

Compiler project document

**Group 10**

**Group members:**

**Ali Sedaghi**

**Mohammad Yarmoghadam**

**Amir Masoud Golestane**

**Hasti Karamdel**

[Introduction 4](#_Toc1933659474)

[What is Understand? 4](#_Toc986973901)

[This project 4](#_Toc1242027326)

[Our sections 5](#_Toc64351599)

[Project division 6](#_Toc478206189)

[What is expected? 6](#_Toc2078457151)

[Implementation 8](#_Toc631888716)

[Initializing 8](#_Toc1298002101)

[Import/Importby 13](#_Toc1335529070)

[Modify (Deref) Partial and Modifyby (Deref) Partial 19](#_Toc2133972446)

[Tests and results 25](#_Toc241590522)

[Procedure and Challenges 33](#_Toc1332036837)

[Conclusions and Recommendations 35](#_Toc628066678)

[Additional Sources 36](#_Toc1455779300)

# **Introduction**

## What is Understand?

Understand is a static analysis tool focused on source code comprehension, metrics, and standards testing. It is designed to help maintain and understand large amounts of legacy or newly created source code. It provides a cross-platform, multi-language, maintenance-oriented IDE (interactive development environment).

Understand uses more than 50 different graphs to help you visualize exactly what your code is doing and how it is built. Browse call trees, explore dependencies, verify UML structures or design your own graphs with the API.

Understand has architecture features that help you create hierarchical aggregations of source code units. You can name these units and manipulate them in various ways to create interesting hierarchies for analysis.

## This project

Unfortunately, the Understand API source code is not publicly available, making it difficult to change, customize, and reuse in new activities and environments which appears in academic researches.

This project aims to provide an open-source implementation of the Understand Python API to analyze the source codes. We primarily focus on implementing the API for Java programs using Python programming languages and compiler tools such as ANTLR. To develop an open-source implementation of Understand Python API, we look at the structures used by Understand for analyzing source codes.

## 

## Our sections

Most of the data captured by Understand involves Entities and References.

Entity: An Entity is anything in the code that Understand captures information on: i.e., A file, a class, a variable, a function, etc. In the Perl API, Entities are represented with the Understand::Ent class. In Python, it is the Understand.Ent class.

Reference: A specific place where an entity appears in the code. A reference is always defined as a relationship between two entities. e.g., function Bar is called on line 14 of function Foo. In the Perl API, References are represented with the Understand::Ref class. In Python, it is the Understand.Ref class.

Every entity and reference have a unique set of attributes that can be queried by the API. A few of the attributes you can view for an entity would be its name, its type, any associated comments, what kind of entity it is, and if it has them: its parent entity and its parameters. On the other hand, a reference would have both of the entities associated with it as well as the file, line, and column where the reference occurs and what kind of reference it is.

This project should support different reference kinds that we can have in our java code, each of these references can also use different entities. This is the table of the reference kinds which are each implemented by a different group.

You can see the reference kinds in this table:

<https://m-zakeri.github.io/OpenUnderstand/reference_kinds/>

You can see the entity kinds in this table:

<https://m-zakeri.github.io/OpenUnderstand/entity_kinds/>

Our group has tried to implement a code for the entities of Import/Import by & Modify (Deref) Partial and Modifyby (Deref) Partial. To recognize these entities, we have to use different entity kinds to Analize our java code and find the references of each time they are used.

## Project division

We decided it is best to decide the project to different parts we could each do separately and also a part we could all participate together. Therefor two of us worked on import/import by and the two other worked on modify/modify by. In the end we all gathered to study and test our codes and write this documentation.

## What is expected?

We need to implement a standalone python code which scans through a java project. Then it must be able to detect the reference kind we want (import or modify) and fill our database with the attributes we need for each kind.

Import/ImportBy:

Java Import Demand indicates a file has an on-demand import statement for a package or class. For example, if a file named file.java has imported a class, we would store the following data in our table:

* File id
* Imported class
* Entity kind id
* Line and Col in which import is used
* And also the responding entity which is imported by the importing entity

Reference:

<https://m-zakeri.github.io/OpenUnderstand/reference_kinds/#java-import-and-importby>

Modify (Deref) Partial and Modifyby (Deref) Partial:

Java Modify and Modifyby indicates that a variable’s value is modified or both read and set, as with the increment (++), decrement (--), and assignment/operator combinations (\*=, /=, ...). This reference is like Modify and Modifyby. But, it is used when an entity modifies some elements of a variable of collection type. Same as import we also need to detect in which scope a variable is being modified, for example if it has been modified in a function, we would add the following data to our table:

* File id
* Scope id
* Entity kind id
* Line and Col in which import is used
* And also the responding entity which is modified by the modifying entity

Reference:

<https://m-zakeri.github.io/OpenUnderstand/reference_kinds/#java-modify-and-modifyby>

**All of the implementation codes can be found here:**

<https://github.com/mohammadym/OpenUnderstand>

# **Implementation**

## Initializing

This section of the code is repeated in both of our files, because we implemented a standalone python code for each of the commands.

Imports:

First, we need to import antlr, Java Lexer, Parser and Listener. Then to connect and use the database we need to import the API and functions needed to fill the tables, the models for entity kinds and etc.

import os  
from antlr4 import \*  
from pathlib import Path  
from gen.javaLabeled.JavaLexer import JavaLexer  
from gen.javaLabeled.JavaParserLabeled import JavaParserLabeled  
from gen.javaLabeled.JavaParserLabeledListener import JavaParserLabeledListener  
from oudb.fill import main as db\_fill  
from oudb.api import create\_db, open as db\_open  
from oudb.models import KindModel, EntityModel, ReferenceModel

Benchmark settings:

To run and test each of the benchmark projects we need to set a project index and add the names of the projects that we need to be tested. We also need to add a path so that the project files can be read and tested from there, to handle this matter we have made a similar path for each of the projects and the only thing that differs each path is the project name itself; resulting an array for the paths of each project. We also need to set the path of our database file which includes the final results and tables.

PRJ\_INDEX = 3  
REF\_NAME = "import"  
  
  
def get\_project\_info(index, ref\_name):  
 project\_names = [  
 'calculator\_app',  
 'JSON',  
 'testing\_legacy\_code',  
 'jhotdraw-develop',  
 'xerces2j',  
 'jvlt-1.3.2',  
 'jfreechart',  
 'ganttproject',  
 '105\_freemind',  
 ]  
 project\_name = project\_names[index]  
 db\_path = f"../../databases/{ref\_name}/{project\_name}"  
 if ref\_name == "origin":  
 db\_path = db\_path + ".udb"  
 else:  
 db\_path = db\_path + ".oudb"  
 project\_path = f"../../benchmarks/{project\_name}"  
  
 db\_path = os.path.abspath(db\_path)  
 project\_path = os.path.abspath(project\_path)  
  
 return {  
 'PROJECT\_NAME': project\_name,  
 'DB\_PATH': db\_path,  
 'PROJECT\_PATH': project\_path,  
 }

**Init:**

In the beginning of the class, we set the database name, project name and directory and also the two arrays with the names and paths of the files.

def \_\_init\_\_(self, db\_name, project\_dir, project\_name=None):  
 self.db\_name = db\_name  
 self.project\_dir = project\_dir  
 self.project\_name = project\_name  
 self.files = []

Initdb:

We used createdb to create our database then used the fill function to add the models, the database is added to the path we gave it in the beginning and is ready to use.

def init\_db(self):  
 create\_db(self.db\_name, self.project\_dir, self.project\_name)  
 db\_fill()  
 db\_open(self.db\_name)

Get Java Files:

The project directories that we have include many files, that is why we have to find the .java files in order to continue. For each java file we add the name and path to the arrays we have.

def get\_java\_files(self):  
 for dir\_path, \_, file\_names in os.walk(self.project\_dir):  
 for file in file\_names:  
 if '.java' in str(file):  
 path = os.path.join(dir\_path, file)  
 path = path.replace("/", "\\")  
 path = os.path.abspath(path)  
 self.files.append((file, path))  
 add\_java\_file\_entity(path, file)

Get Parent:

This function checks our database for the parent entity if there is a parent entity with the id of the “java file” entity kind and also the name and the path given, it will return its object.

def get\_parent(parent\_file\_name, files):  
 file\_names, file\_paths = zip(\*files)  
 parent\_file\_index = file\_names.index(parent\_file\_name)  
 parent\_file\_path = file\_paths[parent\_file\_index]  
 parent\_entity = EntityModel.get\_or\_none(  
 \_kind=KindModel.get\_or\_none(\_name="Java File").get\_id(),  
 \_name=parent\_file\_name,  
 \_longname=parent\_file\_path,  
 )  
 return parent\_entity, parent\_file\_path

Add Java file entity:

This function gets each java file name and path from the arrays and sets its entity in the entity kinds table, or if there already is an id for that entity, it will return the object.

def add\_java\_file\_entity(file\_path, file\_name):  
 kind\_id = KindModel.get\_or\_none(\_name="Java File").get\_id()  
 obj, \_ = EntityModel.get\_or\_create(  
 \_kind=kind\_id,  
 \_name=file\_name,  
 \_longname=file\_path,  
 \_contents=FileStream(file\_path, encoding="utf-8"),  
 )  
 return obj

Get parse tree:

This function gets the file from the file path given(using fileStream) then creates a java lexer for that file. From that lexer we can tokenize it and make our parser.

def get\_parse\_tree(file\_path):  
 file = FileStream(file\_path, encoding="utf-8")  
 lexer = JavaLexer(file)  
 tokens = CommonTokenStream(lexer)  
 parser = JavaParserLabeled(tokens)  
 return parser.compilationUnit()

## Import/Importby

After getting all of the java files and adding the entity kinds to our database table, we need to focus on finding the import references in our files.

Import Listener:

The goal of this class is to create a list of objects for each “import” in our file. The object should have the attributes needed to reference an import. At first we get the Imported classes name and check if it already is in the array of names we created before, because if it isn’t that means our imported class is a java built-in class type. Then with the help of our listener we get the line and column where the import is used in our file. After setting all the attributes we shape our object for that import and add it to the list.

class ImportListener(JavaParserLabeledListener):  
 def \_\_init\_\_(self, files):  
 self.repository = []  
 self.files = files  
  
 def enterImportDeclaration(self, ctx: JavaParserLabeled.importDeclaration):  
 imported\_class\_longname = ctx.qualifiedName().getText()  
 imported\_class\_name = imported\_class\_longname.split('.')[-1]  
  
 is\_built\_in = False  
 imported\_class\_file\_name = imported\_class\_name + ".java"  
 if imported\_class\_file\_name not in [file[0] for file in self.files]:  
 is\_built\_in = True  
 imported\_class\_file\_name = None  
  
 line = ctx.children[0].symbol.line  
 col = ctx.children[0].symbol.column  
  
 self.repository.append({  
 'imported\_class\_name': imported\_class\_name,  
 'imported\_class\_longname': imported\_class\_longname,  
 'is\_built\_in': is\_built\_in,  
 'imported\_class\_file\_name': imported\_class\_file\_name,  
 'line': line,  
 'column': col,  
 })

Imported Entity Listener:

Same as the previous class this listener indicates the kind of our imported entity

class ImportedEntityListener(JavaParserLabeledListener):  
 def \_\_init\_\_(self, name):  
 self.body = None  
 self.branches = None  
 self.type = None  
 self.name = name  
  
 def enterClassDeclaration(self, ctx: JavaParserLabeled.ClassDeclarationContext):  
 if self.name == ctx.IDENTIFIER().getText():  
 self.body = ctx.getText()  
 self.branches = ctx.parentCtx.children  
  
 def enterInterfaceDeclaration(self, ctx: JavaParserLabeled.InterfaceDeclarationContext):  
 if self.name == ctx.IDENTIFIER().getText():  
 self.body = ctx.getText()  
 self.branches = ctx.parentCtx.children  
  
 def enterEnumDeclaration(self, ctx: JavaParserLabeled.EnumDeclarationContext):  
 if self.name == ctx.IDENTIFIER().getText():  
 self.body = ctx.getText()  
 self.branches = ctx.parentCtx.children

Add Imported Entity:

At first we will check if the entity is a built in java class the kindid of it should be set to the id of "Java Unknown Class Type Member", also this kind of class doesn’t have a parent. If the entity isn’t built in we will set its kind using get kind method. We will also get the parent of this entity from the functions explained. At last we will get the class body to add the contents of that class to our table.

def add\_imported\_entity(i, files):  
 if i['is\_built\_in']:  
 imported\_entity, \_ = EntityModel.get\_or\_create(  
 \_kind=KindModel.get\_or\_none(\_name="Java Unknown Class Type Member").get\_id(),  
 \_parent=None,  
 \_name=i['imported\_class\_name'],  
 \_longname=i['imported\_class\_longname'],  
 )  
 else:  
 parent\_entity, parent\_file\_path = get\_parent(i['imported\_class\_file\_name'], files)  
 prefixes, class\_body, kind = get\_imported\_entity(parent\_file\_path)  
 entity\_kind = get\_kind\_name(prefixes, kind)  
 imported\_entity, \_ = EntityModel.get\_or\_create(  
 \_kind=KindModel.get\_or\_none(\_name=entity\_kind).get\_id(),  
 \_parent=parent\_entity.get\_id(),  
 \_name=i['imported\_class\_name'],  
 \_longname=i['imported\_class\_longname'],  
 \_contents=class\_body,  
 )  
 return imported\_entity

Get Imported Entity:

This method returns the imported entity attributes. At first we gret the parse tree of the path we have, and we also pass the file path to the imported entity listener explained before. After getting our listener, we can start walking in the tree. As we walk in the tree we check if the kind of our branch is class, interface or enum. Therefore set the corresponding attributes and return it.

def get\_imported\_entity(file\_path):  
 tree = get\_parse\_tree(file\_path)  
 listener = ImportedEntityListener(Path(file\_path).stem)  
 walker = ParseTreeWalker()  
 walker.walk(listener=listener, t=tree)  
  
 prefixes = ""  
 kind = ""  
 for branch in listener.branches:  
 if type(branch) == JavaParserLabeled.ClassDeclarationContext:  
 kind = "Class"  
 break  
 elif type(branch) == JavaParserLabeled.InterfaceDeclarationContext:  
 kind = "Interface"  
 break  
 elif type(branch) == JavaParserLabeled.EnumDeclarationContext:  
 kind = "Enum Class"  
 break  
 prefixes += branch.getText() + " "  
 return prefixes, listener.body, kind

Kind Type:

This method helps us figure out the exact kind name for each entity. After checking the prefixes we can assemble them in one string(in the right order) as the final name.

def get\_kind\_name(prefixes, kind):  
 p\_static = ""  
 p\_abstract = ""  
 p\_generic = ""  
 p\_type = "Type"  
 p\_visibility = "Default"  
 p\_member = "Member"  
  
 if "static" in prefixes:  
 p\_static = "Static"  
  
 if "generic" in prefixes:  
 p\_generic = "Generic"  
  
 if "abstract" in prefixes:  
 p\_abstract = "Abstract"  
 elif "final" in prefixes:  
 p\_abstract = "Final"  
  
 if "private" in prefixes:  
 p\_visibility = "Private"  
 elif "public" in prefixes:  
 p\_visibility = "Public"  
 elif "protected" in prefixes:  
 p\_visibility = "Protected"  
  
 if kind == "Interface":  
 p\_member = ""  
  
 if kind == "Method":  
 p\_type = ""  
  
 s = f"Java {p\_static} {p\_abstract} {p\_generic} {kind} {p\_type} {p\_visibility} {p\_member}"  
 s = " ".join(s.split())  
 return s

Add references:

This function gets the imported and importing entity and adds them in the final table of our database.

def add\_references(importing\_ent, imported\_ent, ref\_dict):  
 ref, \_ = ReferenceModel.get\_or\_create(  
 \_kind=KindModel.get\_or\_none(\_name="Java Import").get\_id(),  
 \_file=importing\_ent.get\_id(),  
 \_line=ref\_dict['line'],  
 \_column=ref\_dict['column'],  
 \_ent=imported\_ent.get\_id(),  
 \_scope=importing\_ent.get\_id(),  
 )  
 inverse\_ref, \_ = ReferenceModel.get\_or\_create(  
 \_kind=KindModel.get\_or\_none(\_name="Java Importby").get\_id(),  
 \_file=importing\_ent.get\_id(),  
 \_line=ref\_dict['line'],  
 \_column=ref\_dict['column'],  
 \_ent=importing\_ent.get\_id(),  
 \_scope=imported\_ent.get\_id(),  
 )

Main:

First we make an instance of the project and call init\_db. After setting our arrays with the java files found, we need to match the corresponding name and path of each file, then for each file we find the importing entity as explained before. We need to use get\_parse\_tree to get our tree, then use the listener we made for import before so we can walk in our tree. Now we have our repository which is the list with all of our import objects. For each item in that list we need to find the imported entity using the add imported entity method explained before. At last we have our importing and imported entity and also the matching repository object in the listener so we pass them to add\_references which adds them to the final references table in our database.

def main():  
 info = get\_project\_info(PRJ\_INDEX, REF\_NAME)  
 p = Project(info['DB\_PATH'], info['PROJECT\_PATH'], info['PROJECT\_NAME'])  
 p.init\_db()  
 p.get\_java\_files()  
  
 for file\_name, file\_path in p.files:  
 importing\_entity = add\_java\_file\_entity(file\_path, file\_name)  
  
 tree = get\_parse\_tree(file\_path)  
 listener = ImportListener(p.files)  
 walker = ParseTreeWalker()  
 walker.walk(listener, tree)  
  
 for i in listener.repository:  
 imported\_entity = add\_imported\_entity(i, p.files)  
 add\_references(importing\_entity, imported\_entity, i)

## Modify (Deref) Partial and Modifyby (Deref) Partial

We have defined most of the initial functions and set ups in the previous sections, in this section we will describe the functions that are specifically designed to find the Modifyderef Partial command and shape our reference table in the database.

Get prefixes:  
In order to find the name of our entity kind with get kind name method(explained in the previous parts) we need to have these prefixes.  
  
def get\_prefixes(ctx, ctx\_type):  
 branches = ctx.parentCtx.children  
 prefixes = ""  
 for branch in branches:  
 if type(branch).\_\_name\_\_ == ctx\_type:  
 break  
 prefixes += branch.getText() + " "  
 return prefixes  
  
Class Listener:  
This class, is out listener which checks through our file and adds the classes to the database table of entities.   
  
class ClassListener(JavaParserLabeledListener):  
 def \_\_init\_\_(self, files, file\_name):  
 self.files = files  
 self.file\_name = file\_name  
  
 def enterClassDeclaration(self, ctx: JavaParserLabeled.ClassDeclarationContext):  
 parent\_entity, parent\_file\_path = get\_parent(self.file\_name, self.files)  
 prefixes = get\_prefixes(ctx, "ClassDeclarationContext")  
 kind\_name = get\_kind\_name(prefixes, kind="Class")  
 obj, \_ = EntityModel.get\_or\_create(  
 \_kind=KindModel.get\_or\_none(\_name=kind\_name).get\_id(),  
 \_parent=parent\_entity.get\_id(),  
 \_name=ctx.IDENTIFIER().getText(),  
 \_longname=parent\_file\_path,  
 \_contents=ctx.getText(),  
 )  
  
Modify deref Listener:  
This is the main part of this section’s code. Search scope checks current until it finds a type name that is in the type names list; and if it finds a class or method it returns current. In the following two methods we check whether the scope is a class or a method. Therefore we add the entity to our table setting the attributes kind, parent, name, long name using the methods explained before. At last we also add the id and content and return the object. The enter expressions 6 and 21 are for “=” and “+,-,\*,...” expression which can modify a variable. Last but not least we have the main function which after being called from the enter expressions, checks whether there is an opening for a scope and uses search scope to declare that. After that we check if the scope is a class or a method using those two functions explained earlier; and set the correct name and entity kind for each one of them and pass it on to add reference for the final result to be added to our database.   
  
class ModifyListener(JavaParserLabeledListener):  
 def \_\_init\_\_(self, files, file\_name):  
 self.files = files  
 self.file\_name = file\_name  
  
 @staticmethod  
 def search\_scope(ctx, type\_names):  
 # Traverse bottom up until reaching a class or method  
 current = ctx.parentCtx  
 while current is not None:  
 type\_name = type(current).\_\_name\_\_  
 if type\_name in type\_names:  
 return current  
 current = current.parentCtx  
 return None  
  
 def make\_scope\_class(self, ctx, file\_name):  
 prefixes = get\_prefixes(ctx, "ClassDeclarationContext")  
 kind\_name = get\_kind\_name(prefixes, kind="Class")  
 kind\_id = KindModel.get\_or\_none(\_name=kind\_name).get\_id()  
 name = ctx.IDENTIFIER().getText()  
 parent\_entity, parent\_file\_path = get\_parent(file\_name, self.files)  
 content = ctx.getText()  
  
 obj = EntityModel.get\_or\_none(  
 \_kind=KindModel.get\_or\_none(\_name=kind\_name).get\_id(),  
 \_parent=parent\_entity.get\_id(),  
 \_name=name,  
 \_longname=parent\_file\_path,  
 )  
 return {  
 "id": obj.get\_id(),  
 "kind\_id": kind\_id,  
 "parent\_id": parent\_entity.get\_id(),  
 "name": name,  
 "longname": parent\_file\_path,  
 "content": content  
 }  
  
 def make\_scope\_method(self, ctx, file\_name):  
 prefixes = get\_prefixes(ctx, "MethodDeclarationContext")  
 kind\_name = get\_kind\_name(prefixes, kind="Method")  
 kind\_id = KindModel.get\_or\_none(\_name=kind\_name).get\_id()  
 name = ctx.IDENTIFIER().getText()  
 content = ctx.getText()  
 parent\_ctx = self.search\_scope(ctx, ["ClassDeclarationContext"])  
 parent\_entity = self.make\_scope\_class(parent\_ctx, file\_name)  
  
 obj, \_ = EntityModel.get\_or\_create(  
 \_kind=kind\_id,  
 \_parent=parent\_entity['id'],  
 \_name=name,  
 \_longname=f"{parent\_entity['name']}.{name}",  
 \_contents=content,  
 )  
 return {  
 "id": obj.get\_id(),  
 "kind\_id": kind\_id,  
 "parent\_id": parent\_entity['id'],  
 "name": name,  
 "longname": f"{parent\_entity['name']}.{name}",  
 "content": content  
 }  
  
 def enterExpression6(self, ctx: JavaParserLabeled.Expression6Context):  
 self.modify\_deref\_partial(ctx)  
  
 def enterExpression21(self, ctx: JavaParserLabeled.Expression21Context):  
 self.modify\_deref\_partial(ctx)  
  
 def modify\_deref\_partial(self, ctx):  
 lhs\_text = ctx.children[0].getText()  
 scope = None  
 if "[" in lhs\_text and "]" in lhs\_text:  
 var\_name = ctx.children[0].children[0].getText()  
 var\_entity = add\_var\_entity(var\_name)  
  
 line, col = str(ctx.start).split(",")[3][:-1].split(':')  
  
 file\_entity, \_ = get\_parent(self.file\_name, self.files)  
 file\_id = file\_entity.get\_id()  
  
 ref\_dict = {'line': line, 'column': col, 'file\_id': file\_id, 'text': ctx.getText()}  
  
 scope\_ctx = self.search\_scope(ctx, ["ClassDeclarationContext", "MethodDeclarationContext"])  
 if type(scope\_ctx).\_\_name\_\_ == "ClassDeclarationContext":  
 scope = self.make\_scope\_class(scope\_ctx, self.file\_name)  
 elif type(scope\_ctx).\_\_name\_\_ == "MethodDeclarationContext":  
 scope = self.make\_scope\_method(scope\_ctx, self.file\_name)  
  
 add\_references(scope, var\_entity, ref\_dict)  
  
Add variable entity:  
Add each variable entity to the database table of entities.  
  
def add\_var\_entity(var\_name):  
 obj, \_ = EntityModel.get\_or\_create(  
 \_kind=KindModel.get\_or\_none(\_name="Java Unresolved Variable").get\_id(),  
 \_name=var\_name,  
 \_longname=var\_name,  
 )  
 return {  
 "id": obj.get\_id(),  
 "kind\_id": KindModel.get\_or\_none(\_name="Java Unresolved Variable").get\_id(),  
 "name": var\_name,  
 "longname": var\_name,  
 }  
  
Add references:  
Just as we explained in the last section, this method adds the final objects to our reference table in the database. The kind id is set to either “Java Modify Deref Partial” or “Java Modifyby Deref Partial”. The file, line and column in which the modifying has happened are all set in their attributes. We have also added the entity which does the action and the scope the action has taken place on.   
  
def add\_references(scope, ent, ref\_dict):  
 ref, \_ = ReferenceModel.get\_or\_create(  
 \_kind=KindModel.get\_or\_none(\_name="Java Modify Deref Partial").get\_id(),  
 \_file=ref\_dict['file\_id'],  
 \_line=ref\_dict['line'],  
 \_column=ref\_dict['column'],  
 \_ent=ent['id'],  
 \_scope=scope['id'],  
 )  
 inverse\_ref, \_ = ReferenceModel.get\_or\_create(  
 \_kind=KindModel.get\_or\_none(\_name="Java Modifyby Deref Partial").get\_id(),  
 \_file=ref\_dict['file\_id'],  
 \_line=ref\_dict['line'],  
 \_column=ref\_dict['column'],  
 \_ent=scope['id'],  
 \_scope=ent['id'],  
 )  
  
Main:  
After creating an instance of the project and initializing the database and setting the java files; we get then parse tree of each file and traverse through it. We make our listeners and walk through them, as explained before the expressions and scopes of the file are checked for a modified variable in the scope mentioned. After detecting and setting the attributes of each modify deref partial that has happened, we have the final results as a table.   
  
def main():  
 info = get\_project\_info(PRJ\_INDEX, REF\_NAME)  
 p = Project(info['DB\_PATH'], info['PROJECT\_PATH'], info['PROJECT\_NAME'])  
 p.init\_db()  
 p.get\_java\_files()  
  
 for file\_name, file\_path in p.files:  
 tree = get\_parse\_tree(file\_path)  
 walker = ParseTreeWalker()  
  
 class\_listener = ClassListener(p.files, file\_name)  
 listener = ModifyListener(p.files, file\_name)  
  
 walker.walk(class\_listener, tree)  
 walker.walk(listener, tree)

# **Tests and results**

There were several benchmark projects tested and compared, so we will share the final tables of a few results for some of the projects. (The rest can be found on our source files).

For testing our benchmark projects we added a g10 test file:

import understand as und  
import os   
  
PRJ\_INDEX = 0  
REF\_NAME = "origin"  
  
  
def get\_project\_info(index, ref\_name):  
 project\_names = [  
 'calculator\_app',  
 'JSON',  
 'testing\_legacy\_code',  
 'jhotdraw-develop',  
 'xerces2j',  
 'jvlt-1.3.2',  
 'jfreechart',  
 'ganttproject',  
 '105\_freemind',  
 ]  
 project\_name = project\_names[index]  
 db\_path = f"../../databases/{ref\_name}/{project\_name}"  
 if ref\_name == "origin":  
 db\_path = db\_path + ".udb"  
 else:  
 db\_path = db\_path + ".oudb"  
 project\_path = f"../../benchmarks/{project\_name}"  
  
 db\_path = os.path.abspath(db\_path)  
 project\_path = os.path.abspath(project\_path)  
  
 return {  
 'PROJECT\_NAME': project\_name,  
 'DB\_PATH': db\_path,  
 'PROJECT\_PATH': project\_path,  
 }  
  
  
def test\_understand\_kinds():  
 info = get\_project\_info(PRJ\_INDEX, REF\_NAME)  
 db = und.open(info['DB\_PATH'])  
 for ent in db.ents():  
 for ref in ent.refs("Import"):  
 print(f'1. ref name: {ref.kindname()}')  
 print(f'2. ref scope: {ref.scope().longname()} || kind: {ref.scope().kind()}')  
 print(f'3. ref ent: {ref.ent().longname()} || kind: {ref.ent().kind()}')  
 print(f'4. file location: {ref.file().longname()} || line: {ref.line()}')  
 print("-" \* 25)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test\_understand\_kinds()

Then we compared the results of our database to Understand’s results.

**The resulted tables are as follows:**

**Import(calculator app example)**

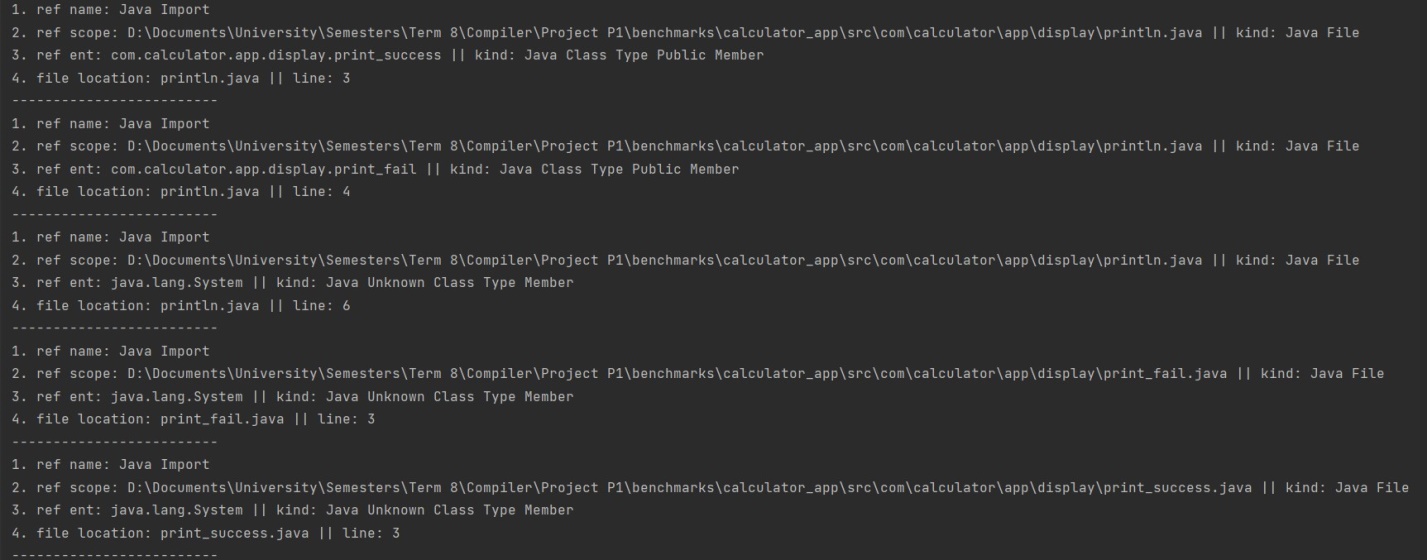


Figure 1. CMD results

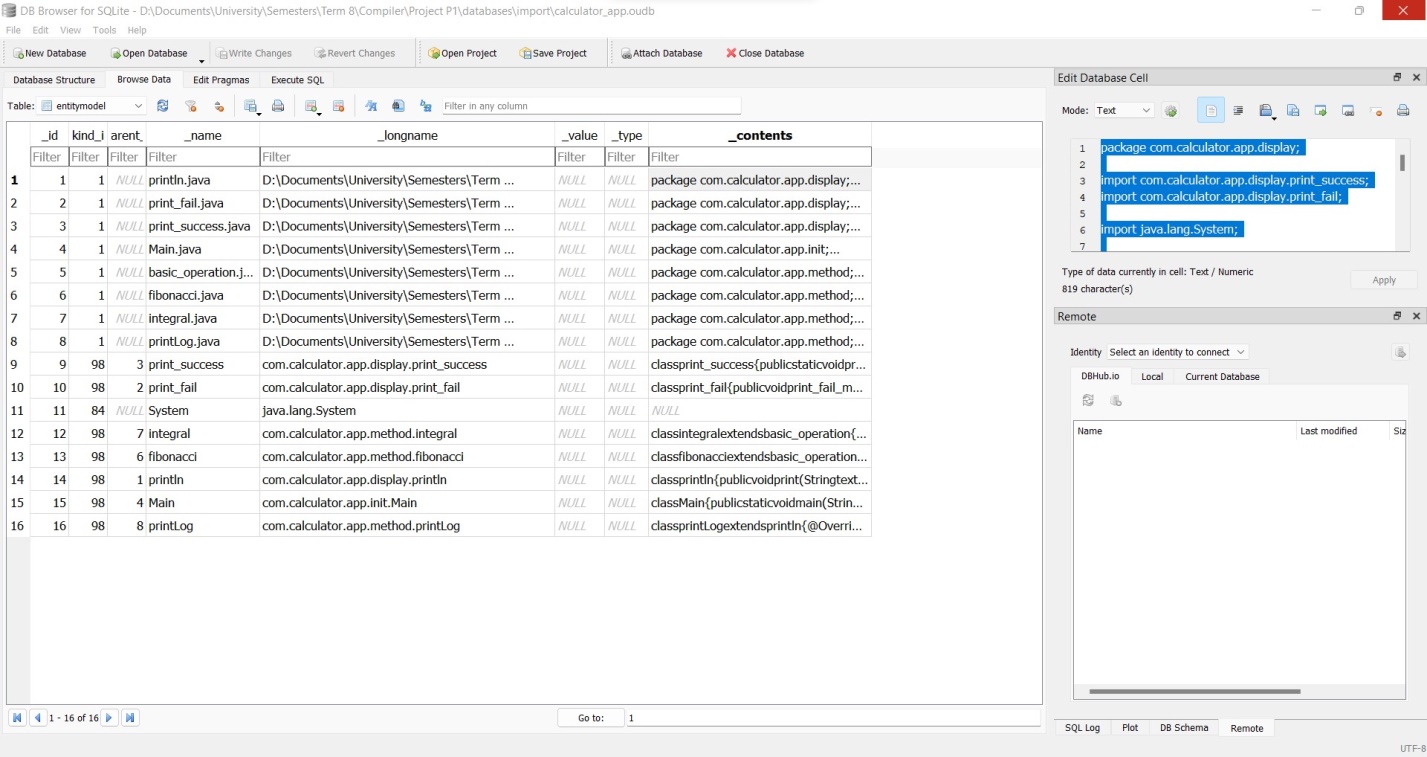


Figure 1. Database entity table

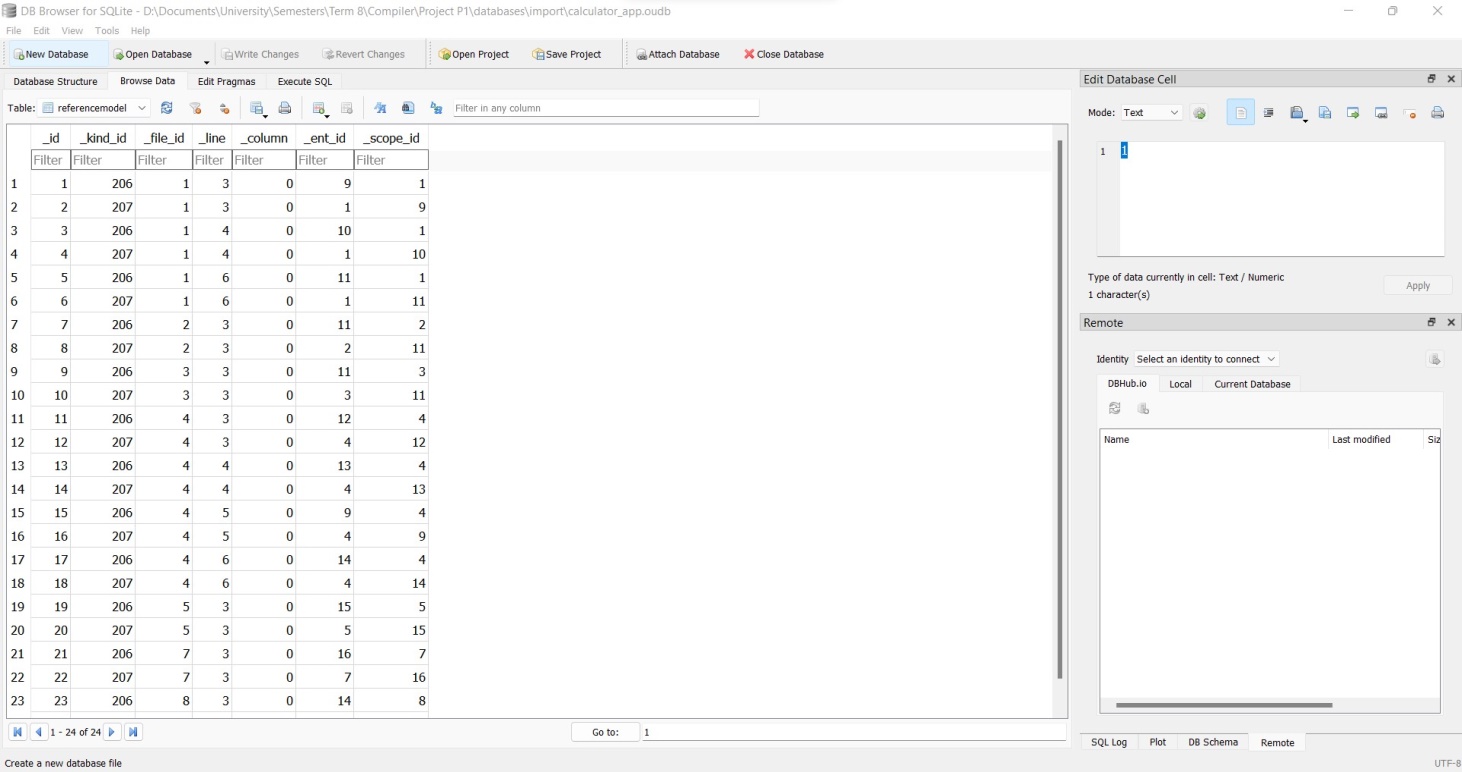


Figure 1. Database reference table

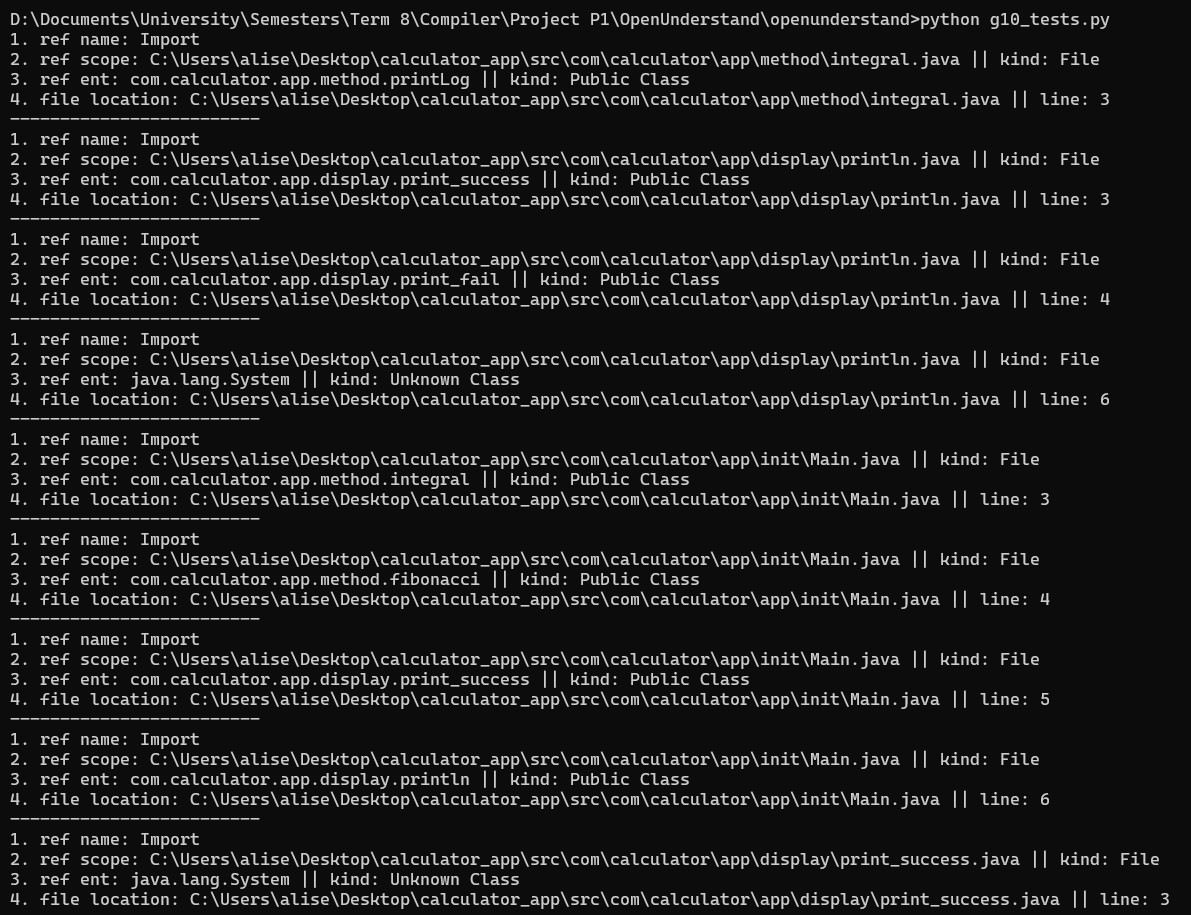


Figure 1. Understand results

**Import(xerces2j example)**

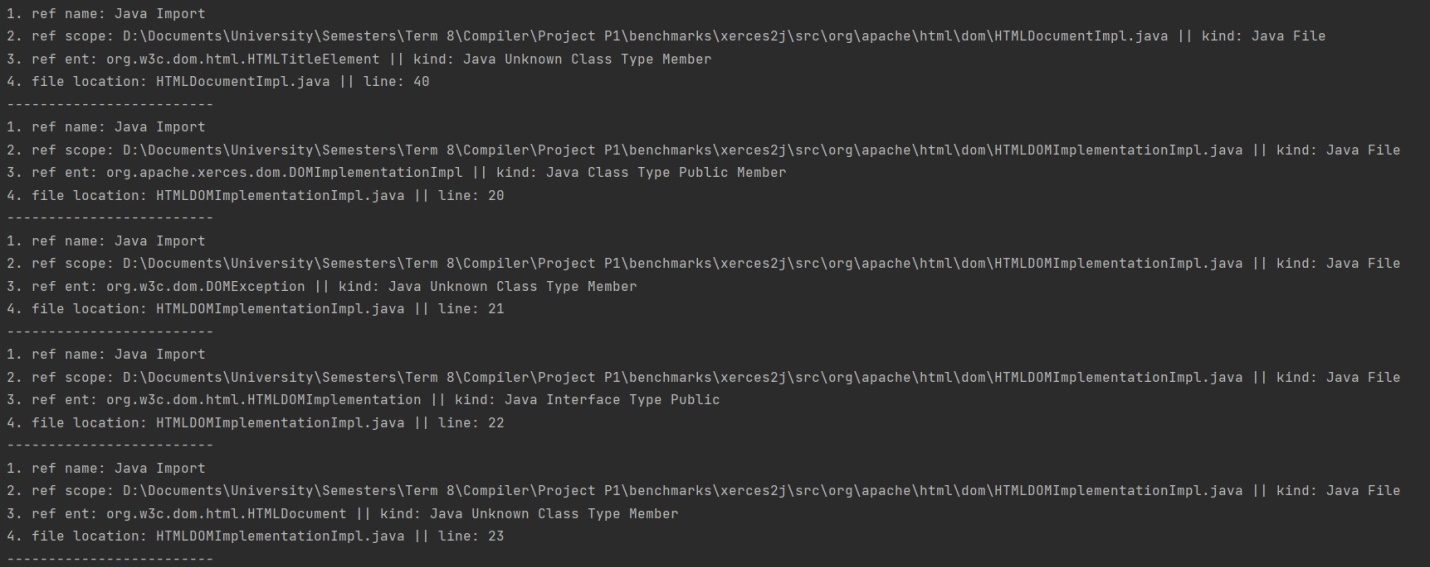


Figure 2.1 CMD results

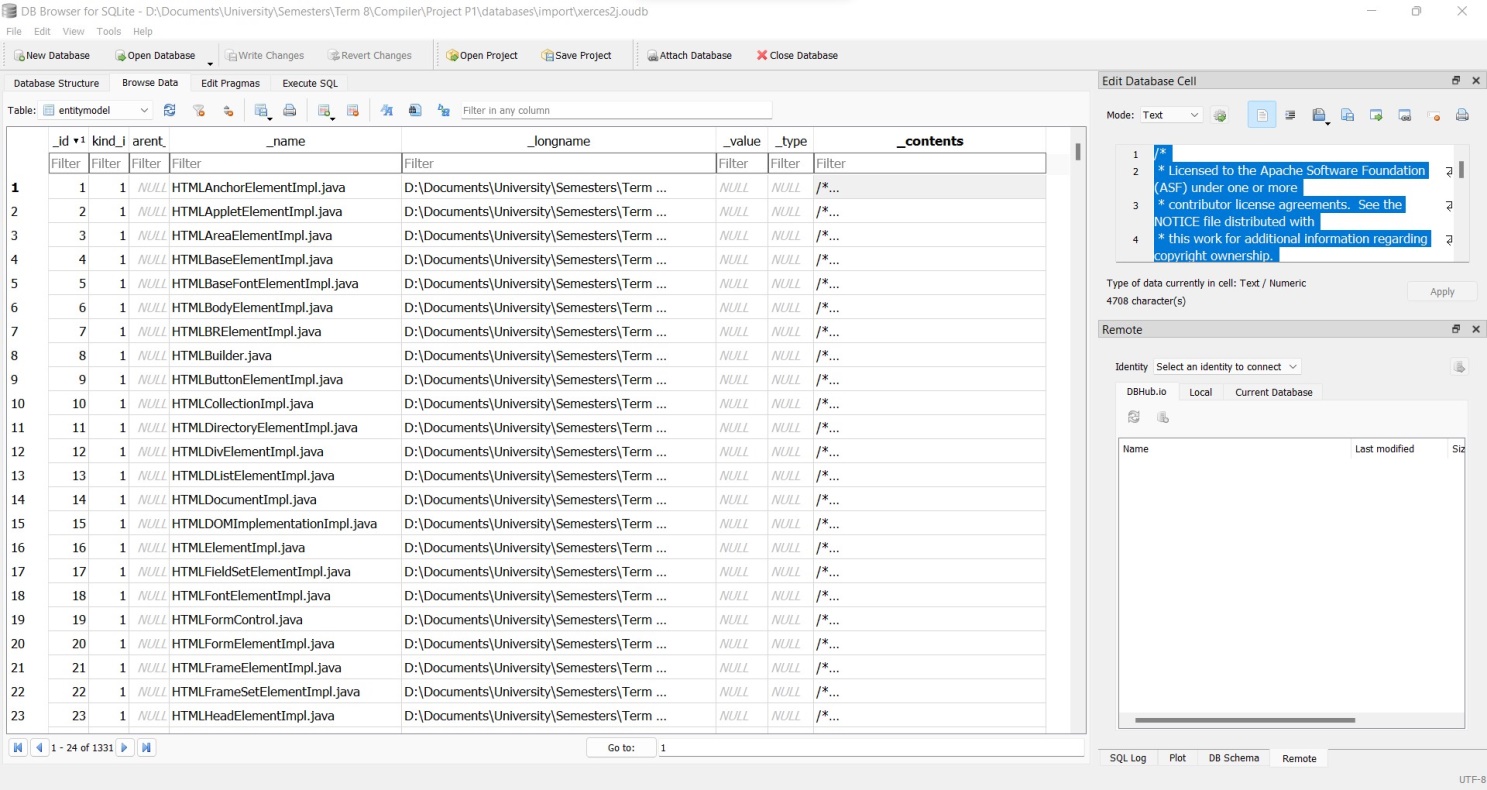


Figure 2.2 Database entity table

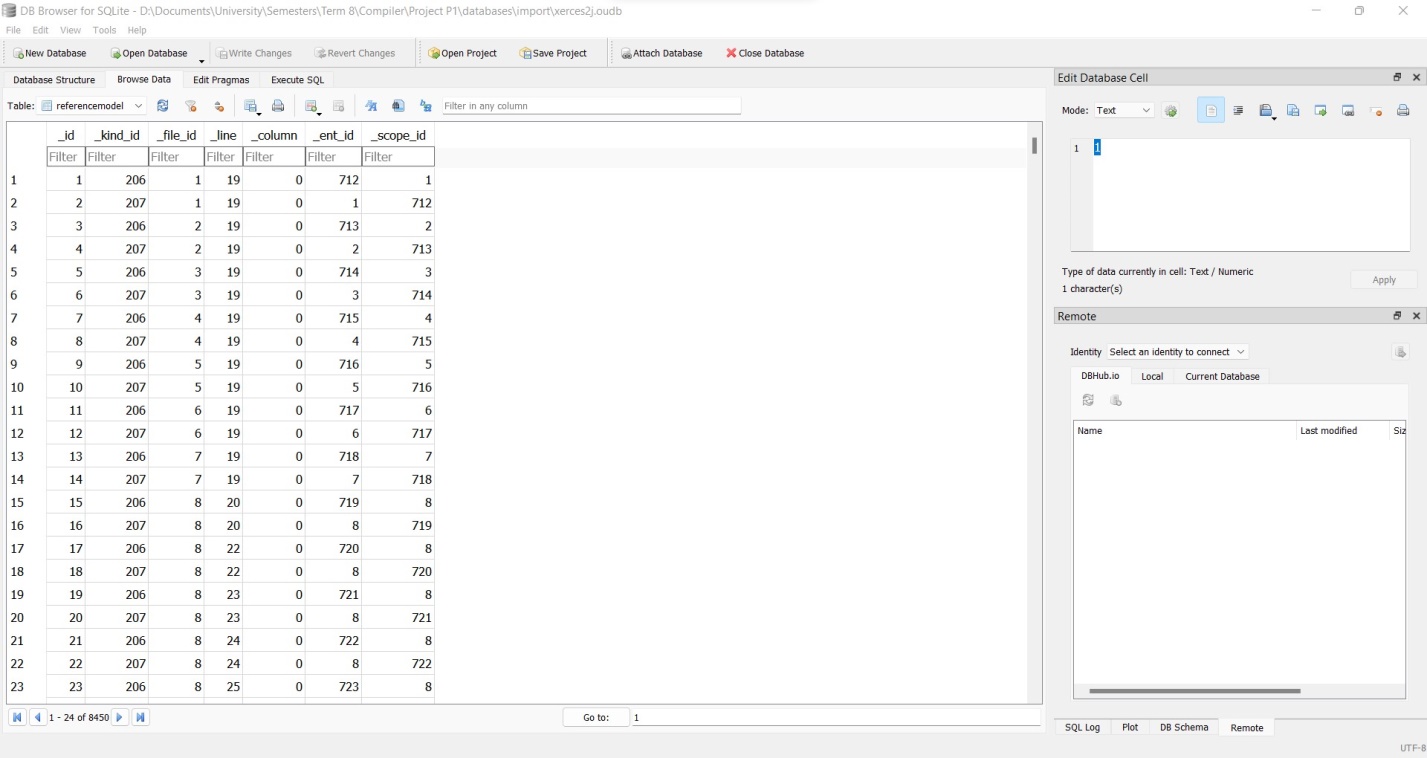


Figure 2.3 Database reference table



Figure 2.4 Understand results

**Modify deref(jhotdraw-develop example)**

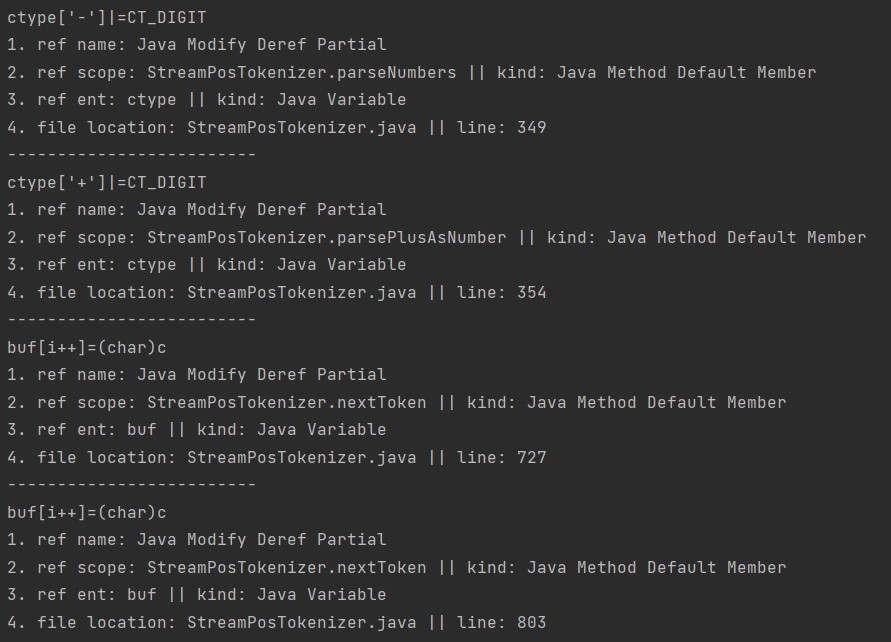


Figure 3.1 CMD results

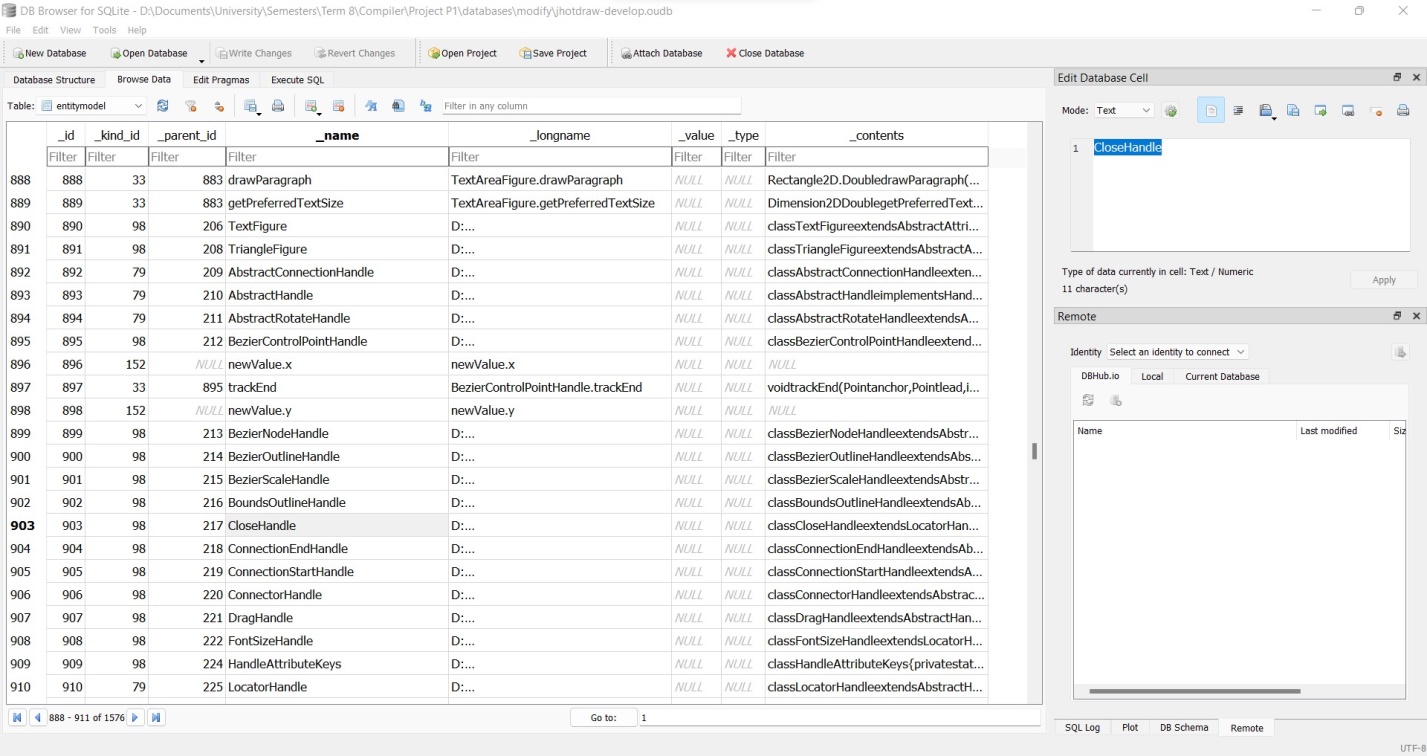


Figure 3.2 Database entity table

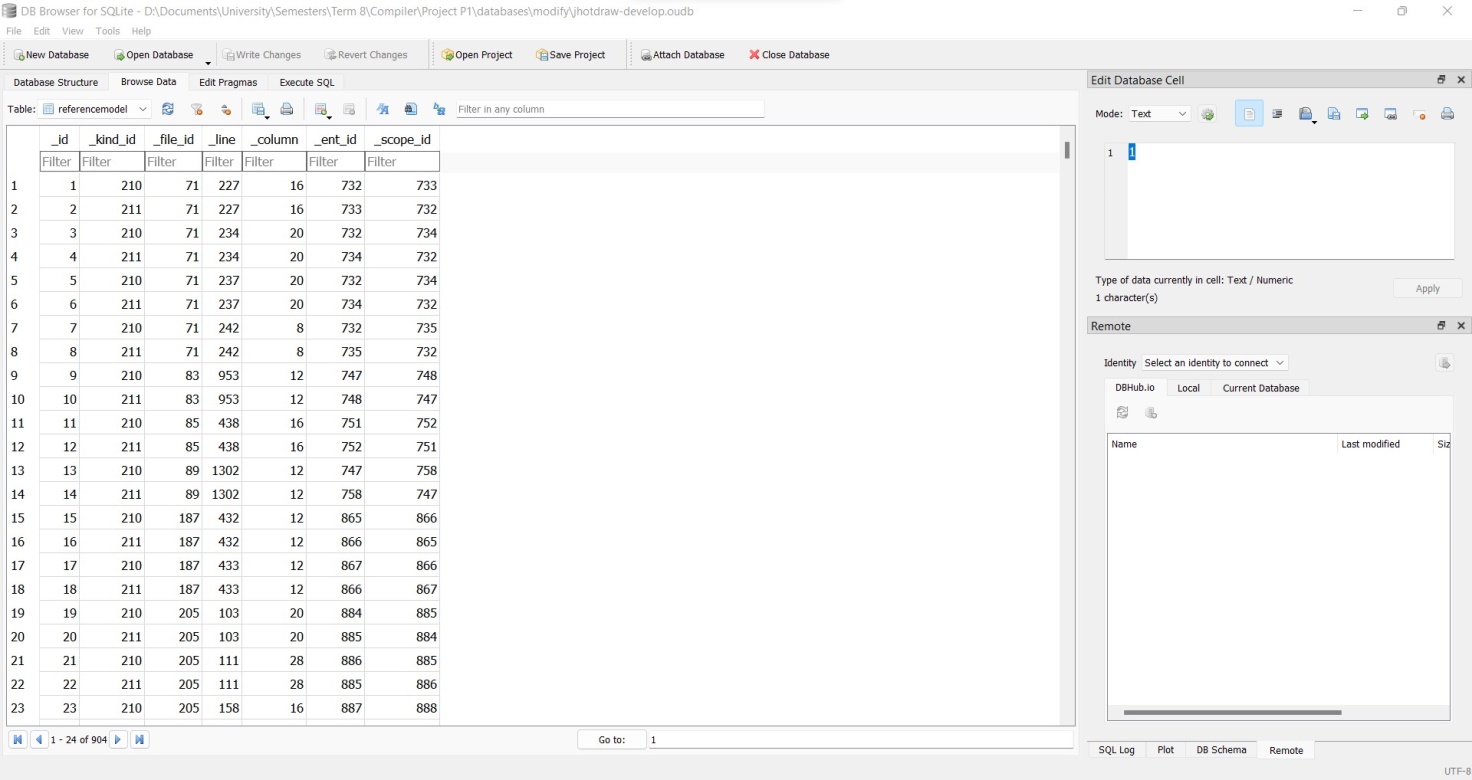


Figure 3.3 Database reference table

**Modify deref(JSON example)**

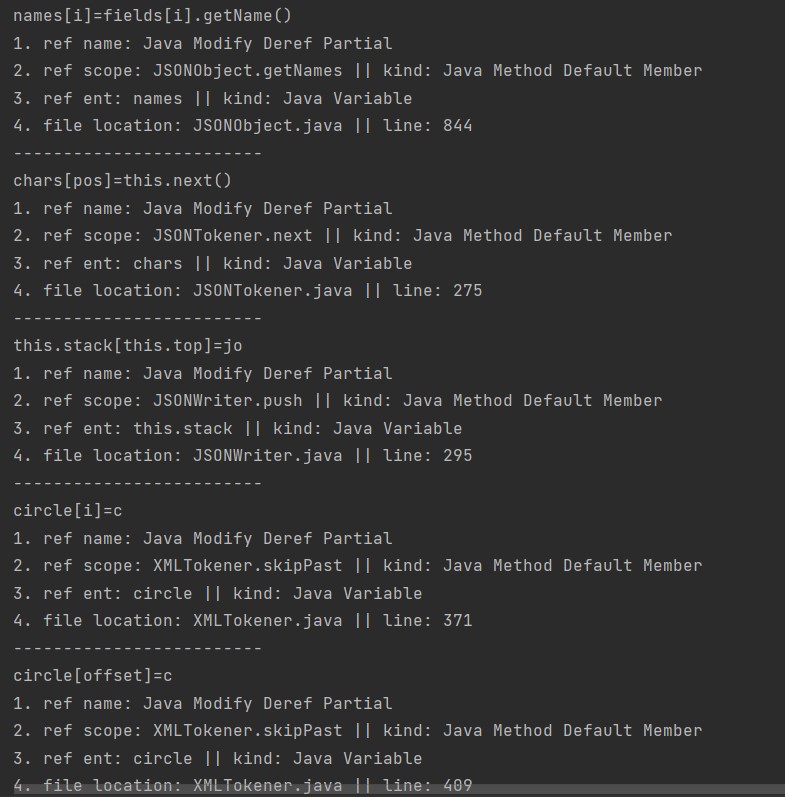


Figure 4.1 CMD results

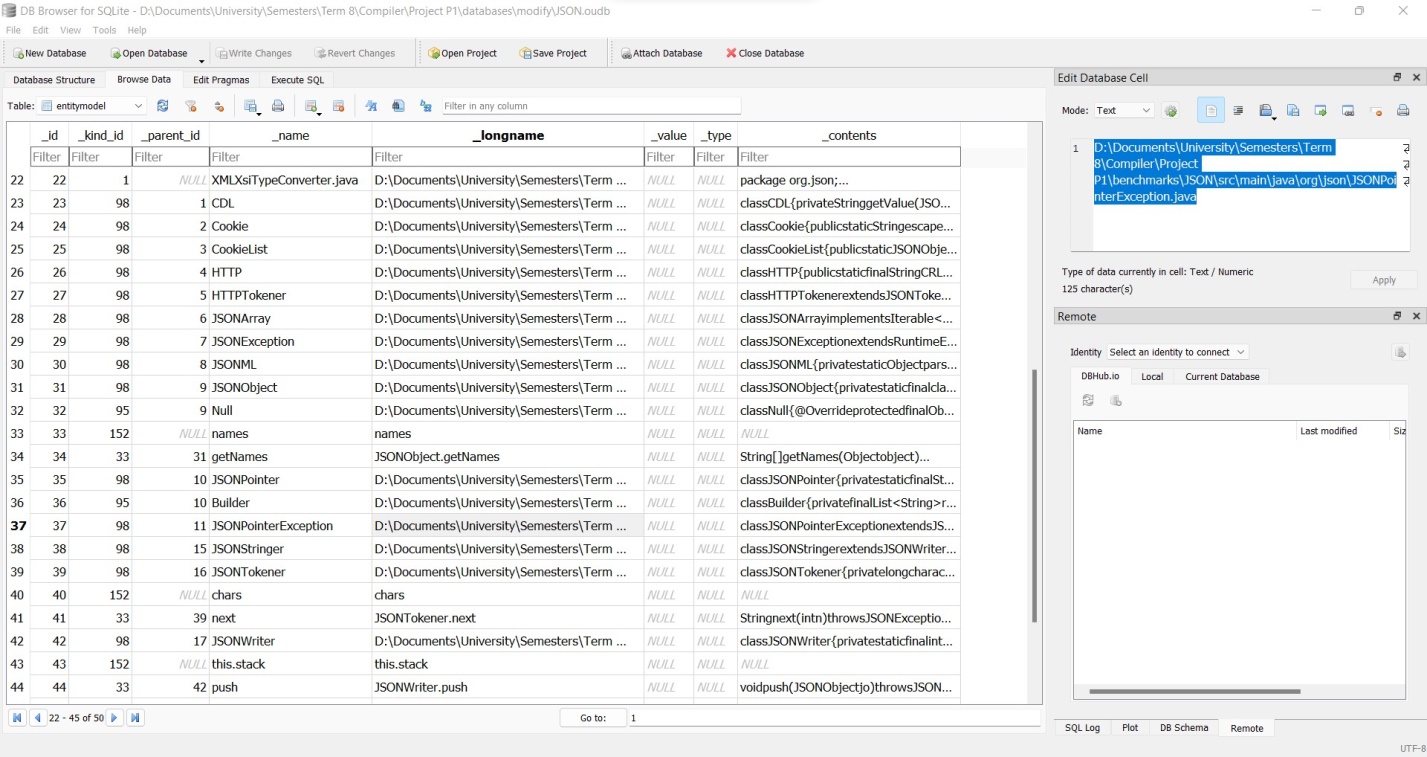


Figure 4.2 Database entity table

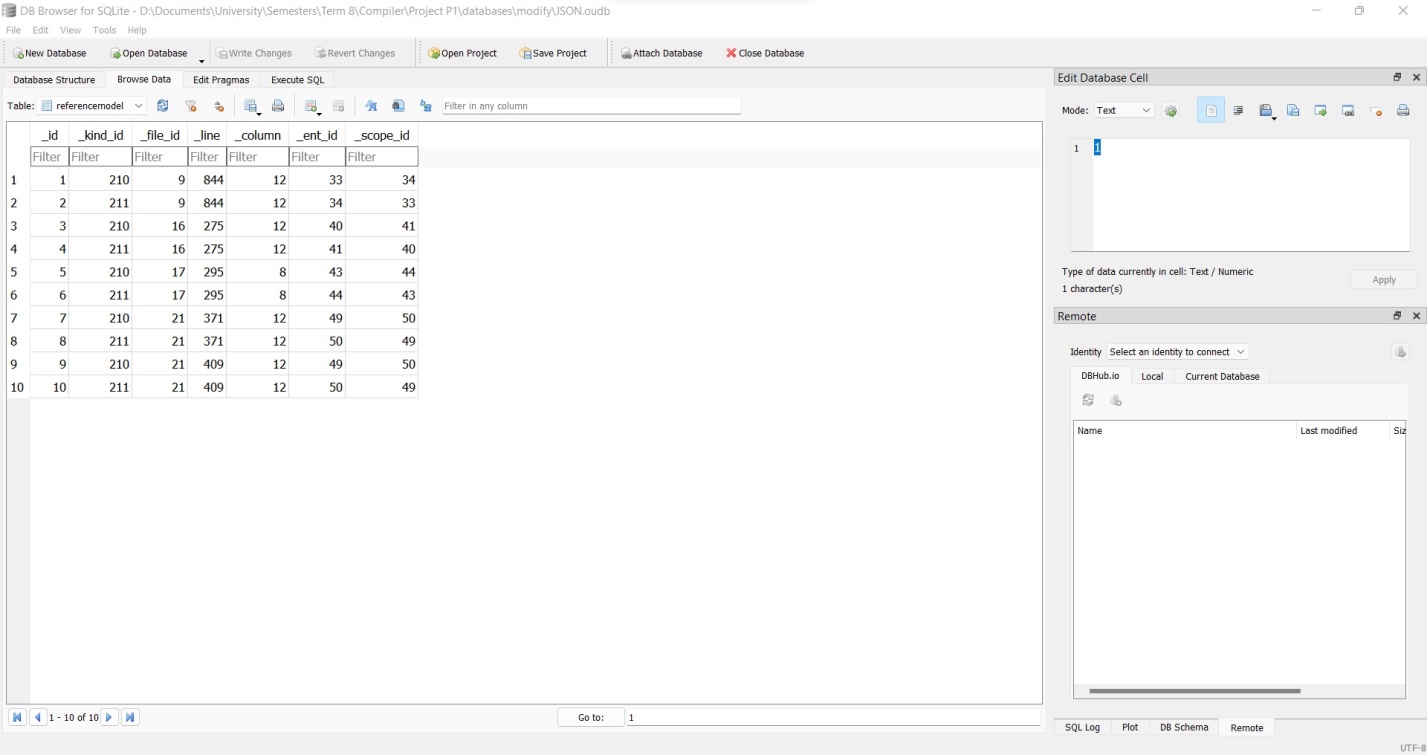


Figure 4.3 Database reference table

.

# **Procedure and Challenges**

As we all know in every project there are many challenges and problems waiting for us. The first step to every project is always one of the hardest ones since it is like entering a whole new world. As for this project our team tried to read and search about the purpose of this project, so the first step was to find out what Understand is and what it does as a tool.

After searching different sources and reading some parts of this tool’s user manual; we figured out that going through your codes for a simple purpose or error is very time and energy consuming, while this tool simply allows you to see most of the aspects your code’s structure has.

After figuring out the necessity of this project and attending to the classes with additional notes about the project structure, we had a 5 days period to learn more about the structure of the code and our tasks, by reading the documentation of this project. Then we installed and tried working with different tools needed to complete this project such as DB browser for SQLite and understand itself.

After all the setup was done we tried to study the codes in the GitHub repository and detect the main problems and issues. The next step was how to divide the work between us, although most of the time the team worked together but some of the parts were done individually. The challenging part was how to handle different entity kinds and add all the necessary data to our database tables correctly. Apart from that issue considering all of the possible options was confusing, hard and time consuming. But we tried to get help from the documentation and the type of data that we could see in the parse tree and Understand’s supported types.

The implementation was done but we still needed to test it, we tried running one of the test cases and checking the tables, which seemed like a success. We also compared it to the results that understand would give us. But as we tried different benchmarks the execution speed was slow and ended up troubling us.

The last part was to write a documentation that indicates all the work and effort done during this time. We tried to write this documentation as simple and clear as possible, that is to explain every part of our code. But because of the connections the codes have it was hard to maintain a fine flow for the reader’s mind and we hope that we succussed doing so.

# **Conclusions and Recommendations**

After finishing this project we think Understand is a useful tool and having an open source Api can help a lot to customize and analyze your code. Now that we can spot out each import we can take control of our sources, inheritance and structure of our code, therefore we can prevent spaghetti code from happening and get one step closer to having a clean code. It goes the same for modify, detecting where our entities and variables are changing can help a lot with debugging and to prevent with changes in wrong scopes for our variables.

For future work this project can be developed to much more completed version of itself for example we can add more entity detection and analyze the code more so that each reference has more attributes. After doing so all of the features have to go through more filters of testing, since the runtime isn’t in the favor of many test the codes should be optimized many times, each time getting better and more efficient.

# **Additional Sources**

<https://www.scitools.com/>

<https://documentation.scitools.com/pdf/understand.pdf>

<https://m-zakeri.github.io/OpenUnderstand/>

<https://en.wikipedia.org/wiki/Understand_(software>)