Compiler project document

Group 8

Group members:

Mohsen akbari

Ali geranmaye

Seyed mohsen eslampanah

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.

At phase two of this project the goal is to implement an extended version of the Sci-tools Understand APIs for computing source metrics. The following

APIs are used to commute source code metrics at different abstraction levels:

- understand.Db.html
- understand.Ent.html

The computation of source code metrics consists of two steps. First, developing the required APIs for querying the database created in Phase one of the project. Second, querying the database using the developed API to find the appropriate entities and reference kinds involved in computing metrics according to the metric definitions. You can find more references and details in this link:

Core source code metrics development

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 throws/throwsby & dotref/dotrefby. 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, this sections was addressed in the phase ne

In the second phase we implemented CountLineCode, CountLinetDecl, CountLineExe and CountLineComment. Source code metrics are essential components in the software measurement process. They are extracted from the source code of the software, and their values allow us to reach conclusions about the quality attributes measured by the metrics.OpenUnderstand supports the following source code metrics for the Java programming language:

Source code metrics

Project division

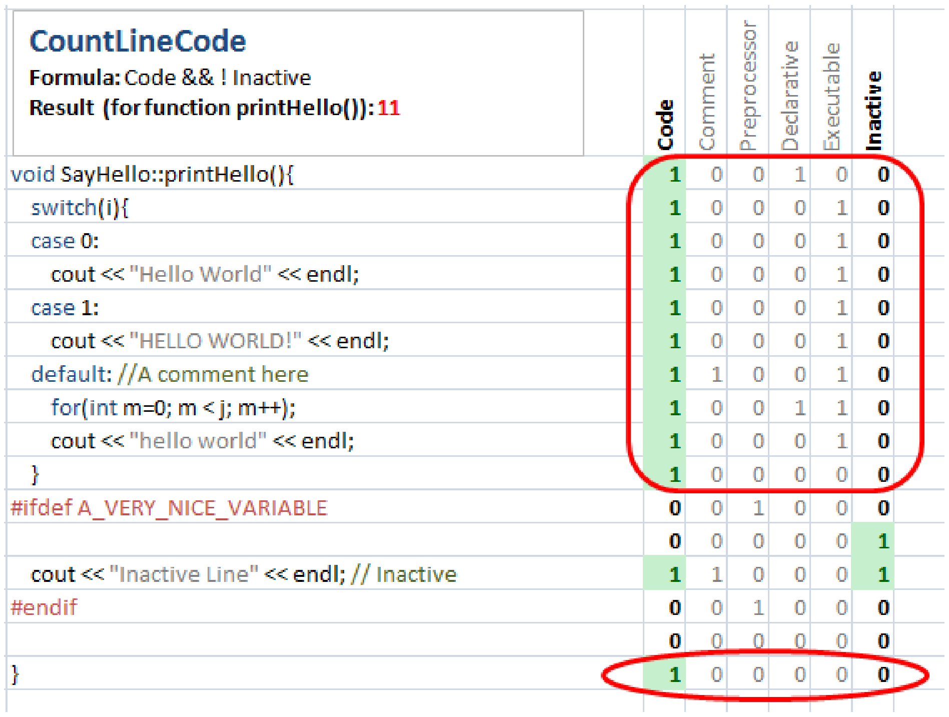
We decided it is best to decide the project to different parts we could each do separately and also a part we could all participate together. Therefor two of us worked on dotref/dotrefby and the two other worked on throws/throwsby. In the end we all gathered to study and test our codes and write this documentation.
Also in the second phase we took the same approach. Therefore, we were divided in two groups of two. Although in this phase a lot more discussing and sharing was needed and the unity was much more tight than the first phase.

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.

CountLineCode:

The number of lines that contain source code. Note that a line can contain source and a comment and thus count towards multiple metrics. For Classes this is the sum of the CountLineCode for the member functions of the class.
For Java we count:
Project, File, Class, Interface, Method



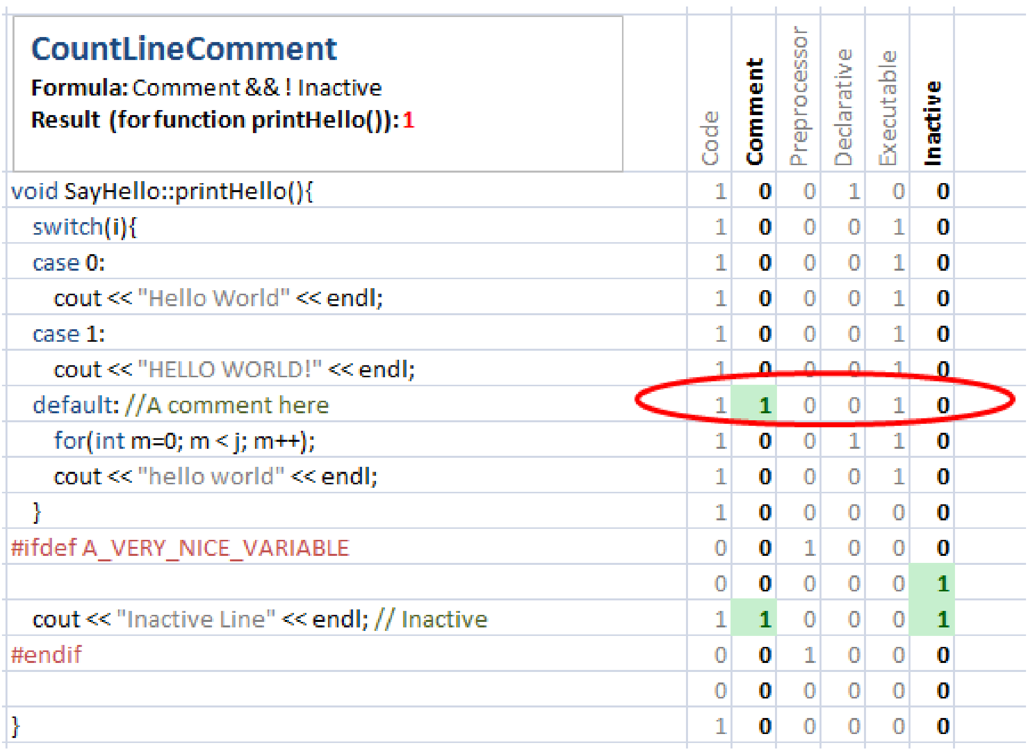
CountLineCode
Formula: Code && ! Inactive
Result (for function printHello()): 11

	Code	Comment	Preprocessor	Declarative	Executable	Inactive
void SayHello::printHello(){	1	0	0	1	0	0
switch(i){	1	0	0	0	1	0
case 0:	1	0	0	0	1	0
cout << "Hello World" << endl;	1	0	0	0	1	0
case 1:	1	0	0	0	1	0
cout << "HELLO WORLD!" << endl;	1	0	0	0	1	0
default: //A comment here	1	1	0	0	1	0
for(int m=0; m < j; m++);	1	0	0	1	1	0
cout << "hello world" << endl;	1	0	0	0	1	0
}	1	0	0	0	0	0
#ifdef A_VERY_NICE_VARIABLE	0	0	1	0	0	0
	0	0	0	0	0	1
cout << "Inactive Line" << endl; // Inactive	1	1	0	0	0	1
#endif	0	0	1	0	0	0
	0	0	0	0	0	0
}	1	0	0	0	0	0

CountLineComment:

Number of lines containing comment. This can overlap with other code counting metrics. For instance int j = 1; // comment has a comment, is a

source line, is an executable source line, and a declarative source line.For Java we count:
Project, File, Class, Interface, Method

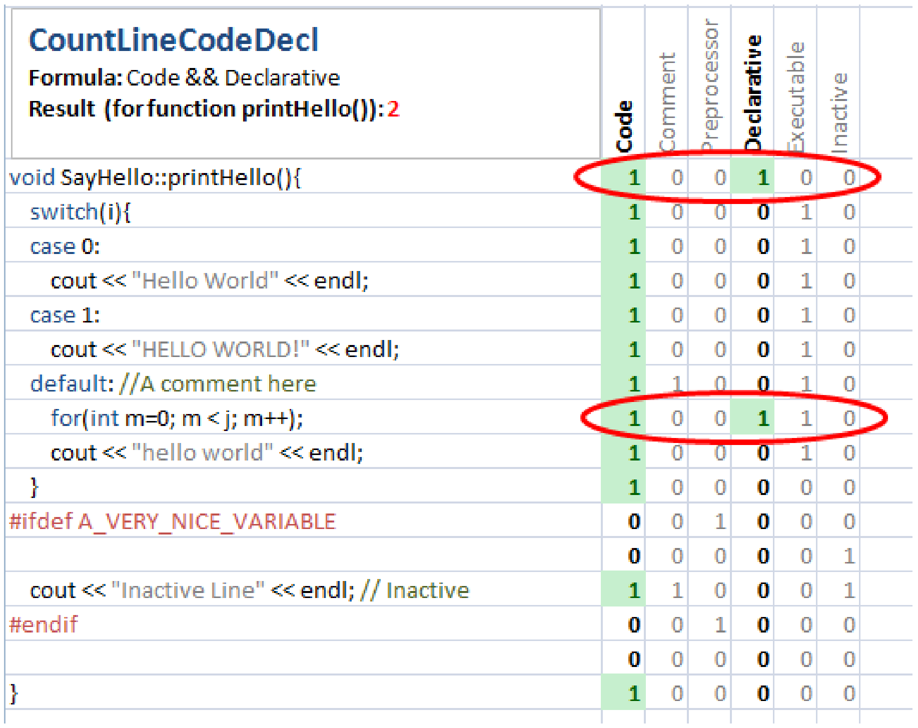


CountLineComment
Formula: Comment && ! Inactive
Result (for function printHello()): 1

	Code	**Comment**	Preprocessor	Declarative	Executable	**Inactive**
void SayHello::printHello(){	1	**0**	0	1	0	**0**
switch(i){	1	**0**	0	0	1	**0**
case 0:	1	**0**	0	0	1	**0**
cout << "Hello World" << endl;	1	**0**	0	0	1	**0**
case 1:	1	**0**	0	0	1	**0**
cout << "HELLO WORLD!" << endl;	1	**0**	0	0	1	**0**
default: //A comment here	1	**1**	0	0	1	**0**
for(int m=0; m < j; m++);	1	**0**	0	1	1	**0**
cout << "hello world" << endl;	1	**0**	0	0	1	**0**
}	1	**0**	0	0	0	**0**
#ifdef A_VERY_NICE_VARIABLE	0	**0**	1	0	0	**0**
	0	**0**	0	0	0	**1**
cout << "Inactive Line" << endl; // Inactive	1	**1**	0	0	0	**1**
#endif	0	**0**	1	0	0	**0**
	0	**0**	0	0	0	**0**
}	1	**0**	0	0	0	**0**

CountLineDecl:

Number of lines containing declarative source code. Note that a line can be declarative and executable - int i =0; for instance.
For Java we count:
Project, File, Class, Interface, Method

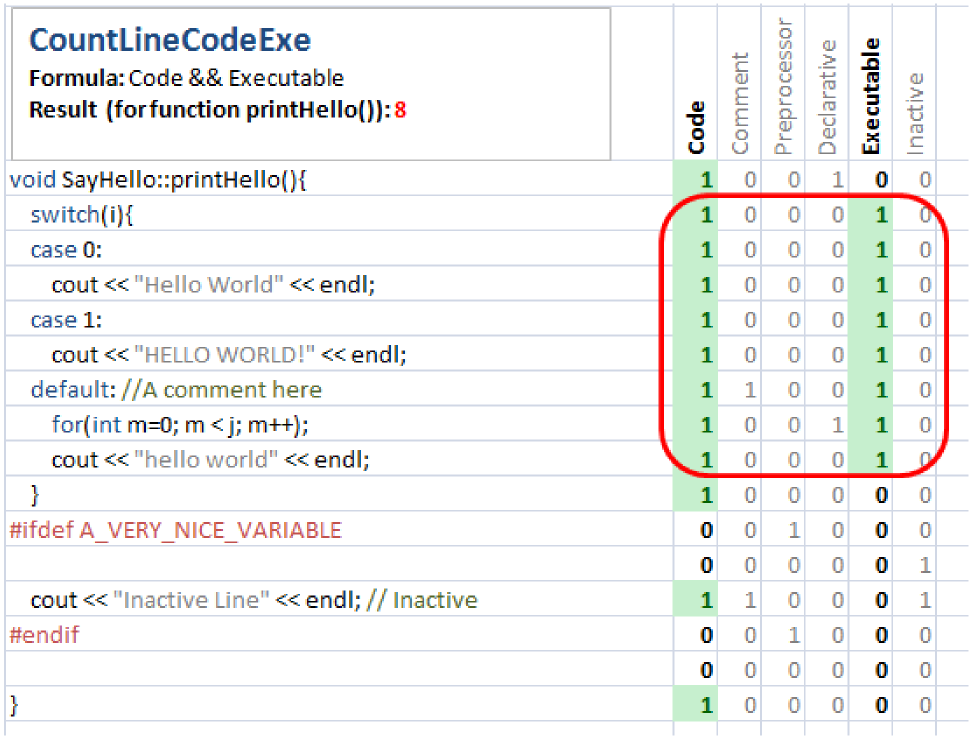


CountLineCodeDecl
Formula: Code && Declarative
Result (for function printHello()): 2

	Code	Comment	Preprocessor	Declarative	Executable	Inactive
void SayHello::printHello(){	1	0	0	1	0	0
switch(i){	1	0	0	0	1	0
case 0:	1	0	0	0	1	0
cout << "Hello World" << endl;	1	0	0	0	1	0
case 1:	1	0	0	0	1	0
cout << "HELLO WORLD!" << endl;	1	0	0	0	1	0
default: //A comment here	1	1	0	0	1	0
for(int m=0; m < j; m++);	1	0	0	1	1	0
cout << "hello world" << endl;	1	0	0	0	1	0
}	1	0	0	0	0	0
#ifdef A_VERY_NICE_VARIABLE	0	0	1	0	0	0
	0	0	0	0	0	1
cout << "Inactive Line" << endl; // Inactive	1	1	0	0	0	1
#endif	0	0	1	0	0	0
	0	0	0	0	0	0
}	1	0	0	0	0	0

CountLineExe:

Number of lines containing executable source code.
For Java we count:
Project, File, Class, Interface, Method



CountLineCodeExe
Formula: Code && Executable
Result (for function printHello()): 8

	Code	Comment	Preprocessor	Declarative	Executable	Inactive
void SayHello::printHello(){	1	0	0	1	0	0
switch(i){	1	0	0	0	1	0
case 0:	1	0	0	0	1	0
cout << "Hello World" << endl;	1	0	0	0	1	0
case 1:	1	0	0	0	1	0
cout << "HELLO WORLD!" << endl;	1	0	0	0	1	0
default: //A comment here	1	1	0	0	1	0
for(int m=0; m < j; m++);	1	0	0	1	1	0
cout << "hello world" << endl;	1	0	0	0	1	0
}	1	0	0	0	0	0
#ifdef A_VERY_NICE_VARIABLE	0	0	1	0	0	0
	0	0	0	0	0	1
cout << "Inactive Line" << endl; // Inactive	1	1	0	0	0	1
#endif	0	0	1	0	0	0
	0	0	0	0	0	0
}	1	0	0	0	0	0

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()
```

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 excel and printed in the terminal.

```
try:
    # metrics lot
    listener = LineOfCode(file_address=file_address)
    p.Walk(listener, tree)
    print(f'CuntLineCode={listener.get_countLineCode}')
    print(f'CuntLineComment={listener.get_countLineComment}')
    print(f'CuntLineExe={listener.get_countLineExec}')
    print(f'CuntLineDecl={listener.get_countLineDecl}')
    df_clc=pd.DataFrame(data=stringify(listener.get_countLineCode),index=[1])

df_clcomment=pd.DataFrame(data=stringify(listener.get_countLineComment),index
=[1])
    df_cld=pd.DataFrame(data=stringify(listener.get_countLineDecl),index=[1])
    df_cle=pd.DataFrame(data=stringify(listener.get_countLineExec),index=[1])
    writer = pd.ExcelWriter(
        r'C:\Users\Lenovo\Desktop\compiler\compiler-project-
dev\openunderstand\countlinecode.xlsx')
    df_clc.to_excel(writer,'countLineCode',index=False)
    df_clcomment.to_excel(writer,'countLineComnnet',index=False)
    df_cld.to_excel(writer,'countLineDecl',index=False)
    df_cle.to_excel(writer,'countLineExe',index=False)
    writer.save()
except Exception as e:
    # print("An Error occurred for reference declare in file:" + file_address
+ "\n" + str(e))
    pass
```

Counting the number of Code lines

For this section we have to count the number of source code lines. For this we just remove the blank lines and count the number of </n> tokens

CountLineCode:

this function returns a tuple (all lines,all tokens) between the start and stop tokens witch is obtained from antlr listener ctx.start and ctx.stop

```
def find(self,start,stop):
    all_lines = []
    singl_line=[]
    all_tokens=[]
    flag = False
    line=start.line
    for i in self.tokens:
        if i.line == start.line and i.column == start.column:
            flag=True
        if flag:
            all_tokens.append(i)
            if i.line==line and i.channel==0:
                singl_line.append(i)
                #print((i))
            elif i.line>line and i.channel==0:
                line+=i.line-line
                #print(i)
                all_lines.append(singl_line)
                singl_line=[]
                singl_line.append(i)
        if i.line == stop.line and i.column == stop.column:
            all_lines.append(singl_line)
            flag=False
    return all_lines, all_tokens
```

CountLineComment:
this function returns the number of comment lines using token.type attribute witch either can be <//> that counts as a single line comment or </ * */> that the number of lines in it can be found by counting </n> in the text attribute of the token.

```
def countLineCodeComment(self,tokens):
    counter=0
    for i in tokens:
        if i.type == 110 and i.channel==1:
            counter+=1
        elif i.type==109 and i.channel==1:
            str=i.text.split()
            counter+=len(str)
    return counter
```

CountLineDecl:
this function returns the number of Declaration lines using token.type attribute witch can be all declaration keywords including <int> <boolean> <double> <float> <new> <short> <long>

```
def countLineCodeDecl(self,tokens):
    counter=0
    for j in tokens:
        for i in j:
            if (i.type in [31,9,48,27,14,8,5,28,29,20,3,37]) and
i.channel==0:
                #print(i.type)
                counter+=1
                break

    return counter
```

CountLineExe:
this function returns the number of Executable lines using token.type attribute In two steps first we discard all the punctuation tokens if they are the only token within a ,line second we use antlr listener enter the right production rules to count the executable lines

```
def countLineCodeExec(self,tokens):
    counter=0
    for j in tokens:
        for i in j:
            if i.channel==0:
                if i.type in [64,67,63,62,61,68]:
                    pass
                else:
                    counter+=1
                    break
    return counter
```

Listener:
this is the main part of the program, first we init the listener so we can have lists that represent the result of our metrics respected to their hierarchies than we start entering the right statements for executable codes we should ignore the signature part of methods and classes so we enter their body not their declarations than we use the self.find(ctx.start,ctx.stop) to find every unit tokens and their lines than we pass the tokens[0] witch is the tokens in every lines to the self.countLineDecl() and self.countLineExe() to find the decl lines and exec lines ins them and we pass the tokens[1] witch is the all

tokens in the unit to self.countLineComment() to find the comment lines and we can use len(tokens[0]) to find the number of source code lines in every unit

```
class LineOfCode(JavaParserLabeledListener):

    def __init__(self,file_address):
        self.file_address = file_address
        file_stream = FileStream(self.file_address)
        self.tokens= JavaLexer(file_stream).getAllTokens()
        self.method_countLineExec = []
        self.method_countLineCode = []
        self.method_countLineComment = []
        self.method_countLineDecl = []
        self.class_countLineDecl = {}
        self.class_countLineExec = {}
        self.class_countLineCode = {}
        self.class_countLineComment = {}
        self.interface_countLineDecl = {}
        self.interface_countLineComment = {}
        self.interface_countLineCode = {}


    @property
    def get_countLineDecl(self):
        return {**self.class_countLineDecl,**self.interface_countLineDecl}

    @property
    def get_countLineCode(self):
        return {**self.class_countLineCode,**self.interface_countLineCode}

    @property
    def get_countLineComment(self):
        return
{**self.class_countLineComment,**self.interface_countLineComment}

    @property
    def get_countLineExec(self):
        return self.class_countLineExec

    def enterCompilationUnit(self,
ctx:JavaParserLabeled.CompilationUnitContext):
        tokens=self.find(ctx.start,ctx.stop)
        self.class_countLineCode['file']=len(tokens[0])

self.class_countLineComment['file']=self.countLineCodeComment(tokens[1])
    def enterClassDeclaration(self,
ctx:JavaParserLabeled.ClassDeclarationContext):
        tokens=self.find(ctx.start,ctx.stop)
        self.method_countLineDecl.append(('class
'+ctx.IDENTIFIER().getText(),self.countLineCodeDecl(tokens[0])))
        self.method_countLineExec.append(('class
'+ctx.IDENTIFIER().getText(), self.countLineCodeExec(tokens[0])))
        self.method_countLineComment.append(('class
'+ctx.IDENTIFIER().getText(), self.countLineCodeComment(tokens[1])))
        self.method_countLineCode.append(('class
'+ctx.IDENTIFIER().getText(), len(tokens[0])))

    def enterMethodDeclaration(self,
ctx:JavaParserLabeled.MethodDeclarationContext):
        tokens=self.find(ctx.start,ctx.stop)
        self.method_countLineDecl.append(('method
'+ctx.IDENTIFIER().getText(), self.countLineCodeDecl(tokens[0])))
        self.method_countLineComment.append(('method
'+ctx.IDENTIFIER().getText(), self.countLineCodeComment(tokens[1])))
        self.method_countLineCode.append(('method
'+ctx.IDENTIFIER().getText(),len(tokens[0])))

    def enterMethodBody(self, ctx:JavaParserLabeled.MethodBodyContext):
        tokens=self.find(ctx.start,ctx.stop)
        self.method_countLineExec.append(('method
'+ctx.parentCtx.IDENTIFIER().getText(),self.countLineCodeExec(tokens[0])))

    def exitClassDeclaration(self,
ctx:JavaParserLabeled.ClassDeclarationContext):
        self.class_countLineExec[ctx.IDENTIFIER().getText()] =
self.method_countLineExec
        self.method_countLineExec=[]
        self.class_countLineDecl[ctx.IDENTIFIER().getText()] =
self.method_countLineDecl
        self.method_countLineDecl=[]
        self.class_countLineComment[ctx.IDENTIFIER().getText()] =
self.method_countLineComment
        self.method_countLineComment=[]
        self.class_countLineCode[ctx.IDENTIFIER().getText()] =
self.method_countLineCode
        self.method_countLineCode=[]
        #print(self.class_countLineDecl)

    def enterInterfaceDeclaration(self,
ctx:JavaParserLabeled.InterfaceDeclarationContext):
        tokens=self.find(ctx.start,ctx.stop)
        self.interface_countLineDecl['interface '+ctx.IDENTIFIER().getText()]
= self.countLineCodeDecl(tokens[0])
        self.interface_countLineComment['interface
'+ctx.IDENTIFIER().getText()] = self.countLineCodeComment(tokens[1])
        self.interface_countLineCode['interface '+ctx.IDENTIFIER().getText()]
= len(tokens[0])
        #print((self.interface_countLineDecl))
```

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).

The resulted tables are as follows:

Import(test.java example)

first we should explain the criteria witch open understand calculate its metric are somewhat deferent from what we have calculated our metrics upon for example in this simple test file

```
public interface FooFace {   // +0
  void doFoo();              // +0
}

public class Foo1 implements FooFace {   // +0
  private char name;                   // +0
  /* sada
  adsd
  dfsd
  aw
  rer
   asda */
}
public class Foo2 implements FooFace {   // +0
  private char name;
  public int fooo(int a)
  {
    int b=a;

  }

}
```

upon for example in this simple test file open understand counts 5 countLineComment but with our standards we should count 11
in other parts we have the same logic
so the results would be

- **Test.ajva**

	CLCode	CLComment	CLDecl	CLExe
Understand	13	5	8	1
Our Work	13	11	8	1

Second thing is open understand calls the metrics on the package as well

For example in AddDictObjectAction.java file in the jvlt 1.3.2 we have a package called actions

```
package net.sourceforge.jvlt.actions;

import net.sourceforge.jvlt.core.Reinitializable;

public class AddDictObjectAction extends DictObjectAction {
   public AddDictObjectAction(Reinitializable obj) {
      super(obj);
   }
}
```

the metrics of the class

- **Jvlt-1.3.2/AddDictObjActions.java**

	CLCode	CLComment	CLDecl	CLExe
Understand	7	0	4	1
Our Work	7	0	4	1

But the metric of the package is included in the file

- **Jvlt-1.3.2/actions.package**

	CLCode	CLComment	CLDecl	CLExe
Understand	436	2	181	186
Our Work	0	0	0	0

So this results in a major difference between the open understands CountLineDecl , CountLineExe and our results.
In the document of the project we were told to find the result of the metrics in this hierarchy:
Project, File, Class, Interface, Method
So we did what we were told and didn’t have information on the package.

The end results for the Jvlt projects:

- **Jvlt-1.3.2**

	CLCode	CLComment	CLDecl	CLExe
Understand	23807	1179	8953	13676
Our Work	23087	4226	4951	11706

- **Calculator_app**

The calculator_app didn't have any packages so the results of open understand and our metrics were close:

	CLCode	CLComment	CLDecl	CLExe
Understand	126	20	58	46
Our Work	126	20	62	43

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.
In the second phase one of the hardest parts were the details we had missed when we tested the benchmark projects and compared the results. The main code base was kind of easy to develop; but after running the benchmark projects we had to refactor our code to get the result closer and closer to understand’s results. Especially in the bigger projects the number that understand would detect was far more bigger than our results. This refactoring and fixing the details took a lot of effort, energy and time. For example in the cyclomatic task understand would count 2 for enums(value, value of) but we thought it is supposed to be only for methods. Or for example it didn’t count a cyclomatic for abstract methods, but it did for the methods of an interface class. Since there wasn’t any clear source to detect these, we had to go through all of the logs and a lot of comparing had to be done. Therefore we spent and awful lot of time on these details to get the closest results possible.
In the count statement task the main problem was that it had a lot of entities that had to be considered, so the details were even more complex.

Apart for that it would give us the sum of all the statement counts. For example it would give us the sum of statement counts in each of the methods of a class.

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)

Phase two specific:

Source code metrics

Core source code metrics development

https://www.gatevidyalay.com/compiler-design

https://www.gatevidyalay.com/first-and-follow-compiler-design

https://docs.sonarqube.org/latest/extend/executable-lines/#:~:text=Executable%20lines%20data%20is%20used,what%20coverage%20engines%20would%20calculate.