Compiler Course Project

Mohammad Mahdi Abdollahpour

1. Introduction

The goal of this project is to implement a software to mimic the functionality of Contain, ContainIn, Extend Couple, and Extend CoupleBy queries in the Scitools Understand software analysis toolkit. I used ANTLRv4 in C# runtime along with SQLite database to save the results. To find more information about Understand, the database schemas and the terminology visit here.

2. Problem Description

2.1. Contain and ContainIn

In Understand, Contain and ContainIn queries are inverse of each other. I will just explain Contain functionality. This query calculates all the references between the following entities and their containing package:

- Java Class
- Java Interface
- Java Enum
- Java Annotation Type

To be more specific, Understand searches all .java files and finds type declarations for the above entities and foreach of the declaration, it defines a Contain reference with Id of the declared entity (_ent_id in the database) and Id of the package (_scope_id in the database).

In addition to the user defined entities, Understand defines Contain references based on import statements. For statements like java.x.y.C, it defines C as an unknown class entity and java.x.y as an unknown package and defines a Contain reference between them. The rationale is that it assumes that the java.x.y package should contain the definition for C. For statements like java.x.y.*, it searches the whole source file to find any usage of any unknown class. However Understand can not successfully define an entity with a meaningful long name (package name + class/interface name) for such cases, thus it fills the reference table and entity table with a lot of useless orphan entities and references. Because of this, I ignored the second type of import statements in my implementation.

2.2. Extend Couple and Extend CoupleBy

This query finds relationships in the form of ClassA extends ClassB. This relationship can be between both classes and interfaces. Moreover, according to Understand, this relationship can

be in the form of <U extends ClassA> which is part of bounded type arguments in java. Like the previous reference types, Understand creates reference rows in the referencemodel table by referring to the ids of ClassA and ClassB for the former. For the latter, Understand creates dummy entities in the form of ClassA.U with no content and refers to their id when creating the reference record. Although there is logically a coupling between the class or interface that uses bounded parameters and ClassA, I ignored this form due to the malformed definition of the dummy entity proposed by Understand.

3. Implementation

In an overview, my implementation is based on the Visitor class defined in ANTLR. I started exploring the parse tree from the compilation unit and recursively searched for the required entities. More specifically, I looked for type declarations and import statements and added saved enough details e.g., the file paths. Then I extracted entities from the data borrowed from the previous step. In the final step, I constructed the previously-mentioned references using two former steps data. Below I further explained the core parts of the source code.

JavaExplorer: It is a high-level class that explores .java files in the source directory and calls JavaPackageVisitor on each of them. It saves extracted entities and references using the visitor. It also has a method to export the extracted data into the SQLite database using the Dapper ORM.

JavaPackageVisitor: It is an ANTLR visitor. It's main responsibility is to find entities (classes, interfaces, enums, annotation types, and packages) and the references between them.

EntityKindUtils: It contains tools to extract exact entity kind names according to the predefined OpenUnderstand database. It uses the information from the ANTLR context (mainly the modifiers) to construct the kind name string. JavaExplorer uses these names to find the correct entity id from the database.

4. Results

Like all clone softwares there will be some differences in comparison to their original counterpart in terms of functionalities and outcome. I have run both implementations over benchmark projects and have written many SQL queries to compare the results between my implementation and Understand. Here, I will mention the most important ones.

I first compared entity records in the entitymodel table. I found no needed entity records in the Understand result that does not appear in my implementation. All files, packages, classes, interfaces, enums, etc. have been found correctly. The kind_id column correctly corresponds to the type of the entity. parent_id column is correctly filled with the corresponding parent

entities. names and longnames are filled exactly the same. Also, the contents column is correctly filled with the exact source text of the corresponding entity. There is only one major difference in this table that also affects the results of other sections: Nested classes and interfaces are not present in the table which is expected according to my implementation and the Java grammar that I used. In this grammar, nested class and interface declarations are not defined as type declarations and have different and distinct definitions. Nested classes and interfaces in a class are part of member declaration rule and nested types in interfaces are part of interface member declaration rule; hence, they need their own specific checks while analyzing the parse tree. Moreover, since these entities can be nested in arbitrary depths, their parents are required to be tracked carefully which makes the analysis relatively tedious.

Additionally, I checked the referencemodel table. Considering the differences mentioned in previous sections, there is no major difference in this table. However, it is worth mentioning the minor ones. In Understand, the inverse reference of Extend Couple is ExtendBy CoupleBy with id of 240, however in my implementation it is Extend CoupleBy. This is due openunderstand implementation of pre-filling the database which does not contain the former. However, it is very easy to change the string if needed in my implementation. Another minor difference is the column number of the Extend reference. I used the column in which the original class is defined (ClassA in the first example). However, Understand uses the column where the referenced class is written. There are also a few references that are marked as Extend by Understand which make no sense because the corresponding entities are methods and they do not even include the extends keyword.

5. Future Works

There are multiple possibilities to improve the current project. Currently, the most crucial lacking is the non-existence of a shared module to detect entities and store their details in a set of well-defined data structures. The current development method, which comprises totally separate groups who implement a subset of references, is far below the optimal point. Each group needs to waste a considerable amount of time reimplementing many shared parts, especially regarding the entity finder part. This issue can be resolved by proper task assignment and project management methodology. Moreover, implementing the necessary checks to find the nested classes and interfaces can be considered as another important improvement. An automated and smart testing mechanism can be also beneficial, although it may be relatively challenging. This is because one-to-one naive record matching may not always be very helpful. There may be some unwanted or useless behavior from Understand as mentioned earlier which is not ideal to be copied. Hence, an automated test-suite that is based on the definition of done for each task would be more appealing for the developers.

6. Usage

6.1. Run from the source

In order to use this project, you have to follow these steps:

1. Clone the OpenUnderstand project.
2. Modify openunderstand/ound.py
    a. Change dbname and project directory based on your need
3. Build and run the C# project using two command line arguments separated with space:
    a. java project directory path
    b. SQLite .db file path

Example: There are some java projects in the benchmark directory at the OpenUnderstand repo. To run on benchmark/jhotdraw-develop we can do:

1. git clone https://github.com/m-zakeri/OpenUnderstand/
2. cd OpenUnderstand/openunderstand
3. Modify openunderstand/ound.py
    a. dbname=database_ound.db
    b. project_dir=Path('../benchmark/jhotdraw-develop').resolve().as_posix()
4. python ound.py
5. cd ../../UndContain/UndContain
6. dotnet run ../../OpenUnderstand/benchmark/jhotdraw-develop ../../OpenUnderstand/openunderstand/database_ound.db

6.2. Python script

I prepared a python script (run.py) that creates the database and calls the executable binary of the C# project. To use this script you need to fill the strings for the following variables:

- ound_exec: Should point to the C# executable. Use the .exe file if run on Windows
- prj_dir: Should point to the root of a java project that you want to analyze
- db_path: Path of the SQLite db storing the results
- refs: Exact string representation of the required references