

ECOTE - Final Project

Date: 09.06.2021

Semester: 2021L

Author and Group: Caner Kaya – Group 104

Subject: Task 8. CD3

I. General overview and assumptions

The project shows on the graph the data member's dependency between classes of the Java code. The definition of dependency between classes is If a class uses an member of the another class. A library will not be used to parse java code, simple java parser will be implemented from scratch using regular expressions

Assumptions

- The Java code given as an input is written and interpreted without error.
- There are no comment lines in the code.
- The definitions of the all classes are made in the given file.
- Variable names can be used multiple times in different scopes, but the class names must be unique.
- Other than basic data structures, data structures under java.util should not be used as members of classes.
- The fundamental classes of Java (e.g String, Exception...) or classes imported from a package will not be shown in dependencies.
- The data member's usage with any method will not be considered as a data member's dependency.
- The "this" keyword should be used to avoid confusion when accessing a public member, even if there is no local member with the same name.
- Since interfaces do not have a body in Java, no dependencies were sought on them.
- If an inner class is dependent, outer classes are not included in the dependency list. (designed this way to better show where the dependency comes from)

II. Functional requirements

- The program should recognize names, fields and scopes of the defined classes.
- The program should determine that a data member of another class is used in a class.

- The program should construct a graph of the dependencies it determines.

Syntax

Java Class Definition

modifier	type	class	ClassName	extends	ClassName	{....}
public	abstract					
private	static	<i>class keyword</i>	<i>a valid string</i>	<i>If class have inheritance</i>	<i>a valid String</i>	<i>class scope</i>
protected	final	<i>must be written</i>				
internal	∅					
∅						

Java Class Definition

Example Java Class Definitions:

```
public abstract class Car extends Vehicle { ... }
protected final class Building { ... }
private class Teacher {...}
class Student{ ... }
```

Each keyword is separated from the next by a space character. There are 6 keywords can be used to define a class. *Access modifier* specifies permission to access the class. If it is not defined on the code, it is treated as private by the compiler. To define a different type of class, the *type* keywords are used. If the type is not specified, it refers to the regular java class. It is necessary to write the “class” keyword while defining the class. Then a class name is given to the class. It is recommended that class names begin with a capital letter, but it is not required. If the class inherits from another class, the definition continues with the “extends” keyword. After that, the name of the inherited class is included.

After these definitions, the square brackets are opened and the code segments until the square brackets are closed indicate the class scope.

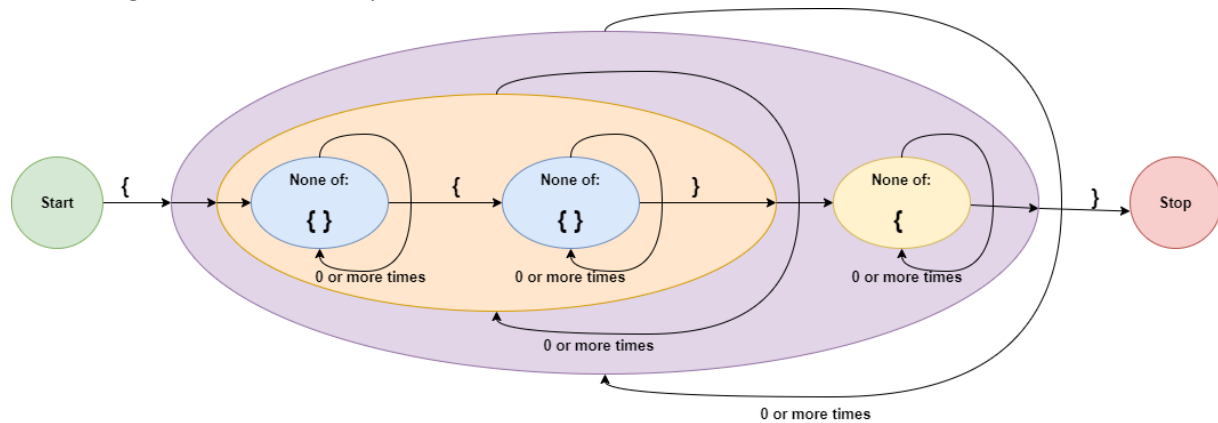
Regular Expression to detect the Java Class definition:

```
(private|public|protected|internal)?\s+(abstract|final|static)?\s+class\s*(\w+)\s*[\^{]*{
```

Java Class Scope

After determining the class definition, it is necessary to determine the class scope. Curly brackets are used at the start and end of the class's scope, but we can also see the opening and closing brackets inside of the class scope, indicating local scopes like inner class, functions, expressions etc. Against this, it is expected that the number of brackets opened is equal to the number of closed brackets and the brackets are in the correct order (first opening and then closing).

In the status chart below, It is shown how to determine where the end of class scope. The purple group after the open braces represents the body of the class. The body of the class may be empty. The orange group in the class's body represents the inner scopes in the class. A class may have more than one inner scope or none at al. The yellow group after the orange field represents the global definitions and operations within the class. We understand that the body of the class is ended with the closing braces and the scope of the class ends.



The regular expression obtained according to the figure is as follows:

`{([{}]*{[{}]*})*[{}]*}`

Java Global/Local Scope

```
outer
├── inner
│   └── innerMost
```

Another issue that should be considered before examining the definitions and assignments within classes is the local and global scopes. Inner scopes can access and modify data from outer scopes. Therefore, it is necessary to search on the tree from bottom to top in order to determine on which variable an operator is performed. Creating the class tree will be discussed in the implementation section.

Java Class Member Definition

Class members can be defined anywhere in the scope of the class in java. Class members also known as attributes. We can talk about 3 types of attributes definition.

Example 1:

```
public String name;           //defining a member
```

Example 2:

```
private String name = new String("Jakub"); //defining a member and dynamically assigning an object
```

Example 3:

```
protected String name = aName; //defining a member and assigning a reference of an object
```

access modifier	variableType	variableName	=	new	variableType	()	;
private							
public	basic data	A valid string	not concerned with this assignment part				end of
Protected	type or a						definition
Ø	class						

We are not concerned with this assignment part, as we know the code is error free. Since there is no type mismatch error in the code, it will be sufficient to learn the variableType that comes before the variableName to find out the type of the variable.

We can write the regular expression below by looking at the table above. With this expression, we can detect the attributes defined in the class and the details of the attributes.

Regular Expression: `(private|public|protected)?\s*(\w+)\s*(\w+)\w*;`

Java Data Member Usage

In order to determine the data member dependency of the classes, we have to follow the objects belonging to other classes which is defined within the classes or taken as parameters. In order to determine the object definitions, we must have a list of classes. The method of obtaining class names is shown in the previous steps. When the data member belonging to one of these objects is accessed, we detect the dependency. Java syntax examples of data member usage are shown below.

Example 1: Object defined as attribute can be accessed through method.

```
class aClass {
    public Book book;
    ...
    void readBook(){
        int pageNumber = this.book.pageNumber;
    }
    ...
}
```

Example 2: Dynamically defined object can be accessed in same scope

```
class aClass {
    ...
```

```

    void readBook(){
        Book book = new Book();
        book.pageNumber = 250;
    }
}

```

Example 3: An object taken as a parameter can be accessed

```

class aClass {
    public Book book;
    void readBook(Notebook book){
        int pageNumber = book.pageNumber; // it is an Notebook object
    }
}

```

Example 4: The object of a outer level class can be accessed

```

class aClass {
    public Book book;
    class innerClass{
        void readBook(){
            int pageNumber = book.pageNumber; // If there is no such member
            //of the inner class, the outer class is checked
        }
    }
}

```

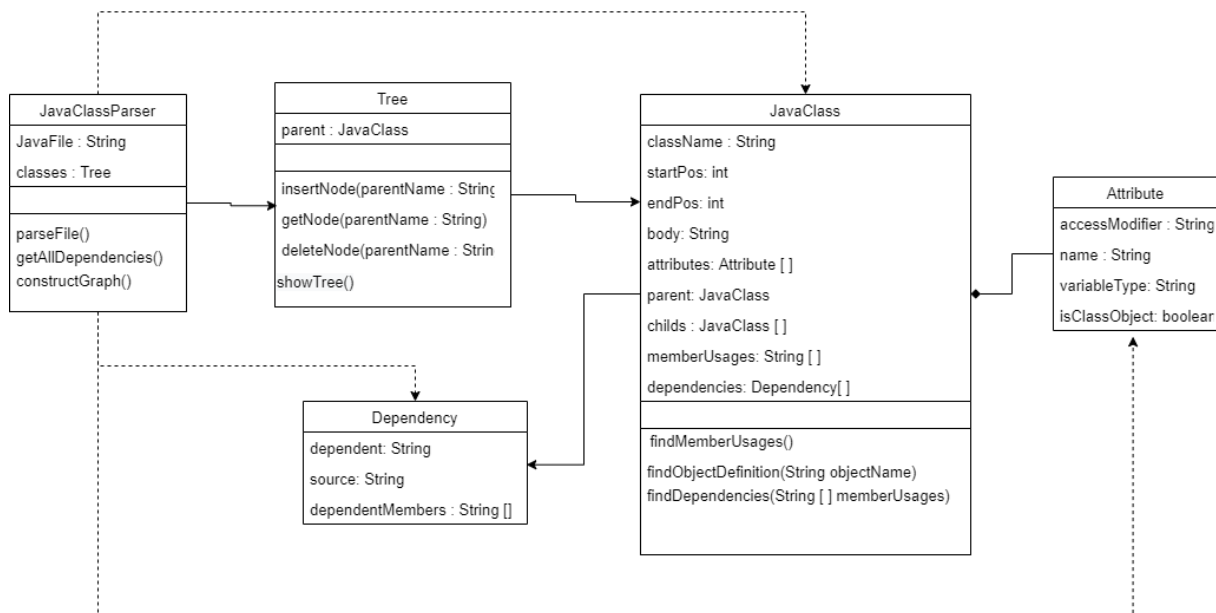
Regular Expression for member usages : `(?: (\w+) \.)? (\w+) \. (\w+)`

The above regular expression can also detect the method usage, we should clear them with regular expression below from the results.

Regular Expression for method usages: `(?: (\w+) \.)? (\w+) \. (\w+) \.`

III. Implementation

General architecture



The UML diagram of the project is shown above. Only important methods are included in UML. **JavaClassParser** class parses the given file, detects classes and dependencies and keeps them in the tree structure. And after that constructs dependency graph.

Data structures

Since there may be nested classes in the file, the tree structure is used instead of keeping the class as a list. The nodes of the tree consist of a **JavaClass** class. There is a dummy class named **ROOT** in the root node of the tree. All outer classes defined in the file are available as child of this **ROOT** class. If a class contains an inner class(es), these are also added to the tree as the child of the class.

In the example below, we see that the classes of **Person** and **Vehicle** are defined as outer classes, two classes named **Car** and **Truck** are defined within the **Vehicle** class, and a class named **Cargo** is also defined in the **Truck** class.

```

ROOT
├── Person
├── Vehicle
│   ├── Car
│   └── Truck
│       └── Cargo

```

Module descriptions

(with algorithms at implementation level)

In this section, the structure of some methods seen in the UML diagram is explained.

Attribute Class: The attribute class is a part of the **JavaClass** class, as seen in UML, it cannot exist by itself. It holds attribute information of the **JavaClass**. The `isClassObject` flag is true if the attribute belongs to a class's object defined in the file. `isClassObject` flag is useful when determining

dependencies between classes. For example, among the attributes accessed with the “this” keyword, only those belonging to objects are of interest.

Dependency Class: It keeps the name of the dependent class and the information on which class it depends and on which data members this dependency is provided.

Java Class: The `JavaClass` class is also a tree node. That's why it has members named `parent` and `child`. While each class can have only one parent, it can have more than one child. Information such as `className`, `startPos`, `endPos`, `body`, `parent`, is sent with the parameter through the constructor when creating the class. This information is provided by regular expressions as explained in the syntax section. Other informations are assigned via methods belonging to the class.

findAttributes(): Using regular expressions the attributes belonging to the class are stored.

findMemberUsages(): Lists members whose `isClassObject` flag is true and the data members of objects that come to the function as parameters or dynamically defined objects are accessed. Within `memberUsages`, the name of the member, its usage and position is kept as a string, separated by commas.

findDependencies(String [] memberUsages): Takes the `memberUsages` list as a parameter and determines which classes the objects in this list belong to.

#CASE 1

IF the member is accessed with “this” keyword

//the object defined as an attribute

THEN The member is the attribute of the class it is in. The dependency is determined by looking at the `variableType` of the attribute.

#CASE 2

ELSE IF the data member is defined in the scope in which it is located

//dynamically defined object

THEN The dependency is determined by looking at the variable Type where is defined.

#CASE 3

ELSE IF the data member is a method parameter in the scope it is in

// object taken as a parameter of a method

THEN The dependency is determined by looking at the variable type of the argument

#CASE 4

ELSE

// object is an attribute defined in outer class.

THEN The parent class of the current class is checked and if the name of the object is in the attribute list of this class, the variable type of the object is found. Otherwise, a parent is checked until it is found.

Input/output description

The implemented program is an Python console application. Program takes a code file with java extension as input. Constructs a graph showing the dependencies of the classes with .png extension as output. The text version of the graph and structured information about Intermediate steps are optionally output as a text file.

Usage: main.py [-h] --input FILEPATH [--output FILEPATH] [--details BOOLEAN]
 [--labels BOOLEAN] [--pngsize PNGSIZE PNGSIZE]

arguments:

-h, --help	Show this help message and exit
--input FILEPATH	(required) The path of the java file. Extension must be .txt or .java.
--output FILEPATH	The path of the output file(s)
--details BOOLEAN	Creates a text file that consist parsing details by default. If this file is not desired, this flag is used with 'False' parameter .
--labels BOOLEAN	show/hide the labels of the member that caused the dependency on the graphic. This name information, shown by default, can make the graph invisible if dependency number is high.
--pngsize PNGSIZE PNGSIZE	Specifies the inches size of the output graphic [inch][inch]

<u>Arguments</u>	<u>Errors</u>
--input FILEPATH	The Cause : If input is not given Output: usage: main.py [-h] --input FILEPATH [--output FILEPATH] [--details BOOLEAN] [--labels BOOLEAN] [--pngsize PNGSIZE PNGSIZE] main.py: error: the following arguments are required: --input The Cause: If the file is not found Output: The file could not be opened The Cause: If the extension is not .java or .txt Output: file doesn't end with one of ('java', 'txt')
--output FILEPATH	The Cause: If the file could not be created Output: The file could not be created

IV. Functional test cases

TEST 1 :

Input File

```
1. class Contact{
2.     public String address;
3.     public String telephoneNumber;
4.     public String email;
5.     public Contact(String address, String telephoneNumber, String email) {
6.         this.address = address;
7.         this.telephoneNumber = telephoneNumber;
8.         this.email = email;
9.     }
10. }
11.
12. class Person{
13.     public String nameSurname;
14.     public Contact contact;
15.     public Person(String nameSurname, Contact contact) {
16.         super();
17.         this.nameSurname = nameSurname;
18.         this.contact = contact;
19.     }
20.     void changeEmail(String newEmail) {
21.         this.contact.email = newEmail;}
22. }
23.
```

Output txt

Root

├─ Contact
└─ Person

[CLASS] Contact : 1-10

---Attribute Objects---

| None

---Dynamic Objects---

| None

---Parameters---

| String email : 5

--Member Usages---

[]

[CLASS] Person : 12-22

---Attribute Objects---

| Contact contact : 14

---Dynamic Objects---

| None

---Parameters---

| Contact contact : 15

| String newEmail : 20
--Member Usages---
['this.contact.email,21']

-----LOGS-----

CASE 1: Used an Attribute Object

Person(12-22) dependent by Contact(1-10) because this.contact.email,21 is member of Contact Object,14

Dependencies: [[['Person', 'Contact'], 'this.contact.email,21']]

Output graph



TEST 2 :

Input File

```
1. class Engine{
2.     private int power;
3.     public Engine(int power){this.power = power;}
4.     int getPower(){return power;}
5. }
6. class Car{
7.     public Car( Engine engine ){this.engine = engine;}
8.     void setEngine(int power){
9.         Engine myEngine = new Engine(power);
10.        System.out.print(myEngine.power);
11.    }
12. }
13.
```

Output txt

Root
├─ Car
└─ Engine

[CLASS] Engine : 1-5

---Attribute Objects---

| None

---Dynamic Objects---

| None

---Parameters---

| int power : 3

--Member Usages---

[]

[CLASS] Car : 6-12

---Attribute Objects---

| None

---Dynamic Objects---

```

| Engine myEngine : 9
---Parameters---
| Engine engine : 7
| int power : 8
| power : 9
--Member Usages--
['myEngine.power,10']

```

-----LOGS-----

CASE 2: Used a Dynamically defined object

Car(6-12) dependent by Engine(1-5) because myEngine.power,10 is member of Engine Object,9

Dependencies: [[['Car', 'Engine'], 'myEngine.power,10']]

Output graph



TEST 3:

Input File

```

1. class Bill{
2.     public float amount;
3.     public Bill(float amount){this.amount = amount; }
4. }
5. class Client{
6.     private float balance;
7.     public Client (float balance){this. balance = balance; }
8.     boolean payBill( Bill bill){
9.         if( balance >= bill.amount){
10.             balance -= bill.amount;
11.             return true;
12.         }
13.         return false;
14.     }
15. }
16.

```

Output txt

```

Root
├─ Bill
└─ Client

```

[CLASS] Bill : 1-4

---Attribute Objects---

| None

---Dynamic Objects---

| None

---Parameters---

| float amount : 3

--Member Usages--

[]

[CLASS] Client : 5-15

---Attribute Objects---

| None

---Dynamic Objects---

| None

---Parameters---

| float balance : 7

| Bill bill : 8

--Member Usages--

['bill.amount,9', 'bill.amount,10']

-----LOGS-----

CASE 3: Used a taken as parameter object

Client(5-15) dependent by Bill(1-4) because bill.amount,9 is member of Bill Object,8

CASE 3: Used a taken as parameter object

Client(5-15) dependent by Bill(1-4) because bill.amount,10 is member of Bill Object,8

Dependencies: [[['Client', 'Bill', 'bill.amount,9'], [['Client', 'Bill', 'bill.amount,10']]]

Output graph



TEST 4:

Input File

```
1. class Pen{
2.     int price;
3.     public Pen(int price) {
4.         this.price = price;
5.     }
6. }
7. class Pencil{
8.     int price;
9.     public Pencil(int price) {
10.        this.price = price;
11.    }
12. }
13.
14. class Person{
15.     protected Pen pencil;
16.
17.     public Person(Pen pencil) {
18.         this.pencil = pencil;
19.     }
20.     class Child{
21.         public Child() {}
```

```

22.             int sayPencilPrice(Pencil pencil){
23.                 return pencil.price;
24.             }
25.         }
26.     }

```

Output txt

Root

```

├─ Pen
├─ Pencil
├─ Person
└─ Child

```

[CLASS] Pen : 1-6

---Attribute Objects---

| None

---Dynamic Objects---

| None

---Parameters---

| int price : 3

--Member Usages---

[]

[CLASS] Pencil : 7-12

---Attribute Objects---

| None

---Dynamic Objects---

| None

---Parameters---

| int price : 9

--Member Usages---

[]

[CLASS] Person : 14-26

---Attribute Objects---

| Pen pencil : 15

---Dynamic Objects---

| None

---Parameters---

| Pen pencil : 17

| Pencil pencil : 22

--Member Usages---

[]

[CLASS] Child : 20-25

---Attribute Objects---

| None

---Dynamic Objects---

| None

---Parameters---

| Pencil pencil : 22

--Member Usages---

['pencil.price,23']

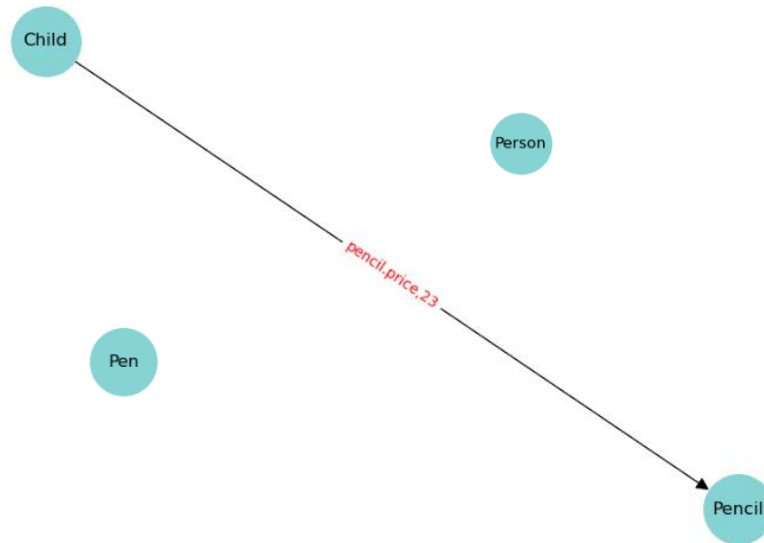
-----LOGS-----

CASE 3: Used a taken as parameter object

Child(20-25) dependent by Pencil(7-12) because pencil.price,23 is member of Pencil Object,22

Dependencies: [[['Child', 'Pencil'], 'pencil.price,23']]

Output graph



TEST 5:

Input File

```
1.
2. class Birthday{
3.     public int year;
4.     public int month;
5.     public int day;
6.     public Birthday(int year, int month, int day) {
7.         this.year = year;
8.         this.month = month;
9.         this.day = day;
10.    }
11. }
12.
13. class Person{
14.     public Birthday bday;
15.     public Person(Birthday bday) {
16.         this.bday = bday;
17.     }
18.     public boolean isAdult() {
19.         if(this.bday.year + 18 > 2021) {
20.             return true;
21.         }
22.         return false;
23.     }
24. }
25.
26.
27. class ChildProfile {
```

```

28.         public String authority;
29.         public ChildProfile() {
30.             authority = "Cartoons";
31.         }
32.     }
33.
34. class AdultProfile{
35.     public String authority;
36.     public AdultProfile() {
37.         authority = "All Content";
38.     }
39. }
40.
41. class User{
42.     Person person;
43.     public User(Person person) {
44.         this.person = person;
45.         boolean isAdult = person.isAdult();
46.
47.         if(isAdult){
48.             AdultProfile aProfile = new AdultProfile();
49.             System.out.println(aProfile.authority);
50.         }else {
51.             ChildProfile cProfile = new ChildProfile();
52.             System.out.println(cProfile.authority);
53.         }
54.     }
55.
56. }
57.

```

Output txt

Root

- ├─ AdultProfile
- ├─ Birthday
- ├─ ChildProfile
- ├─ Person
- └─ User

[CLASS] Birthday : 2-11

---Attribute Objects---

| None

---Dynamic Objects---

| None

---Parameters---

| int day : 6

--Member Usages---

[]

[CLASS] Person : 13-24

---Attribute Objects---

```
| Birthday bday : 14
---Dynamic Objects---
| None
---Parameters---
| Birthday bday : 15
--Member Usages---
['this.bday.year,19']
```

```
-----
[CLASS] ChildProfile : 27-32
---Attribute Objects---
| None
---Dynamic Objects---
| None
---Parameters---
| None
--Member Usages---
[]
```

```
-----
[CLASS] AdultProfile : 34-39
---Attribute Objects---
| None
---Dynamic Objects---
| None
---Parameters---
| None
--Member Usages---
[]
```

```
-----
[CLASS] User : 41-56
---Attribute Objects---
| Person person : 42
---Dynamic Objects---
| AdultProfile aProfile : 48
| ChildProfile cProfile : 51
---Parameters---
| Person person : 43
| isAdult : 47
--Member Usages---
['aProfile.authority,49', 'cProfile.authority,52']
```

-----LOGS-----

CASE 1: Used an Attribute Object

Person(13-24) dependent by Birthday(2-11) because this.bday.year,19 is member of Birthday Object,14

CASE 2: Used a Dynamically defined object

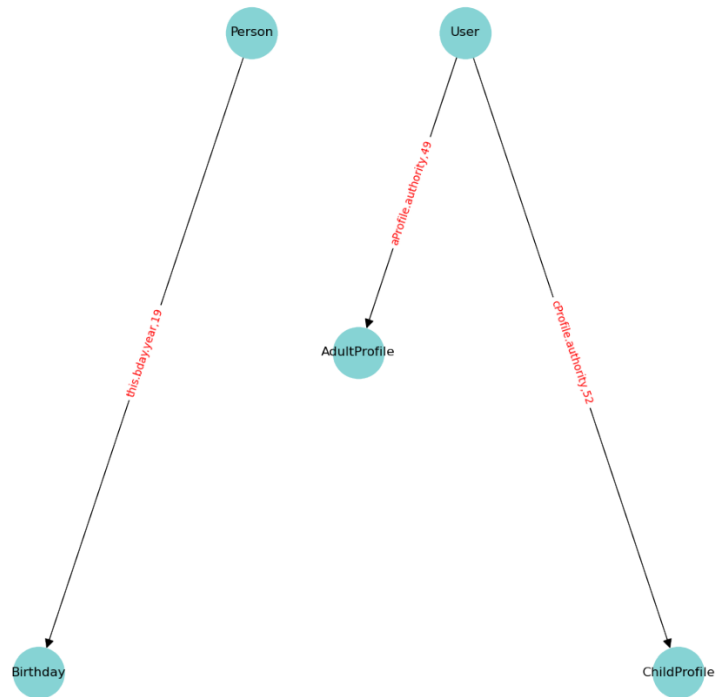
User(41-56) dependent by AdultProfile(34-39) because aProfile.authority,49 is member of AdultProfile Object,48

CASE 2: Used a Dynamically defined object

User(41-56) dependent by ChildProfile(27-32) because cProfile.authority,52 is member of ChildProfile Object,51

Dependencies: [[['Person', 'Birthday'], 'this.bday.year,19'], [['User', 'AdultProfile'], 'aProfile.authority,49'], [['User', 'ChildProfile'], 'cProfile.authority,52']]

Output graph



TEST 6: Input File

```

1.  class Engine{
2.      private int power;
3.      public Engine(int power){this.power = power;}
4.      int getPower() {return power;}
5.  }
6.  class Car{
7.      private Engine engine;
8.      public Car( Engine engine ){this.engine = engine;}
9.
10.     int getCarPower(){ return this.engine.getPower(); }
11. }
12.

```

Output txt

Root
├ Car
└ Engine

```

-----
[CLASS] Engine : 1-5
---Attribute Objects---
| None
---Dynamic Objects---
| None
---Parameters---
| int power : 3
--Member Usages---

```

[]

[CLASS] Car : 6-11

---Attribute Objects---

| Engine engine : 7

---Dynamic Objects---

| None

---Parameters---

| Engine engine : 8

--Member Usages--

[]

-----LOGS-----

Dependencies: []

Output graph



TEST 7:

Input File

```
1. class Superman{
2.     int health;
3.     int power;
4.
5.     public Superman(int health, int power) {
6.         super();
7.         this.health = health;
8.         this.power = power;
9.     }
10.
11.     void attack(Batman target) {
12.         target.health -= power;
13.     }
14. }
15.
16. class Batman{
17.     int health;
18.     int power;
19.
20.     public Batman(int health, int power) {
21.         this.health = health;
22.         this.power = power;
23.     }
24.
25.     void attack(Superman target) {
26.         target.health -= power;
27.     }
28. }
```

29. }

Output txt

Root

└─ Batman
└─ Superman

[CLASS] Superman : 1-14

---Attribute Objects---

| None

---Dynamic Objects---

| None

---Parameters---

| int power : 5

| Batman target : 11

--Member Usages---

['target.health,12']

[CLASS] Batman : 16-29

---Attribute Objects---

| None

---Dynamic Objects---

| None

---Parameters---

| int power : 20

| Superman target : 25

--Member Usages---

['target.health,26']

-----LOGS-----

CASE 3: Used a taken as parameter object

Superman(1-14) dependent by Batman(16-29) because target.health,12 is member of Batman Object,11

CASE 3: Used a taken as parameter object

Batman(16-29) dependent by Superman(1-14) because target.health,26 is member of Superman Object,25

Dependencies: [[['Superman', 'Batman'], 'target.health,12'], [['Batman', 'Superman'], 'target.health,26']]

Output graph



TEST 8: Empty File

Input File

Output txt

Root

-----LOGS-----

Dependencies: []

Output graph