به نام خدا

فاز اول پروژه کامپایلر گروه 2

اعضا: (سامان محمدی رئوف، محمد باربد امیرمزلقانی، محمد پارسا دورعلی، پارسا نوروزی)

Contents

مقدمه : 2
روش پیشنهادی و روند کار : 2
main.py: 2
Cast_CastBy.py: 6
Contain_ContainBy.py: 12
ارزیابی : 16
مشکلات و چالش ها : 20
نتیجه و کار های آتی : 21

مقدمه :

پروژه Open Understand ابزاری است برای بررسی ویژگی های کد های زبان جاوا . این ابزار توانایی یافتن رفرنس ها و موجودیت های کد جاوا را در دسترس قرار می دهد .همان طور که می دانید منظور از موجودیت تمامی فایل ها ، کلاس ها ، توابع و .. کد است و منظور از رفرنس مکان خاصی است که دو موجودیت با هم ارتباط پیدا می کنند .

در این فاز از پروژه ما به پیاده سازی رفرنس های cast/castby و contain/containin پرداختیم.فایل های Cast_CastBy و Contain_ContainBy را ساخته و فایل main را تغییر دادیم.

به طور کلی مسیریک پروژه جاوا گرفته میشود و فایل های آن از یکدیگر جدا میشوند. سپس به صورت جداگانه برای هر فایل listenerهایی ساخته می شوند و برای هر کدام Contain_ContainBy و Cast_CastBy کال میشود. بعد برای هرکدام اطلاعات مرتب سازی شده و به جدول های entity و reference در دیتابیس add میشوند. در نهایت نتیجه کار ذخیره شدن این دو رفرنس و موجودیت های آنان در دیتابیس است .

برای بخش های پیدا کردن محل cast , castBy , Contain, ContainIn روی درخت و پیش برد پروژه و تست نتایج همفکری شده است . بخش نوشته شدن کد ها برعهده آقای محمدی رئوف بوده است.

روش پیشنهادی و روند کار :

main.py:

این فایل شامل کلاس main میباشد. به طور کلی در این فایل تمام فایل های پروژه جاوا را گشته و ابتدا تمام کلاس های موجود در پروژه را پر می کند و در آرایه classes می ریزد و سپس برای همه فایل ها موارد cast , contain را پیدا می کند . با توابع addCastOrCastByReference , addContainAndContainBy داده های مربوط به این رفرنس ها به دیتابیس اضافه می شوند .

شرح کار این فایل به این صورت است که در ابتدا یک ابجکت از کلاس project ساخته میشود. سپس آدرس دیتایس و آدرس پروژه در متغیر ها داد میشود.

در این فایل دو تابع اضافه شده است که کار insertion به دیتابیس را انجام می دهند . با صدا زده شدن listener ها که در ادامه بیشتر توضیح داده شده اند ، در هر یک از بخش های cast , contain اطلاعاتی که نیاز است در دیتابیس ذخیره شوند را در آرایه های مختص خودشان ذخیره کرده ایم .

برای اینکه این اطلاعات را در دیتابیس ذخیره کنیم ، دو تابع addCastorCastByReferences, addContainAndContainBy را داریم . در هر یک، آرایه مربوط به بخش خودش را که حاوی اطلاعات مورد نیاز برای اضافه کردن به دیتابیس است به عنوان ورودی پاس می دهیم . حال روی این آرایه لوپ می زنیم و به ازای هر یک از عناصر آن موارد لازم را به دیتابیس اضافه می کنیم .

برای cast، ابتدا اطلاعات entity ای که بدان cast شده است را به دیتابیس اضافه می کنیم .در این قسمت می دانیم که تابع get_or_create در صورتی که آن داده قبلا در دیتابیس وجود داشته است آن را اضافه نمیکند

در ادامه اطلاعات entityای که حاوی این reference است را به دیتابیس اضافه می کنیم ودر نهایت دو نوع رفرنس cast , castBy را در جدول ReferenceModelاضافه می کنیم . برای contain نیز تابع addContainAndContainBy برای اضافه کردن داده ها به دیتابیس با همین منطق کار می کند .

```
def addCastorCastByReferences(self,cast , file_ent, file_address):
    for ent in cast:
        cast_To =
EntityModel.get_or_create(_kind=self.findKindWithKeywords(ent["kind"],
ent["modifier"]),
                                            _name=ent["name"],
                                            _parent=ent["parent"] if ent["parent"] is
not None else file_ent,
                                            _longname=ent["longname"],
                                            _contents=ent["content"]
                                            )[0]

        cast =
EntityModel.get_or_create(_kind=self.findKindWithKeywords(ent["p_kind"],
ent["p_modifier"]),
                                          _name=ent["p_name"],
                                          _parent=ent["p_parent"] if ent["p_parent"] is
not None else file_ent,
                                          _longname=ent["p_longname"],
                                          _contents=ent["p_content"]
                                          )[0]

        cast_ref = ReferenceModel.get_or_create(_kind=174, _file=file_ent,
_line=ent["line"],
                                                _column=ent["col"], _ent=cast_To,
_scope=cast)
        castBy_ref = ReferenceModel.get_or_create(_kind=175, _file=file_ent,
_line=ent["line"],
                                                  _column=ent["col"], _ent=cast,
_scope=cast_To)

def addContainAndContainBy(self, contain , file_ent , file_address ):
    for ent in contain:
        kind = self.findKindWithKeywords(ent["kind"], ent["modifiers"])
        if kind is not None :
            Contain_class = EntityModel.get_or_create(_kind = kind,
                                                _name = ent["name"],
                                                _parent = ent["parent"] if ent["parent"]
is not None else file_ent,
                                                _longname = ent["longname"],
                                                _contents = ent["content"])[0]
            Contain_package = EntityModel.get_or_create(_kind="72",
                                                        _name=ent["package_name"],
                                                        _parent=ent["package_parent"] if
ent["package_parent"] is not None else file_ent,

_longname=ent["package_longname"],

_contents=ent["package_content"])[0]
            contain_ref = ReferenceModel.get_or_create(_kind=176, _file=file_ent,
_line=ent["line"],
                                                       _column=ent["col"],
_ent=Contain_class, _scope=Contain_package)
            containIn_ref = ReferenceModel.get_or_create(_kind=177, _file=file_ent,
_line=ent["line"],
                                                         _column=ent["col"],
_ent=Contain_package, _scope=Contain_class)
```

سپس در انتهای فایل و درقسمت main، یک لیست خالی به نام classes میسازیم که بعدا برای cast استفاده میشود. تابع ()getListOfFiles را بر روی مسیر ذخیره شده اجرا میکنیم تا لیست فایل ها را دریافت کند و در متغیر بریزد.

```
classes = [] # for cast and cast by
for file_address in files:
    try:
        file_ent = p.getFileEntity(file_address)
        tree = p.Parse(file_address)
    except Exception as e:
        print("An Error occurred in file:" + file_address + "\n" + str(e))
        continue
    try:
        listener = implementListener(classes)
        p.Walk(listener, tree)
    except Exception as e:
        print("An Error occurred in file:" + file_address + "\n" + str(e))
```

سپس همانطور که در کد بالا مشاهده میکنیم بر روی فایل های دریافت شده لوپ میزنیم و برای هر آدرس ابتدا تابع ()getFileEntity را صدا میکنیم و انتیتی مربوط به آن را پیدا میکنیم سپس تابع ()Parse را صدا میزنیم و نتیجه را در متغیر درخت ذخیره میکنیم. و در صورت بروز مشکل ارور را چاپ میکنیم. سپس کلاس ()implementListener را صدا میکنیم و لیست کلاس ها که در ابتدا خالی است را به آن پاس میدهیم. این کلاس آرایه classes را پرخواهد کرد که به صورت مفصل در بخش Cast/CastBy توضیح داده شده است. سپس تابع ()Walk را با listener و tree صدا میزنیم و در صورت بروز مشکل ارور را چاپ میکنیم.

```
for file_address in files:

    try:

        file_ent = p.getFileEntity(file_address)

        tree = p.Parse(file_address)

    except Exception as e:

        print("An Error occurred in file:" + file_address + "\n" + str(e))

        continue
```

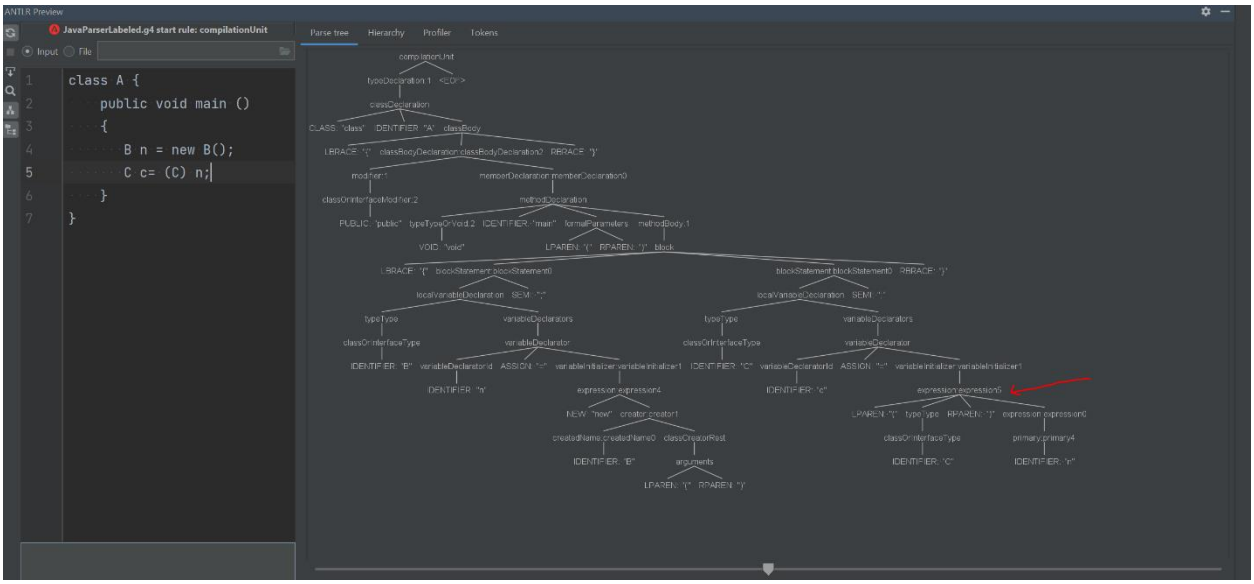
```
try:
    # cast
    listener = CastAndCastBy(classes)
    listener.cast = []
    p.Walk(listener, tree)
    p.addCastorCastByReferences(listener.cast , file_ent , file_address)
except Exception as e:
    print("An Error occurred for reference cast in file:" + file_address +
"\n" + str(e))

try:
    #contain
    listener = ContainAndContainBy()
    listener.contain = []
    p.Walk(listener,tree)
    p.addContainAndContainBy(listener.contain,file_ent,file_address)
except Exception as e:
    print("An Error occurred for reference contain in file:" + file_address +
"\n" + str(e))
```

در ادامه یک لوپ روی همه فایل ها اجرا میکنیم و به ازای هر آدرس یک سری عملیات انجام میدهیم که همانطور که کامنت گذاری شده ابتدا بخش های دیگر پروژه اجرا میشوند سپس به cast و contain میرسیم. در بخش cast شی کلاس CastAndCastBy() ساخته میشود و لیست ساخته شده که در ابتدای main پر شد به آن پاس داده میشود. سپس یک لیست خالی در قسمت cast، listener قرار داده میشود. تابع Walk() را با listener و tree صدا میزنیم و سپس addCastorCastByReferences() را با متغیر های مورد نیاز صدا میزنیم و در صورت بروز مشکل در هر بخشی از این کد ارور مناسب را چاپ میکنیم.

در بخش contain شی کلاس ContainAndContainBy() ساخته میشود .یک لیست خالی در قسمت contain، listener قرار داده میشود. تابع Walk() را با listener و tree صدا میزنیم و سپس addContainAndContainBy() را با متغیر های مورد نیاز صدا میزنیم و در صورت بروز مشکل در هر بخشی از این کد ارور مناسب را چاپ میکنیم.

:Cast_CastBy.py



(Cast in Tree)1 در این شکل نشان داده شده است که محل cast توسط expression5 در درخت مشخص می شود

در ابتدا یک نمونه کد را به عنوان ورودی می دهیم تا بررسی کنیم cast در کدام قسمت از درخت دیده می شود .همان طور که در شکل بالا می بینید ، expression5 می تواند نشان دهنده cast باشد . از این رو می توان فهمید که enterExpression5 می تواند برای پیدا کردن cast مورد استفاده قرار بگیرد . حال به بررسی فایل cast_castBy.py می پردازیم .

این فایل دارای سه class میباشد:

1) ClassEntities

این کلاس دارای یک constructor و 8 فیلد است:

فیلد ها:

1- modifiers
2- name
3- parent
4- kind
5- content
6- longname
7- type
8- value

2) implementListener

این کلاس دارای یک فیلد و یک constructor و یک تابع است:

فیلد ها:

1- classes[]

تابع ها:

1- __init__
2- enterClassDecleration

3) CastAndCastBy

این کلاس دارای 8 فیلد، یک constructor و 1 تابع میباشد:

فیلد ها:

1- classes[]
2- cast[]
3- c_name
4- c_longname
5- c_parent
6- c_kind
7- c_content
8- c_modifier

تابع ها:

1- __init__
2- enterExpression5

شرح کار این فایل به این صورت است که ابتدا فایل های javaParser و یک کلاس کمکی از بخش از پیش زده شده import میشود:

```
from openunderstand.gen.javaLabeled.JavaParserLabeledListener import JavaParserLabeledListener
from openunderstand.gen.javaLabeled.JavaParserLabeled import JavaParserLabeled
import openunderstand.analysis_passes.class_properties as class_properties
from db.api import open as db_open, create_db
from db.models import KindModel, EntityModel, ReferenceModel
from db.fill import main
```

سپس کلاس ClassEntities تعریف میشود. این کلاس دارای 8 فیلد است که در constructor هم زمان با گرفتن ورودی برای ساخت instance از کلاس پر میشوند. کاربرد این کلاس در کلاس ImplementListener و در تابع enterClassDecleration است.

این کلاس بیشتر کاربرد یک struct را دارد.

```
class ClassEntities:
    def __init__(self,name, parent , kind , content , longname  , modifiers):
        self.modifiers = modifiers
        self.name = name
        self.parent = parent
        self.kind = kind
        self.content = content
        self.longname = longname
        self.type = None
        self.value = None
```

بعد از این class، کلاس implementListener پیاده سازی میشود که کاربرد اصلیش برای پر کردن class entity های داخل هر فایل است. (قرار بود این بخش از دیتابیس برداشته شود ولی یکی از چالش های این بخش همین بود که نیاز شد دوباره این entity ها یافت شوند . این چالش در بخش چالشها بیشتر توضیح داده شده است .)

این کلاس یک فیلد آرایه classes دارد که توسط constructor مقدار دهی میشود. این آرایه classes همان آرایه ای است که در کد main به عنوان ورودی داده شد و قرار است کلاس های پروژه در آن ذخیره سازی شوند. پس با استفاده از تابع enterClassDecleratin هر جا که در فایل به کلاس برخورد کنیم فیلد های Entity آن را پر کرده و به آرایه classes اضافه میکنیم. و هر Record را با استفاده از ساختن instance از ClassEntities پر میکنیم.

ابتدا فیلد name را مقدار دهی میکنیم و بعد با استفاده از کلاس و توابع کمکی نوشته شده اسکوپ پرنت را گرفته و با استفاده از join فیلد longname را هم مقدار دهی میکنیم. سپس kind را به صورت هاردکد Class میدهیم. Content و بقیه فیلد را هم به طریق ها پر میکنیم. و با این کار آرایه classes را پر میکنیم.

```
class implementListener(JavaParserLabeledListener):
    classes = []

    def __init__(self , classes):
        self.classes = classes

    def enterClassDeclaration(self, ctx: JavaParserLabeled.ClassDeclarationContext):
        name = ctx.IDENTIFIER().getText()
        scope_parents = class_properties.ClassPropertiesListener.findParents(ctx)
        if len(scope_parents) == 1:
            scope_longname = scope_parents[0]
        else:
            scope_longname = ".".join(scope_parents)

        EntityClass = ClassEntities(name, scope_parents[-2] if len(scope_parents) > 2 else None, "Class",
ctx.getText(),
                                    scope_longname ,
class_properties.ClassPropertiesListener.findClassOrInterfaceModifiers(
                                                ctx))
        self.classes.append(EntityClass)
```

در نهایت آخرین کلاس پیاده سازی شده CastAndCastBy میباشد. دارای دو فیلد آرایه classes و cast است. و 6 فیلد :

c_name

c_longname

c_parent

c_kind

c_contet

c_modifiers

```
class CastAndCastBy(JavaParserLabeledListener):

    classes = []
    cast = []
```

```
def __init__(self , classes):
    self.classes = classes
    self.c_name = ""
    self.c_longname = ""
    self.c_parent = ""
    self.c_kind = ""
    self.c_content = ""
    self.c_modifiers = ""
```

در constructor آرایه classes با آرگومان classes ای که از بیرون میگیرد پر میشود. Classes پر کننده همانی است که در کلاس قبلی مقدار دهی شده است.

```
def __init__(self , classes):
    self.classes = classes
```

بعد وارد تنها تابع این کلاس برای شناسایی و پر کردن داده ها میشویم به نام: enterExpression5.
طی بررسی به عمل آمده و تحلیل کد بر روی درخت انتلر به این نتیجه رسیدیم که Entity ای که cast شده در بخش enterExpression5 قرار میگیرد.

بنابراین این تابع را برای شناسایی cast شدن بازنویسی میکینم:

```
def enterExpression5(self, ctx:JavaParserLabeled.Expression5Context):
    self.c_name = ""
    self.c_longname = ""
    self.c_parent = ""
    self.c_kind = ""
    self.c_content = ""
    self.c_modifiers = ""

    name = ctx.typeType().getText()
    scope_parents = class_properties.ClassPropertiesListener.findParents(ctx)
    [line, col] = str(ctx.start).split(",")[3].split(":")  # line, column
    col = col[:-1]
    print("line"+line)
    print("col" + col)
    print("name : " + name)
```

```
    if len(scope_parents) >= 2:
        parent = scope_parents[-2]
    else:
        parent = None
    for ent in self.classes:
        if ent.name == name:
            self.c_name = name
            self.c_longname = ent.longname
            self.c_parent = ent.parent
            self.c_kind = ent.kind
            self.c_content = ent.content
            self.c_modifiers = ent.modifiers
    print("parent :" + parent)
    for ent in self.classes:
        if self.c_name != "" :
            if ent.name == parent:
                self.cast.append({"name": self.c_name,"longname":self.c_longname , "parent" : self.c_parent
,
                                  "kind" : self.c_kind , "content" : self.c_content , "modifier" :
self.c_modifiers,
                                  "p_name": ent.name, "p_longname": ent.longname, "p_parent":
ent.parent,
                                  "p_kind": ent.kind, "p_content": ent.content, "p_modifier": ent.modifiers
                                  ,"line":line, "col":col})

    print(self.cast)
```

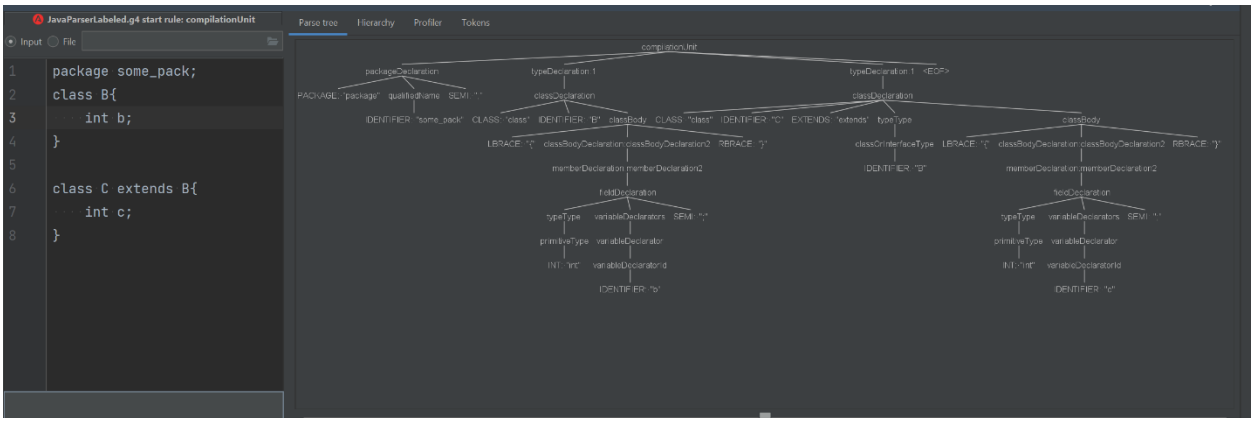
در ابتدا با گرفتن ورودی تابع name را مقدار دهی میکنیم. بعد طبق همان رویه که در implementListener، اسکوپ پرنت را به دست آوردیم مجدد هین کار را کرده و scope_parent را به دست می آوریم. بعد با استفاده از تابع start، line و column را بدست میاوریم. سپس با توجه به scopr_parent مقدار دهی parent را انجام میدهیم.

بعد داخلclasses که حاوی کلاس های پروژه است لوپ میزنیم و entity مورد نظر را با match کردن نام ها پیدا میکنیم سپس تمام فیلد های کلاس را مطابق فیلد های classes مقدار دهی میکنیم.

سپس همین کار را با parent کرده و فیلد های آن را پیدا میکنیم.

در نهایت مجموعه فیلد های خود کلاس cast و parent آنرا در قالب یک dictionary به آرایه cast اضافه میکنیم تا در main بتوان آن داده ها را برای جداول entity و reference استفاده کرد.

:Contain_ContainBy.py



در هر فایل می توان با دیدن package decelartion پکیج مربوطه را یافت و بعد از آن کلاس ها را با class decelartion تشخیص داد (package in 2 Tree)

در زبان جاوا تعریف package در ابتدای فایل قرار می گیرد ، سپس کلاس هایی در آن فایل تعریف می شوند جزو آن پکیج می شوند . پس تنها لازم است تا در هر فایل ابتدا package را توسط enterPackageDeclaration پیدا کنیم و سپس کلاس های آن را توسط enterClassDeclaration پیدا کنیم . حال به توضیح کد می پردازیم .

این فایل دارای یک کلاس کلی به نام ContainAndContainBy میباشد که خود دارای دو لیست contain و packageInfo است.

```
class ContainAndContainBy(JavaParserLabeledListener):
    contain = []
    packageInfo = []
```

در ادامه دو فانکشن enterPackageDeclaration و enterClassDeclaration را داریم که به توضیح جداگانه هر کدام میپردازیم :

1) enterPackageDeclaration

```
def enterPackageDeclaration(self, ctx:JavaParserLabeled.PackageDeclarationContext):
    self.packageInfo = []
    longname = ""
    for x in range(len(ctx.qualifiedName().IDENTIFIER())):
        if  x == 0:
            longname = str(ctx.qualifiedName().IDENTIFIER()[x])
        else:
            longname = longname + "." + str(ctx.qualifiedName().IDENTIFIER()[x])

    self.packageInfo.append({"name":ctx.qualifiedName().IDENTIFIER()[-1],
```

```
                "longname":longname,
                "kind":"Package",
                "contents" : "",
                "parent" : None ,
                "type" : None,
                "value" : None
                })
```

همانطور که در شکل نیز دیده میشود این تابع عملکرد ساده ای دارد هر کجا در طول فایل به package entitiy رسید تمامی اطلاعات مورد نیاز برای پر کردن table را اعم از name و longname و ... به لیست packageInfo که در بالاتر اشاره شد اضافه میکند.

2) enterClassDeclaration

```
def enterClassDeclaration(self, ctx:JavaParserLabeled.ClassDeclarationContext):
    name = ctx.IDENTIFIER().getText()
    print(ctx.IDENTIFIER().getText())
    [line, col] = str(ctx.start).split(",")[3].split(":")  # line, column
    col = col[:-1]
    scope_parents = class_properties.ClassPropertiesListener.findParents(ctx)

    if len(scope_parents) == 1:
        scope_longname = scope_parents[0]
    else:
        scope_longname = ".".join(scope_parents)
```

حال در تابع enterClassDeclaration که هر جا به موجودیت تعریف کلاس رسید فراخوانی میشود به ترتیب این کارها صورت میگیرد :

ابتدا نام آن را میگیریم سپس با استفاده از ctx.start خط و ستون تعریف شده را پیدا میکنیم سپس با کمک تابعی که در classProperties تعریف کردیم والد های آن را میابیم :

```
@staticmethod
def findParents(c):  # includes the ctx identifier
    parents = []
    current = c
    while current is not None:
        if type(current).__name__ == "ClassDeclarationContext" or type(current).__name__ ==
"MethodDeclarationContext"\
```

```
                or type(current).__name__ == "EnumDeclarationContext"\
                or type(current).__name__ == "InterfaceDeclarationContext"\
                or type(current).__name__ == "AnnotationTypeDeclarationContext":
            parents.append(current.IDENTIFIER().getText())
        current = current.parentCtx
    return list(reversed(parents))
```

در اینجا یک حلقه while میزنیم که تا وقتی والد آن not None باشد به طور بازگشتی current را تغییر دهد و تمامی موارد یافت شده را در یک لیست نگهداری کند و در نهایت به صورت برعکس بازگرداند .

```
if len(scope_parents) == 1:
    scope_longname = scope_parents[0]
else:
    scope_longname = ".".join(scope_parents)

scope_longname = "." + scope_longname
packageName = self.packageInfo[0]["name"]
packageLongName = self.packageInfo[0]["longname"]
scope_longname = packageLongName + scope_longname
packageKind = self.packageInfo[0]["kind"]
packageContent = self.packageInfo[0]["contents"]
packageParent = self.packageInfo[0]["parent"]
packageType = self.packageInfo[0]["type"]
packageValue = self.packageInfo[0]["value"]

parent = scope_parents[-2] if len(scope_parents) > 2 else None
kind ="Class"
modifiers = class_properties.ClassPropertiesListener.findClassOrInterfaceModifiers(
    ctx)
content = ctx.getText()
```

در ادامه enterClassDeclaration چک میکنیم اگر والدی نداشت یعنی در واقع قبلش فایل بود اسم خودش را بذارد و قبلش یک نقطه بگذارد . در ادامه اطلاعات پکیج را به کمک تابعی که قبلا توضیح دادیم پیدا میکنیم و modifier ها را نیز با تابع کمکی که در classProperties تعریف شده پیدا میکنیم :

```
@staticmethod
def findClassOrInterfaceModifiers(c):
    m = ""
    modifiers=[]
    current = c
    while current is not None:
        if "typeDeclaration" in type(current.parentCtx).__name__:
            m=(current.parentCtx.classOrInterfaceModifier())
            break
        current = current.parentCtx
    for x in m:
        modifiers.append(x.getText())
    return modifiers
```

در نهایت تمامی اطلاعات به دست آمده را شامل اطلاعات کلاس و پکیج و محل رفرنس را به لیست contain اضافه میکنیم :

```
self.contain.append({
                    "package_name":packageName.getText(),
                    "package_longname" :packageLongName,
                    "package_kind" : packageKind,
                    "package_content" : packageContent ,
                    "package_parent" : packageParent,
                    "package_type" : packageType ,
                    "package_value" : packageValue,
                    "name":name ,
                    "longname" : scope_longname,
                    "parent" : parent,
                    "kind" : kind,
                    "line" :line,
                    "col" : col,
                    "modifiers" : modifiers,
                    "content":content,
                    "type" : None ,
                    "value" : None
                    })
print(self.contain)
```

ارزیابی :

برای این کار از فایل test.py استفاده شده است که نتایج را از خود understand می گیرد و نمایش می دهد .

با اجرا شدن فایل main.py نتایج بدست آمده از Open understand نیز مشخص می شود .

با استفاده از کد زیر در test.py می توان نتایج cast را توسط خود understandپیدا کرد .

```
try:
    import understand as und
except ImportError:
    print("Can not import understand")
db =
und.open("C:/Users/98910/university/Term6/Courses/Compiler/Project/Compiler_OpneUnders
tand/OpenUnderstand-
8b69f877f175bf4ccd6c58ec3601be655157d8ca/benchmark/myJavaTest/myJavaTest.udb")

counter = 0
for ent in db.ents():
    for ref in ent.refs():
        if ref.kindname() == "Cast": # and ref.file().name() == "printLog.java":
            counter = counter + 1
            print(f"ent name: {ent.name()}, ent longname: {ent.longname()}, \n"
                  f"ent parent: {ent.parent()}, ent kind: {ent.kind()}, ent value:
{ent.value()},\n"
                  f"ent type: {ent.type()}, ent contents: {ent.contents()}")
            print("++++++++++++++++++++++++")
            # print(f"file kind: {ref.file().kind()}, parent: {ref.file().parent()},
long name: {ref.file().longname()}"
            #       f"\nvalue: {ref.file().value()}, type: {ref.file().type()},
contents: {ref.file().contents()}, name: {ref.file().name()}")

            print(f"entity: {ent}\n, ref: {ref}\n ref.scope: {ref.scope()}, ref.ent:
{ref.ent()}\n"
                  f"ref.line: {ref.line()}, ref.col: {ref.column()}, ref.file:
{ref.file().name()}")
            print("-------------------------------------------------------")
            print(f"ref.ent.name:{ref.ent().name()},
ref.ent.longname:{ref.ent().longname()} ,ref.ent.kind:{ref.ent().kind()}\n"
                  f"ref.ent.parent:{ref.ent().parent()},
ref.ent.value:{ref.ent().value()},ref.ent.type:{ref.ent().type()}\n"
                  f"ref.ent.contents:{ref.ent().contents()}")
            print("-------------------------------------------------------")
```

با استفاده از کد زیر نیز می توان نتیجه Contain های یک فایل را توسط understand پیدا کرد .

```
try:
    import understand as und
except ImportError:
    print("Can not import understand")
db =
und.open("C:/Users/98910/university/Term6/Courses/Compiler/Proje
ct/Compiler_OpneUnderstand/OpenUnderstand-
8b69f877f175bf4ccd6c58ec3601be655157d8ca/benchmark/myJavaTest/my
JavaTest.udb")

counter = 0
for ent in db.ents():
    for ref in ent.refs():
        if ref.kindname() == "Contain": # and ref.file().name()
== "printLog.java":
            counter = counter + 1
            print(f"ent name: {ent.name()}, ent longname:
{ent.longname()}, \n"
                  f"ent parent: {ent.parent()}, ent kind:
{ent.kind()}, ent value: {ent.value()},\n"
                  f"ent type: {ent.type()}, ent contents:
{ent.contents()}")
            print("++++++++++++++++++++++++++")
            # print(f"file kind: {ref.file().kind()}, parent:
{ref.file().parent()}, long name: {ref.file().longname()}"
            #       f"\nvalue: {ref.file().value()}, type:
{ref.file().type()}, contents: {ref.file().contents()}, name:
{ref.file().name()}")

            print(f"entity: {ent}\n, ref: {ref}\n ref.scope:
{ref.scope()}, ref.ent: {ref.ent()}\n"
                  f"ref.line: {ref.line()}, ref.col:
{ref.column()}, ref.file: {ref.file().name()}")
            print("-----------------------------------------
-----------")
            print(f"ref.ent.name:{ref.ent().name()},
ref.ent.longname:{ref.ent().longname()}
,ref.ent.kind:{ref.ent().kind()}\n"
                  f"ref.ent.parent:{ref.ent().parent()},
ref.ent.value:{ref.ent().value()},ref.ent.type:{ref.ent().type()
}\n"
                  f"ref.ent.contents:{ref.ent().contents()}")
            print("-----------------------------------------
-----------")
```

برای مثال کد زیر را در نظر بگیرید :

```
package some_pack;
class B{
    int b;
}

class C extends B{
    int c;
}
class A {
    public void main ()
    {
       B n = new B();
       C c= (C) n;

    }
}
```

در این فایل جاوا که مشاهده می کنید ، هم ارتباط contain به واسطه some_pack و کلاس ها وجود دارد و هم رابطه cast وجود دارد که در آن شی n از کلاس B به کلاس C تبدیل می شود .

نتیجه understand api به صورت زیر می باشد .

```
ent name: A.main, ent longname: some_pack.A.main,

ent parent: some_pack.A, ent kind: Public Method, ent value: None,

ent type: void, ent contents:    public void main ()

  {

    B n = new B();

    C c= (C) n;


  }

++++++++++++++++++++++++++

entity: A.main
```

```
, ref: Cast some_pack.C javaTest.java(13)
 ref.scope: A.main, ref.ent: some_pack.C
ref.line: 13, ref.col: 13, ref.file: javaTest.java
----------------------------------------------------
ref.ent.name:some_pack.C, ref.ent.longname:some_pack.C ,ref.ent.kind:Class
ref.ent.parent:javaTest.java, ref.ent.value:None,ref.ent.type:None
ref.ent.contents:class C extends B{
   int c;
}
```

حال می توان همین نتیجه را بعد از اجرا کد در main.py و در database نهایی مشاهده کرد .

Table: entitymodel

Filter in any column

	_id	_kind_id	_parent_id	_name	_longname	_value	_type	_contents
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	1	NULL	javaTest.java	C:...	NULL	NULL	package some_pack;...
2	2	39	1	main	A.main	NULL	void	['scopecontent']
3	3	95	A	B	A.B	NULL	NULL	classB{intb;}
4	4	72	1	some_pack	some_pack	NULL	NULL	
5	5	95	1	C	C	NULL	NULL	classCextendsB{intc;}
6	6	95	1	A	A	NULL	NULL	classA{publicvoidmain()...
7	7	95	1	B	some_pack.B	NULL	NULL	classB{intb;}
8	8	95	1	C	some_pack.C	NULL	NULL	classCextendsB{intc;}
9	9	95	1	A	some_pack.A	NULL	NULL	classA{publicvoidmain()...

Table: referencemodel

	_id	_kind_id	_file_id	_line	_column	_ent_id	_scope_id
	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	190	1	12	13	3	2
2	2	191	1	12	13	2	3
3	3	192	1	1	8	4	1
4	4	193	1	1	8	1	4
5	5	174	1	13	12	5	6
6	6	175	1	13	12	6	5
7	7	176	1	2	0	7	4
8	8	177	1	2	0	4	7
9	9	176	1	6	0	8	4
10	10	177	1	6	0	4	8
11	11	176	1	9	0	9	4
12	12	177	1	9	0	4	9

جداول دیتابیس پر شده توسط OpenUnderstand 1

همان طور که می بینید ، در understand api یک cast به همراه مشخصات آن پیدا شده بود . در دیتابیس نیز ، آیدی 174 که مربوط به cast می باشد تنها یک عدد است . (آیدی 175 نیز مربوط به cast by می باشد) و entity های مربوط به این cast را نیز می توانید در جدول اول مشاهده کنید .

مشکلات و چالش ها :

یکی از چالش های پروژه در قسمت پیدا کردن entity های مربوط به cast بود . همان طور که در بخش main.pyمتوجه شدیم ، یک حلقه وجود دارد که روی تمام فایل های یک پروژه جاوا لوپ می زند . این حلقه هر بار با دیدن فایل جدید موارد موجود در آن فایل را پیدا می کند .

در cast اما یک کلاس به کلاس دیگری cast شده و این referenceدر یک کلاس سومی اتفاق می افتد . ممکن است اطلاعات کلاسی که میخواهیم به عنوان entity برای cast بدان اشاره کنیم ، در فایل های بعدی باشد که هنوز مشخصات entity های آن یافت نشده و به عنوان entity شناخته نشده است .

ولی برای اینکه در cast بدان رفرنس بدهیم لازم است تا آن را داشته باشیم . برای این کار همان طور که اشاره شد در main یک حلقه دیگر فقط برای پیدا کردن کلاس های پروژه اضافه کردیم .

```
for file_address in files:
    try:
        file_ent = p.getFileEntity(file_address)
        tree = p.Parse(file_address)
    except Exception as e:
        print("An Error occurred in file:" +
file_address + "\n" + str(e))
        continue
    try:
        listener = implementListener(classes)
        p.Walk(listener, tree)
    except Exception as e:
        print("An Error occurred in file:" +
file_address + "\n" + str(e))
```

با این کار تمام کلاس های پروژه شناخته می شوند و در حلقه اصلی main می توان به راحتی بدان رفرنس داد .

نتیجه و کار های آتی :

در انتها تمامی cast , castBy , Contain , ContainIn های یک پروژه جاوا یافت می شوند و در دیتابیس ریخته می شوند . همان طور که در ارزیابی نشان دادیم نتایج در دیتابیس قابل مشاهده هستند .