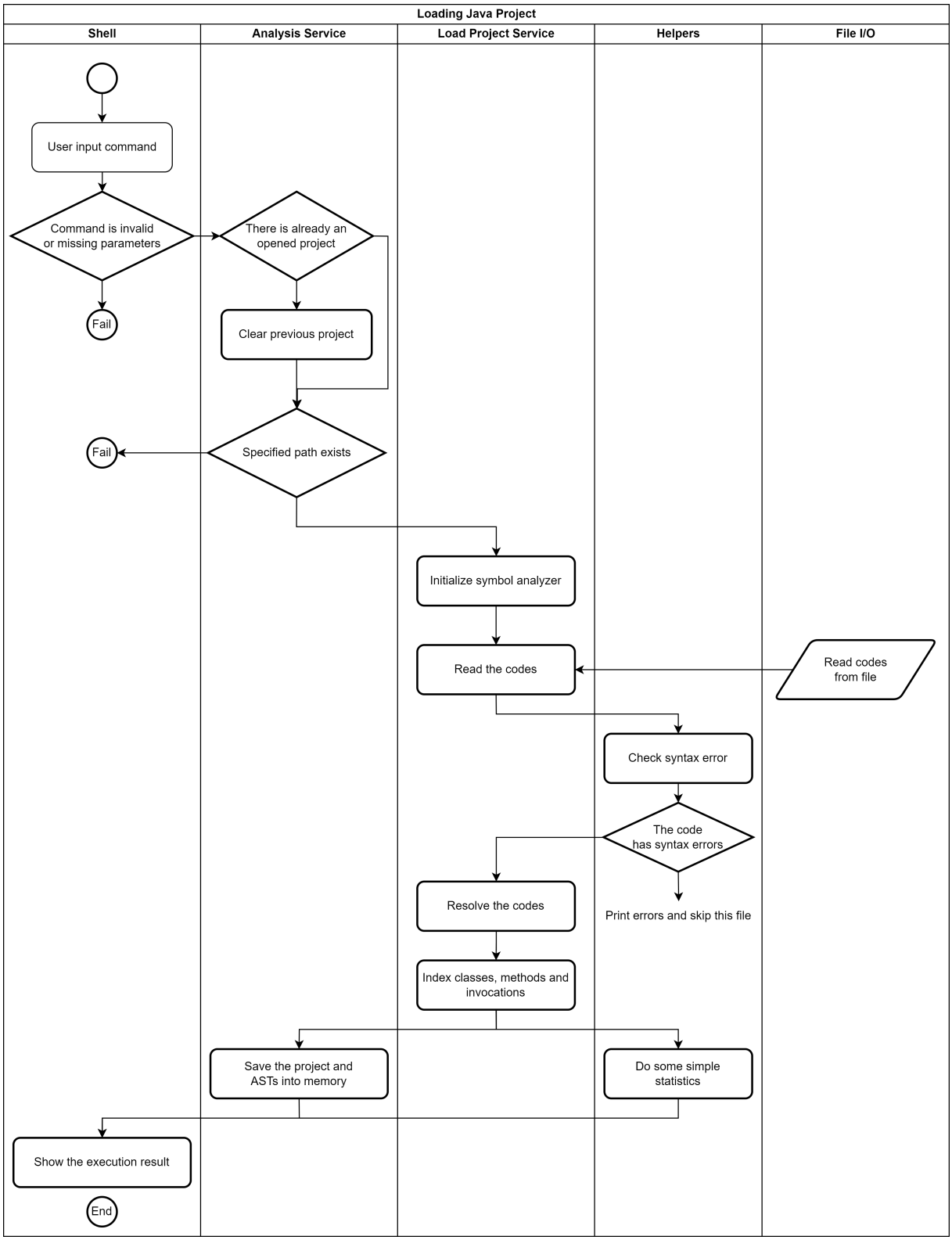

Figure 3: Project loading workflow

Methods invocation relationship querying workflow

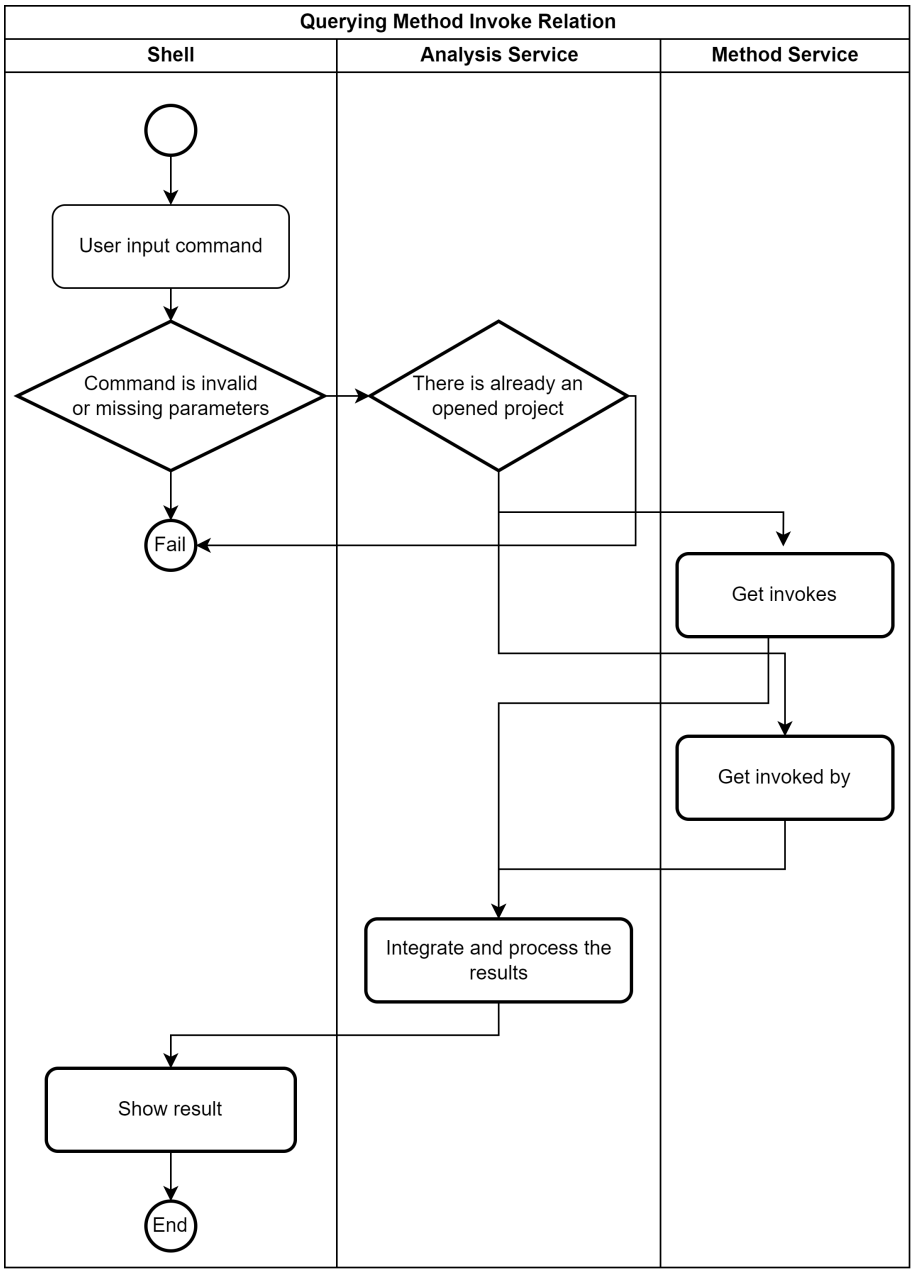

Figure 4: Methods invocation relationship querying workflow

Methods parameters origin querying workflow

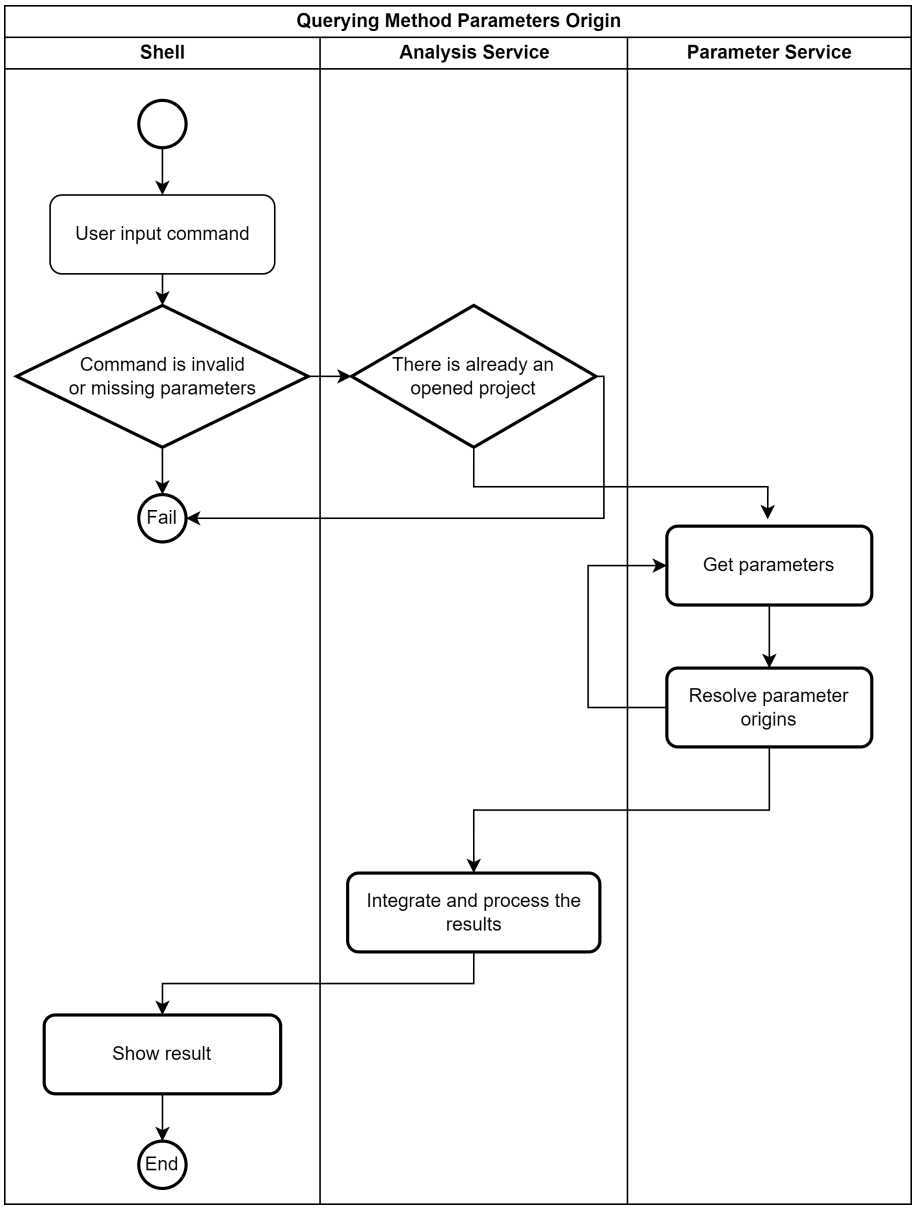

Figure 5: Methods parameters origin querying workflow

Implementation

Major data structures

Abstracting a Java project is a complex task because Java projects are hierarchically composed of packages, files, classes and interfaces, methods, and numerous nested expressions and statements. Moreover, to enhance the efficiency of the program's execution, we need to store and index commonly used metadata, such as the classes and methods within the project. This is because if every time we need to access a specific class and have to search for it and its corresponding file level by level according to the package name, the program's execution efficiency would be terribly poor.

Based on our practical requirements and for the sake of utility and efficiency, we divide a Java project into five levels. Each level is represented and stored by a unique Java class and holds the data for the next level. They are: Java Project, Java Package, Java File, Java Class, and Java Method. Among them, the levels of Project, Class, and Method are the most commonly used, while the other two levels mainly serve to maintain the tree structure of the project.

Efficiency considerations

Key techniques

Test

Test Cases

- Case A: The most basic case
- Case B: Large-scale Java library case

We have chosen the commons-lang library, open-sourced by the Apache Foundation, as our test project. Apache Commons is an excellent project within the Java ecosystem, offering high-quality and reliable components for almost every aspect of Java. This library consists of over two hundred classes and more than three thousand methods, totaling 175,000 lines of code. Choosing such a vast project to test our own is a significant challenge. The version we selected is located at commit gbe417ff07.

Conclusion