Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozáselmélet és Szoftvertechnológiai Tanszék

Ekvivalens Python forráskód-párok generálása

Témavezető:
Szalontai Balázs
doktorandusz

Szerző:
Verebics Péter
programtervező informatikus BSc

Budapest, 2024

Tartalomjegyzék

1. fejezet

Bevezetés

Szakdolgozatom témája Python forráskódok átalakítása, és ezen átalakítások szemléltetése. A motiváció az átalakítások mögött egy olyan adathalmaz generálása amiben ekvivalens és nem ekvivalens forráskód-párok egyaránt szerepelnek. Egy ilyen adathalmazt felhasználhatunk egy mélytanuló neuronháló tanítására, ami forráskód-párok ekvivalenciáját dönti el.

Az ekvivalencia eldöntése fontos feladat, mivel egyre több, kódokat gépi tanulással refaktoráló eszköz létezik. Ezek az eszközök egy kódot változtatva sokszor a kód jelentését is megváltoztatják. Egy ekivalenciát eldöntő neuronháló képes lenne kiszűrni az ilyen eszközök által generált rossz eredményeket, javítva az eszközök hatékonyságán. Tehát ekvivalens és nem ekvivalens kódokat generálva felépíthetünk egy adathalmazt, ami ekvivalenciával felcímkézett kódpárokat tartalmaz, és alkalmas egy fent leírt neuronháló tanítására.

Az általam implementált átalakítások absztrakt szintaxisfák (AST-k) módosításával működnek. Egy forráskód fordítása alatt a szemantikus elemző előállítja a kód AST-jét, ami a kódot egy fa adatstruktúrával reprezentálja. Az AST-nek a szemantikus elemzésben van szerepe, de használhatjuk kódok átalakítására is, mivel visszaakítható forráskóddá.

Az általam megvalósított átalakítások a Python *ast* modult [1] használják, amely része a Python standard könyvtárának.

Az átalakításokat szabályok végzik. Átalakításkor a Python kódból létrehozott AST-n végrehajthatunk egy szabályt. A szabály megváltoztatja az AST-t, amit ha visszaalakítunk kóddá, egy megváltozott Python kódot kapunk.

A szakdolgozatomban ekvivalens és nem ekvivalens szabályokat is definiálok. Egy szabály akkor tekinthető ekvivalensnek, ha a kód szemantikáját nem változtatja meg. Például a Python-ban is teljesül a valós számok körében a szorzás kommutatív tulajdonsága. Tehát, ha egy Python kódban két szám szorzásánál a bal és jobb operandust megcseréljük, akkor a szorzás eredménye nem változik, vagyis ez az átalakítás ekvivalens. Ez a példa természetesen nagyon egyszerű, a szakdolgozatomban összetettebb átalakítások is szerepelnek.

Az adathalmazban az ekvivalens kódok generálásához saját szabályok mellett a *ruff* Python linter és formatter szabályait is felhasználtam. A *ruff* már létező Python lintereket implementál Rust programozási nyelven, így sok más Python refaktoráló eszköz szabályait is képes elvégezni, amlyek tökéletesek az általam implementált szabályok kiegészítésére.

A szakdolgozatom következő fejezeteiben az adathalmaz generálására és az átalakítások szemléltetésére alkalmas szoftver használatát és működését részletezem.

2. fejezet

Felhasználói dokumentáció

Az általam fejlesztett szoftver két felhasználói felülettel rendelkezik. Az egyik egy parancssoros (CLI) program az adathalmaz generálásához, a másik egy grafikus (GUI) alkalmazás az átalakítások szemléltetéséhez, és az adathalmaz böngészéséhez. Mindkét alkalmazás felületének nyelve angol.

2.1. Futtatási környezet

A szoftver egy Python 3.10-es vagy újabb verziójú Python interpreterrel futtatható. Futtatás előtt a szoftver függőségeit installálni kell a *pip* csomagkezelővel. Ezt a legegyszerűbben a szoftver forráskódjának könyvtárából tehetjük meg, a következő parancs kiadásával:

```
$ pip install --editable .
```

2.2. Adatbázis beállítása

Az adathalmaz generálásához szükség van egy *mongodb* adatbázis kliensre [2]. Az adatbázis kiszűri a kódpárok generálása közben a duplikált kódokat, és az adatok lekérdezését is megkönnyíti.

Az adatbázis elérését a forrás könyvtárában a *config/default.ini* útvonal alatt található konfigfájlban lehet beállítani. Egy adatbázist három paraméter határoz meg: *host*, *port*, *database* (az adatbázis neve). Ha szükséges a konfigfájl alapértékeit átírhatjuk.

2.3. Adathalmazt generáló CLI

A szoftver CLI alkalmazásával van lehetőségünk az adathalmaz generálására egy adott csv fájlban található kódokból vagy egy könyvtárban taláható forrásfájlokból. Adathalmazt a következő paranccsal generálhatunk:

```
$ python -m source.persistor <mode> <path>
```

A parancs paraméterei a következők:

1. *mode* - az adatok forrásának típusa, lehetséges értékek:
    - *csv* - csv fájlból olvassa a forrásfájlok tartalmát
    - *dir* - könyvtárból rekurzívan olvassa a forrásfájlokat
2. *path* - az adatok forrásának elérési útvonala

Ha megadtuk a parancsot, a program megpróbálja a forráskódok olvasását, ha az input nem megfelelő akkor leáll.

Futás közben a program kiírja az éppen feldolgozott forráskóddal kapcsolatos információkat, például a kódon végzett átalakítások számát, és azt, hogy el tudta-e menteni az átalakítások eredményeit.

```
reading csv, be patient this might take a while...
-------------------------------------------------------------------------------------------------[0]
No changes to save.
-------------------------------------------------------------------------------------------------[1]
Fixed 13 errors.
1 file reformatted
1 file reformatted
1 file reformatted
Inserted 3 changes on hash f3842737e0fe9141df61f299c99e65d9b20a41ae3c76644ef663483aec8aeb31.
-------------------------------------------------------------------------------------------------[2]
Fixed 12 errors.
1 file reformatted
1 file reformatted
1 file reformatted
Inserted 3 changes on hash bdd72cdcf458519e9e88afa499c0e6363e427fa7ea8c9fe5484a426babed40c1.
```

2.1. ábra. A CLI alkalmazás futás közben

Ha a program végigolvasta a csv fájlt vagy a könyvtárban található forrásfájlokat, leáll. Amikor a program megállt, a forráskód-párok már az adatbázisban vannak.

A *mongoexport* eszköz segítségével az adatbázisból a forráskód-párokat egy csv fájlba exportálhatjuk.

2.4. Átalakításokat szemléltető GUI

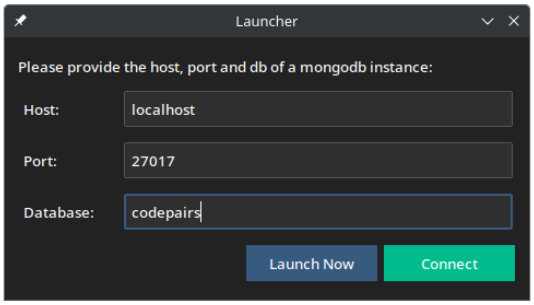
A szoftver GUI alkamazása szemlélteti az átalakításokat. Kipróbálhatunk vele egy vagy több átalakító szabályt, vizualizálhatjuk kódok absztrakt szintaxis fáját, és az adatbázisba bekerült átalakítások eredményét is megnézhetjük.

2.4.1. Alkalmazás indítása

Az alkalmazás a forrás könyvtárából indítható a következő paranccsal:

```
$ python -m source
```

A parancs kiadása után felugró ablakban beállíhatjuk az adatbázis kapcsolathoz szükséges paramétereket: a *host*, *port* illetve *database* értékeit. A *Connect* gombra kattintva az alkalmazás adatbáziseléréssel indul, ha a megadott adatbázishoz 10 másodperc alatt kapcsolódni tud, különben adatbáziselérés nélkül fog elindul. Adatbáziselérés nélkül a *Launch Now* gombbal indíthatjuk az alkalmazást.



2.2. ábra. Az alkalmazást indító ablak

2.4.2. Alkalmazás felülete

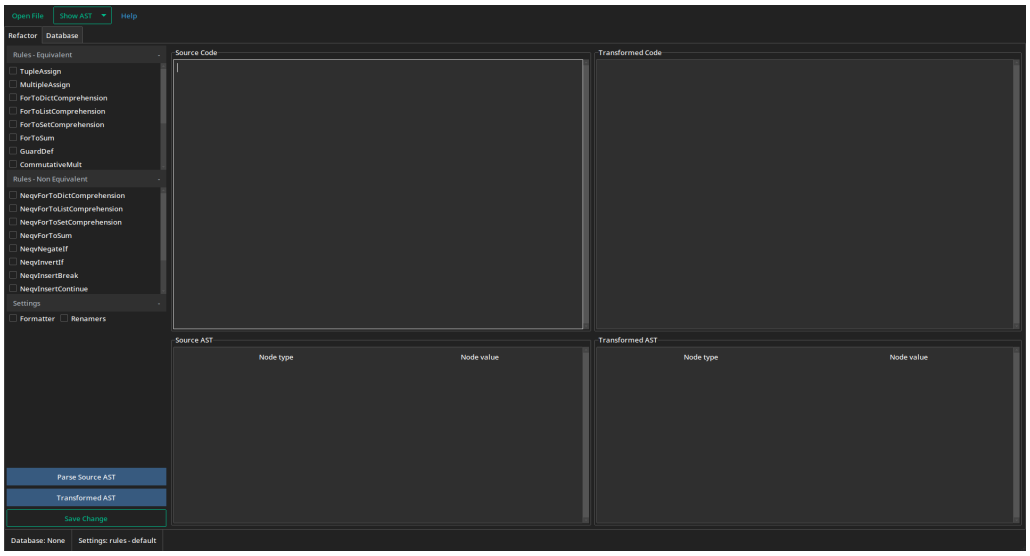
Az GUI alkalmazás felülete funkciók szerint két fő nézetre osztható, a refaktoráló és az adatbázis-böngésző nézetre. A menü gombjai és az állapotsor szövegei a két fő nézeten kívül helyezkednek el.

A refaktoráló nézetben (*Refactor* tab) egy kódon ekvivalensen és nem ekvivalensen átalakító szabályokat próbálhatunk ki, és elmenthetjük a szabályok által végzett átalakítások eredményeit.

Az adatbázis-böngésző (*Database* tab) nézetben az adatbázisba bekerült kódpárokat tekinthetjük meg.

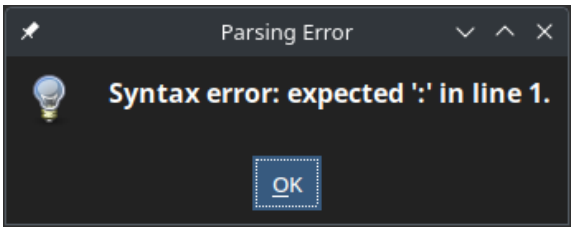
2.4.3. Refaktoráló nézet

Indítás után a felhasználót a refaktoráló nézet fogadja. Egy Python forráskód átalakításához a kódot begépelhetjük a *Source Code* szöveges input mezőbe, vagy egy *.py* fájlból is betölthetjük a menüben látható *Open File* gombra kattintva.

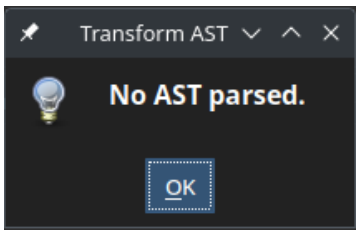


2.3. ábra. Refaktoráló nézet az indítás után

Az átalakítás előtt a begépelt vagy betöltött forráskódból létre kell hozni egy AST-t az elemező (parser) futtatásával, ezt a *Parse Source AST* gombbal tehetjük meg. Ha a megadott forráskódban szintaxis hiba található, vagy valami egyéb okból kifolyólag nem elemezhető, akkor az alkalmazás ezt jelzi egy felugró párbeszéd-ablakkal. Akkor is jelez, ha parse-olás nélkül klikkelünk az átalakító gombra.



(a) elemzési hiba esetén

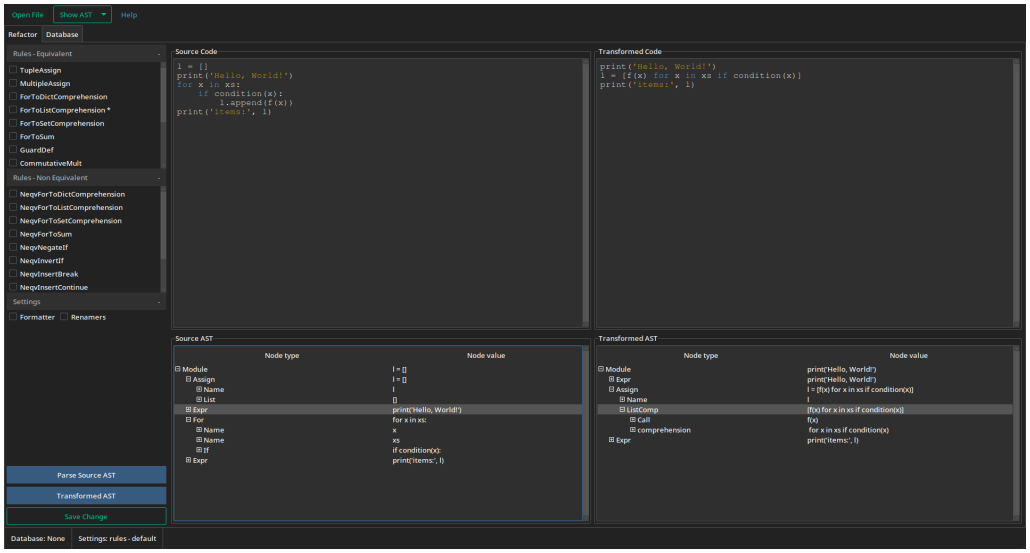


(b) hiányzó AST esetén

2.4. ábra. Hibákat jelző párbeszéd-ablakok

Sikeres elemzés után az AST felépítését a *Source AST* fa-nézeten láthatjuk, az első oszlopban a csúcs típusa, a második oszlopban a csúcs szintaxis fájából generált kód látható.

Elemzés után a fát átalakíthatjuk a *Transform AST* gombra kattintva, ekkor az átalakított fa megjelenik a bal oldali *Transformed AST* fa-nézeten, az átalakított fából generált kód pedig a *Transformed Code* readonly szövegdobozban.



2.5. ábra. Példa egy átalakítás eredményére

Az alkalmazás összesen 28 átalakító szabályt használ, ezek közül 16 ekvivalens és 12 nem ekvivalens eredményt állít elő. Az alkalmazás indításakor az összes ekvivalens szabály ki van választva, ez az alkalmazás alapbeállítása, amit a *'rules - default'* felirat jelez az állapotsorban.

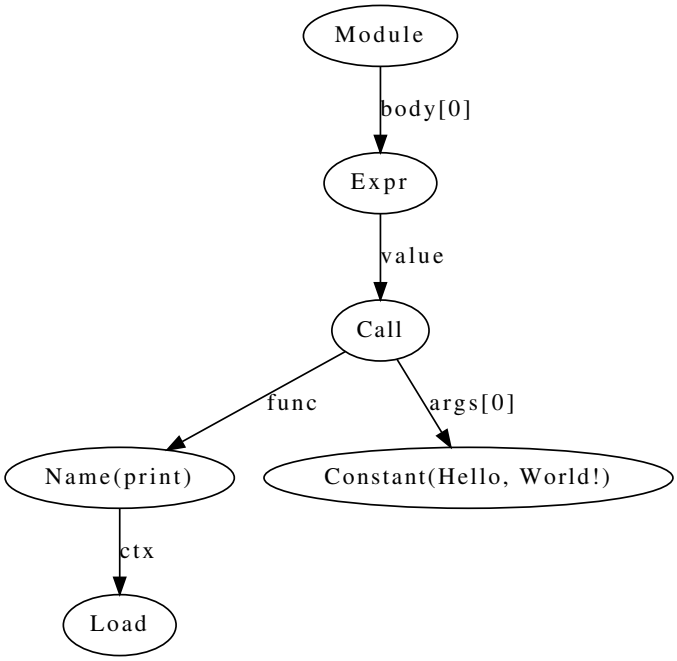
Lehetőségünk van általunk választott szabályok alkalmazására is. A szabályok listája a bal oldali panelen látható. Minden szabály előtt van egy checkbox amivel a szabályt kiválaszthatjuk. Lehetőségünk van egy vagy több szabály kiválasztására is, így könnyen tesztelhetjük egy szabály működését. Ha vannak kiválasztott szabályok, azt a *'rules - custom'* felirat jelzi az állapotsorban.

Átalakításkor a szabályok a bal oldali panelen látható sorrendben, fentről lefele kerülnek végrehajtásra. A panelen a szabályokon kívül található még két checkbox is, ezekkel a *ruff* formatter és az átnevező átalakítások alkalmazását tudjuk beállítani.

Miután átalakítottunk egy forráskódot, kimenthetjük az átalakítás eredményét az adatbázisba, ha van adatbázis kapcsolat. Ezt a refaktoráló nézet bal alsó sarkában elhelyezkedő *'Save Change'* gombra kattintva tehetjük meg. Ha az átalakítást nem lehet elmenteni akkor azt az alkalmazás párbeszéd-ablakban jelzi.

2.4.4. AST-k vizualizálása fagráffal

Az alkalmazás fagráfként is tud AST-ket vizualizálni. Az általunk megadott vagy átalakított kód AST-jének fagráfját a menüben látható *Show AST* lenyíló menügombbal vizualizálhatjuk. A gombra klikkelve két opció közül választhatunk: a *Source* gomb az általunk megadott kód, a *Transformed* gomb pedig az átalakított kód AST-jét vizualizálja, ha ezek léteznek. Az alkalmazás az elkészült fagráf ábráját egy felugró ablakban nyitja meg. Az alábbi ábrán például a *helloworld* Python kódjának AST-jét láthatjuk:

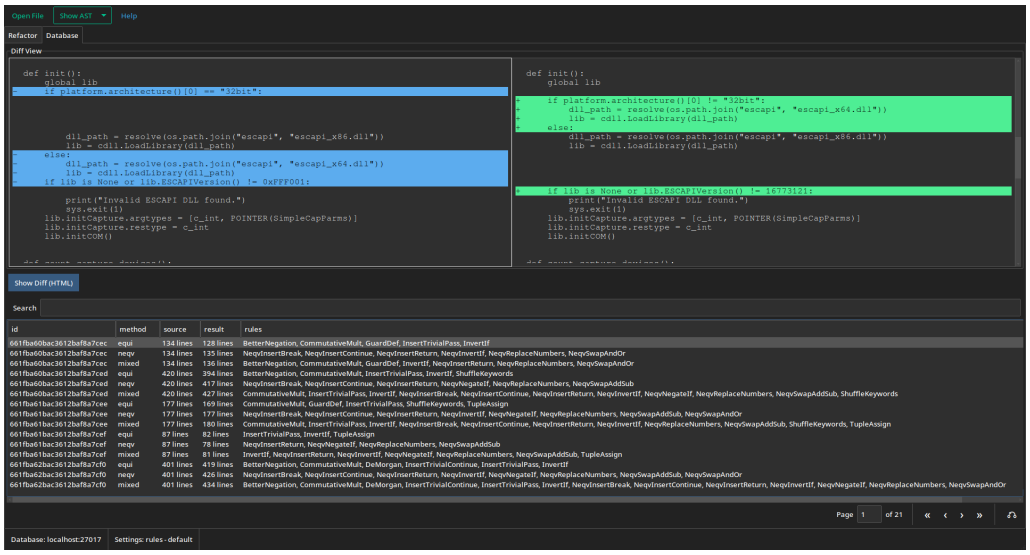


2.6. ábra. A *helloworld* program AST-je

2.4.5. Adatbázis-böngésző nézet

Ebben a nézetben megtekinthetjük az adatbázisba bekerült forráskód-párokat. A nézet csak akkor jön létre, ha az alkalmazásnak van adatbázis elérése, ha nincs, azt a nézeten a *"No database connection."* felirat jelzi. A nézet feladata a forráskód-párok listázása, és a párba állított forráskódok különbségeinek megjelenítése.

A különbségeket két forráskód között könnyen vizualizálhatjuk egy diffel, azaz olyan szövegösszehasonlító programmal, ami a szövegek közti a különbségek listáját állítja elő. A különbségeket a forráskód-párokban ezzel a módszerrel vizualizálom.



2.7. ábra. Adatbázis-böngésző nézet

A nézet tetején találhatóak a diffeket megjelenítő szövegdobozok, ezek alatt egy táblázat látható, soraiban az adatbázisba bekerült forráskód-párokkal. A táblázat soraiban található adatok sémáját a 2.1. táblázat írja le.

Oszlop	**Magyarázat**
id	az eredeti forrráskód azonosítóját tartalmazó oszlop
method	az átalakításra használt módszerre utaló oszlop (például a szabályhalmazra)
source	az eredeti forráskód sorainak számát tartalmazó oszlop
result	az átalakított forráskód sorainak számát tartalmazó oszlop
rules	az átalakításnál alkalmazott szabályok listája (szöveg)

2.1. táblázat. Adatbázis-böngésző nézet táblázatának oszlopai

Ha a táblázat egy sorára, vagyis egy forráskód-párra klikkelünk, akkor a diff nézetben megjelennek az eredeti (bal oldali) és az átakított (jobb oldali) kód közti különbségek.

A forráskód-párok böngészéséhez a táblázat feletti keresőt használathatjuk. A kereső az összes sorban és oszlopban szereplő adatok közt keres. Például megkereshetjük, hogy az adatbázisban mely forráskódokon kerültek alkalmazásra for ciklussal kapcsolatos átalakítások.

3. fejezet

Fejlesztői dokumentáció

Ebben a fejezetben az általam létrehozott szoftver működését mutatom be. A szoftver felépítését, a felhasznált tervezési mintákat és a fontosabb algoritmusok működését is bemutatom. A szoftver forráskódja a jövőben változhat, a frissített API dokumentáció ezért a személyes oldalamon is elérhető [3].

3.1. Csomagok

A szoftver forráskódja több csomagban és modulban található. Minden csomag a szoftver egy jól elkülöníthető rétegét valósítja meg, például az alkalmazási réteget vagy az átalakításokért felelő réteget.

A forráskód öt csomagba van szerveze, ezek az alábbi táblázatban láthatóak.

Csomag	**Rövid leírás**
transformations	átalakítások forráskódja és API az átalakításokhoz
client	adatbázis kliens
model	adatok modellezése és mentése
app	GUI alkalmazás csomagja
tests	egység és egyéb tesztek

3.1. táblázat. A szoftver fő csomagjai

Az szoftver "futtatható" fájljai és fontosabb függvényei a csomagokon kívül, külön modulokban helyezkednek el. A modulokról a 3.6. szekcióban olvashatunk.

3.2. A *transformations* csomag

Ebben a szekcióban ismertetem a *transformations* csomagot. A szekcióban az átalakításokhoz használt *ast* modult és az általam implementált átalakítások működését részletezem.

3.2.1. Az *ast* modul

Az átalakításokat a Python *ast* moduljával valósítottam meg. Az *ast* modul a Python standard könyvárában alapból megtalálható, célja az AST létrehozása a Python absztrakt-nyelvtanának alapján.

A Python absztrakt-nyelvtana a nyelv szintaxisát definiáló környezetfüggetlen nyelvtan, leírását az *ast* modul dokumentációjának [1] elején találhatjuk.

Az absztrakt-nyelvtan leírásában az AST csúcsainak (node-jainak) típusát az *ast.AST*-ből származó osztályok határozzák meg, ezekről szintén a dokumentáció elején, a *Node classes* cím alatt olvashatunk. Egy AST-re tehát általánosan az *ast.AST* osztállyal lehet hivatkozni.

Egy Python forráskód AST-jét az *ast.parse* függvénnyel hozhatjuk létre. Az *ast.parse* ekvivalens a Python beépített *compile* függvényének `ast.PyCF_ONLY_AST` compiler flaggel való meghívásával.

Fontos, hogy AST-t csak olyan forráskódból tudunk létrehozni, amiben nincs szintaxis hiba. A szintaxis hibákat az *ast.parse* a *SyntaxError* exception-nel jelzi.

Az *ast* modulban AST-k bejárására és átalakítására alkalmas segédfüggvények és segédosztályok is találhatóak. Az általam megvalósított átalakításokban az AST bejárását a *NodeVisitor* osztállyal, az AST átalakítását pedig a *NodeTransformer* osztállyal végzem.

Az átalakított AST-ből a kód generálására a *ast.unparse* függvényt használom.

A *NodeVisitor* osztály

Az AST-ket a *NodeVisitor* osztályból származtatott osztályokkal járhatjuk be.

A *NodeVisitor*-ból származtatott osztályokkal olyan, AST-vel kapcsolatos, lekérdezéseket is reprezentálhatunk, amelyek elvégzéséhez az AST bejárása szükséges.

Például egy AST-ben, az adott id-vel rendelkező, *Name* node-ok listáját a 3.1. forráskódon látható *NameVisitor* osztály *get_names* metódusával kaphatjuk meg.

```
1  class NameVisitor(NodeVisitor):
2
3    def get_names(self, root: AST, id: str) -> list:
4      self._id = id
5      self._names = []
6      self.visit(root)
7      return self._names
8
9    def visit_Name(self, node: ast.Name):
10     if self._id == node.id:
11       self._names.append(node)
```

3.1. forráskód. A *NameVisitor* osztály kódja

A *get_names* metódus paraméterei az AST (*root*) és a keresett id (*id*). Először a metódus inicializálja az *_id* és *_names* példány szintű változókat, majd elindítja a bejárást a *root*-ra. Bejárás után visszaadja a *_names*-ben összegyűjtött *Name* node-ok listáját.

A bejárást a *NodeVisitor* osztályból örökölt *visit* metódus meghívásával lehet elindítani egy AST-re. Ez a bejárás **mélységi bejárás**.

Bejárás közben a *Name* node-okat a *NodeVisitor* osztály *visit_Name* metdódusának felülírásával látogatjuk meg. A meglátogatott *Name* node-ot akkor adjuk a listához, ha az id-je egyezik az *_id*-vel.

Ehhez a példához hasonlóan a *NodeVisitor*-ban az összes node típushoz létezik *visit_<node-class>* visitor metódus amit felülírhatunk.

Fontos megjegyezni, hogy amikor felülírjuk a visitor metódust olyan node-típusoknál amik tartalmazhatnak magukkal megegyező típusú node-ot (például *For* node), akkor az összes node meglátogatásához szükséges a *generic_visit* hívása a felülírt metódusban, különben a rekurzió megáll.

NodeVisitor-ból származtatott osztállyal akkor érdemes bejárást végezni, ha mélységi bejárásra van szükségünk a *NodeVisitor* osztály működése miatt.

Amikor egy *NodeVisitor*-ból származatott osztályban a bejárás rekurzív hívása történik, minden gyerekre a gyerek típusához tartozó visitor metódus kerül meghívásra, ha az létezik. Tehát, ha a gyerek típusának visitor metódusát felülírtuk, akkor a gyereket azzal fogja meglátogatni. Ha nem írtuk felül, akkor a *NodeVisitor* osztály *generic_visit* metódusa a node-okat mélységi sorrendben látogatja meg. Ezért folytat a *NodeVisitor* alapból mélységi bejárást.

A *generic_visit*-et is felülírhatjuk. Erre a 3.2. forrráskódban láthatunk példát.

```
1  class TypeVisitor(NodeVisitor):
2
3    def get_nodes(self, root: AST,
4      node_matcher: Callable[[AST], bool]
5    ) -> list:
6      self._node_matcher = node_matcher
7      self._nodes = []
8      self.visit(root)
9      return self._nodes
10
11   def generic_visit(self, node: AST):
12     super().generic_visit(node)
13     if self._node_matcher(node):
14       self._nodes.append(node)
```

3.2. forráskód. A *TypeVisitor* osztály kódja

A 3.2. forráskódban definiált *TypeVisitor* osztály a 3.1. forráskódban definiált *NameVisitor* osztály általánosítása.

A *TypeVisitor* a bejárt AST azon a node-jait adja vissza, amikre a paraméterül kapott *node_matcher*, $AST \rightarrow bool$ típusú, függvény igaz értéket adott vissza. Ez a függvény egy node valamilyen tulajdonságát határozza meg, vagyis több típusú node-ra is működhet. Ezért kell a *generic_visit*-et használni, amivel az összes AST-ben található node-ot meglátogathatjuk.

A *generic_visit* felüírásával természetesen szélességi bejárást is definiálhatunk, viszont szélességi bejárásnál átláthatóbb, ha iteratív megoldást adunk.

A *NodeTransformer* osztály

Az AST-k átalakítása a *NodeTransformer* osztály segítségével valósítható meg. Ez az osztály kimondottan erre a célra használandó.

A *NodeVisitor*-hoz hasonlóan a *NodeTransformer*-ből is származtatni kell az osztályokat. Mivel a *NodeTransformer* maga is a *NodeVisitor* osztályból származik, a működése nagyon hasonló.

A bejárás szinte ugyanúgy működik, mint a *NodeVisitor*-ban. Az AST-t itt is a *generic_visit* vagy *visit_<node-class>* metódusokkal járhatjuk be. A különbség az, hogy az éppen meglátogatott node-ot a *generic_visit* vagy *visit_<node-class>* metódusok visszatérési értékével lehet változtatni.

Ha a bejárás közben a node-ot meglátogató függvény visszatérési értéke `None`, akkor a meglátogatott node törlődik az AST-ből.

Ha a node-ot meglátogató függvény visszatérési értéke nem `None`, hanem egy *ast.AST* típusú node, akkor a meglátogatott node a visszaadott node-ra változik az AST-ben (ha a node-on nem akarunk változtatni akkor változatlanul visszaadjuk).

3.2.2. Az átalakítások működése

Egy átalakítás, működése (magas szinten) a következő lépésekből áll:

1. AST létrehozása egy kódból
2. AST bejárása és elemzése
3. AST bejárása és átalakítása
4. kód generálása az AST-ből

Az átalakításokat végző algoritmusok a *transformers*, *transformers_rename* és *visitors* modulokban találhatóak, ezek a modulok tartalmazzák az AST-t elemző és változtató osztályokat is.

Az általam definiált átalakítások közül egyedül a *transformers_rename* modulban található átnevezéses átalakítások nem használnak szabályokat. Minden más átalakítás szabályokra épít.

3.2.3. Szabályok

Egy szabály az AST egy node-ján vagyis részfáján alkalmazható. Alkalmazásának két lépése van, amit két, AST-n értelmezett, függvény reprezentál:

1. mintaillesztés az adott node-on:

   $match(node : AST) -> Any|None$

2. eredmény generálása és visszaadása:

   $change(node : AST) -> Any|None$

A két lépésnek megfelelő metódusokat a szabályok absztrakt típusát definiáló *transformations.rule.Rule* osztály deklarálja.

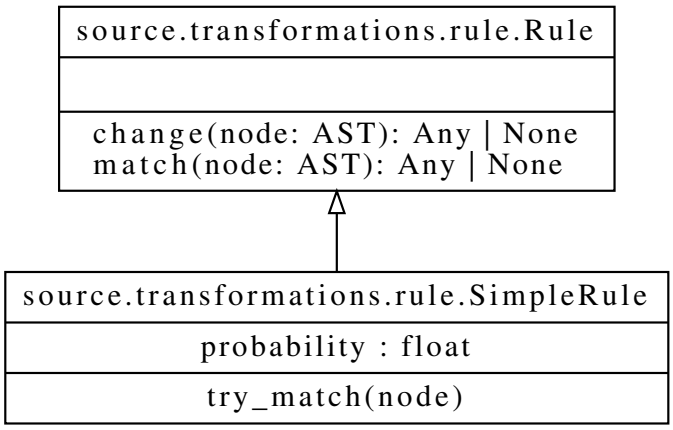
Minden szabálynak a *Rule* osztályból kell származnia és implementálnia kell az abban deklarált két absztrakt metódust.

Egy szabály önmagában nem képes egy egész AST átalakítására. Az átalakítást a *NodeTransformer*-ből származó *RuleTransformer* átalakító osztályok végzik egy szabály segítségével a *transform_ast* metódusban.

A következő alcímek alatt a különböző szabály típusokról és az azokat alkalmazó átalakító osztályokról olvashatunk.

3.2.4. *In-place* szabályok

Az *In-place* szabályok a *SimpleRule* absztrakt osztályból származnak, az osztály UML diagramját a 3.1. ábrán láthatjuk.



3.1. ábra. A *SimpleRule* osztály

Az *In-place* szabályok olyan szabályok, amik az AST egy node-ját alakítják át, ha azon a node-on sikeresen mintaillesztettek, vagyis a node-ot helyben változtatják.

Ezek az általam implementált legegyszerűbb szabályok. Az *Rule*-tól örökölt *match* metódust definiálva mintaillesztenek egy node-ra, a mintaillesztés sikerességének jelentése az *In-place* szabályoknál szabályfüggő.

Egyes szabályokban (pl. az *InvertIf* szabályban) csak mintaillleszteni kell a node-ra, ezeknél a szabályoknál a *match* visszatérési értéke *bool* típusú és a mintaillesztés sikerességére utal.

Más szabályokban (pl. a *GuardDef* szabályban) megkönnyíti a dolgunkat, ha a node-ot mintaillesztés közben "dekonstruáljuk", azaz a node bizonyos attribútumait vagy gyerekeit elmentjük és visszaadjuk, hogy a *change* metódus ezeket fel tudja használni. Ezeknél a szabályoknál a *match* visszatérési értéke *None* típusú, ha a mintaillesztés sikertelen. Sikeres mintaillesztésnél az elmentett attribútumokat vagy gyerekeket adja vissza egy tuple-ben.

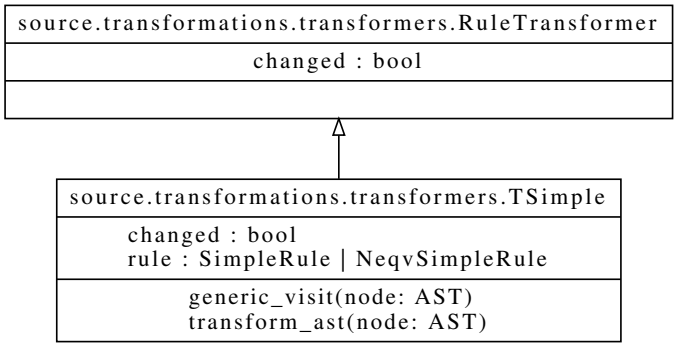
Ekvivalens és nem ekvivalens *In-place* szabályokat is definiáltam. Az ekvivalens szabályok a *rules_eqv.rules_simple* modulban, a nem ekvivalens szabályok pedig a *rules_neqv.rules_simple* modulban találhatóak.

Az *In-place* szabályokban a *change* metódus feladata a node átalakítása és az átalakított node visszaadása. A *change* metódus először mintailleszt a *try_match* meghívásával. A *try_match* visszatérési értéke alapján vagy átalakítja a node-ot, vagy egy *None* visszaadásával jelzi, hogy az adott node-ot nem tudja átalakítani.

Minden *SimpleRule* osztályból származó szabály rendelkezik egy valószínüséggel változóval (*probability* a 3.1. ábrán). Erre azért van szükség, mert a nagyon egyszerű szabályokat nem biztos, hogy minden node-ra alkalmazni szeretnénk. Ennek érdekében egy *In-place* szabály létrehozásakor megadhatunk egy 0 és 1 közötti valószínűséget, ami a szabály alkalmazásának valószínűsége.

A valószínűség implementációja miatt van szükség a *try_match* definiálására is. Ez a metódus először mintailleszt a node-on. Ha a mintaillesztés sikertelen, akkor leáll. Ha a mintaillesztés sikeres, akkor generál egy 0 és 1 közötti számot, amivel a valószínűséget szimulálja. Ha a generált szám kisebb mint a szabály valószínűsége, akkor a mintaillesztést sikeresnek tekintjük, különben sikertelennek.

Az *In-place* szabályokat a *RuleTransformer* osztályból származó *TSimple* osztály alkalmazza. A *TSimple* osztály UML diagramját a 3.2. ábrán láthatjuk.



3.2. ábra. A *TSimple* átalkító osztály

A *TSimple* működése egyszerű, a *generic_visit*-et felülírva mélységi bejárással meglátogatja az AST node-jait. Minden meglátogatott node-ra alkalmazza a *rule* példány szintű változóban található szabályt. Ha a szabály alkalmazható akkor a node-ot átalakítja, különben változatlanul hagyja. A bejárást a *transform_ast* indítja. A bejárás végén az eredmény a *changed* bool-ban található, ami azt jelzi, hogy sikerült-e a szabályt legalább egyszer alkalmazni az AST-n.

Az *In-place* szabályok listáját a 3.2. és 3.3. táblázatokban láthatjuk. [1]

Ekvivalens in-place szabályok		
Szabály	**Magyarázat**	**p**
MultipleAssign	Többszörös értékadást több értékadássá alakító szabály	1.0
TupleAssign	Tuple értékadást több értékadássá alakító szabály	1.0
GuardDef	Függvény else ágbeli return utasítását early return utasítássá alakító szabály	1.0
CommutativeMult	A szorzások jobb és bal operandusát felcserélő szabály	0.5
SingleIf	if utasításon belül található if utasítás feltételének hozzáadása a külső if feltételéhez	1.0
InvertIf	if utasítást if és else ágait megfordító szabály	1.0
DeMorgan	De Morgan azonosságokat alkalmazó szabály	1.0

[1] **p** a szabályok alap valószínűsége

Szabály	**Magyarázat**	**p**
DoubleNegation	Dupla negációkat eltütető szabály	1.0
BetterNegation	Negációkat bevivő szabály	1.0
ShuffleKeywords	Keyword argumentumokat összekeverő szabály	0.66
InsertContinue	continue utasítás beszúrása a ciklus végére	0.11
InsertPass	pass utasítás beszúrása egy függvény definícióba	0.11

3.2. táblázat. Ekvivalens *In-place* szabályok táblázata

Nem ekvivalens in-place szabályok		
Szabály	**Magyarázat**	**p**
NeqvNegateIf	if feltételének negálása az ágak megfordítása nélkül	0.5
NeqvInvertIf	if ágainak megfordítása a feltétel negálása nélkül	0.5
NeqvInsertBreak	break utasítás beszúrása ciklus elejére	0.33
NeqvInsertContinue	continue utasítás beszúrása ciklus elejére	0.33
NeqvInsertReturn	return utasítás beszúrása ciklus elejére	0.33
NeqvSwapAndOr	and és or operátorok felcserélése	0.5
NeqvSwapAddSub	+ és - operátorok felcserélése	0.5
NeqvReplaceNums	Szám konstansokat átíró szabály	0.33

3.3. táblázat. Nem ekvivalens *In-place* szabályok táblázata

3.2.5. *For* szabályok

A *For* szabályok a *transformations* csomag legösszetettebb szabályai. Ezek a szabályok for ciklussal megadott alap programozási tételek megoldásából (pl. összegzés vagy kiválogatás), Python specifikus, comprehension kifejezést használó megoldásokat állítanak elő.

A comprehension egy speciális kifejezés a Python absztrakt nyelvtanában, ami egy iterálható kifejezéstből (iter), egy target-kifejezésből (target) és "if" kifejezések listájából (ifs) áll.

Az "if" kifejezések a nyelvtan definíciójában kifejezések nem if statement-ek. Comprehension kifejezést használhatunk alap mutable adatstruktúrák (list, dict, set) vagy generátorok meghatározására.

Egy *For* átalakítás szemléltetéséhez tekintsük a következő feladat példáját:

1. *Példa.* Adjuk meg az $xs : Iterable$, azon elemeit egy listában, amikre a *condition* függvény teljesül (feltéve, hogy $bool(condition(x))$ az xs minden x elemére értelmes).

Az egyik megoldás, ha létrehozzuk egy üres listát, majd egy for ciklussal iterálunk az xs-en és a listához adjuk azokat az elemeket amelyekre a *condition* függvény "truthy" értéket ad vissza.

A másik megoldás, ha a listát egy list comprehensionnel adjuk meg, ekkor a for ciklus if feltételét a list comprehension végére tesszük.

```
result = []
for x in xs:
  if condition(x):
    result.append(x)
```

3.3. forráskód. Példa megoldása for ciklussal

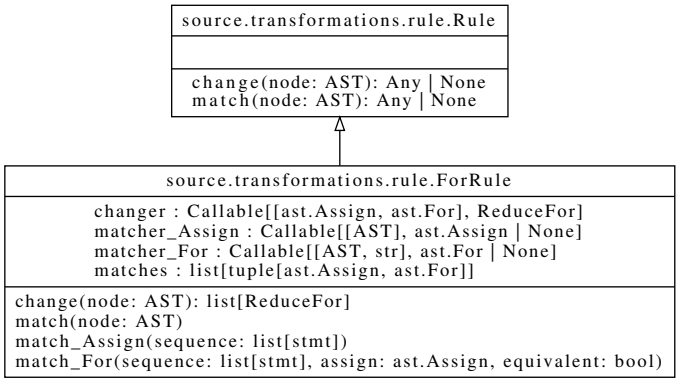
```
result = [
  x for x in xs
  if condition(x)
]
```

3.4. forráskód. Példa megoldása comprehensionnel

Pythonban a legtöbb esetben a második megoldás a preferált az elsővel szemben, rövidebb és általában olvashatóbb is.

A *For* szabályok célja a példához hasonló átalakítások megvalósítása. Ekvivalens és nem ekvivalens *For* szabályokat is implementáltam. A szabályok egy értékadást és egy for ciklust alakítanak comprehensionös kifejezéssé. A szabályok közt vannak list, dict és set comprehensionné alakító szabályok, illetve számok össszegének és feltételes összegének kiszámítására is van egy szabály.

A *For* szabályok a sablonfüggvény (*template method*) tervezési minta segítségével működnek. A mintaillesztést a *ForRule* osztály *match* metódusa végzi. A *match* metódus a sablonfüggvény, a sablonfüggvény lépései pedig a *match_Assign* és *match_For* metódusok.



3.3. ábra. A *ForRule* osztály

A 3.3. ábrán látható *matcher_Assign* és *matcher_For* osztályszintű változók függvények. Ezeket a függvényeket használom a *match_Assign* és *match_For*-ban az *Assign* és *For* node-ok mintaillesztésére.

A mintaillesztésénél a statement listát tartalmazó node-okra kell mintaillleszteni, ezek a Python absztrakt-szintaxisának megfelelően azok a node-ok, amik *body*, *orelse* vagy *finalbody* attribútumokat tartalmaznak.

Az ilyen node-okat a három `match` utasítással ismerehetjük fel, amik a következő esetekre mintaillesztenek:

`AST(body=[_, _, *_])`, `AST(orelse=[_, _, *_])`, `AST(finalbody=[_, _, *_])`.

Ha a node-ban van statementek listáját tartalmazó attribútum, akkor arra az attribútumra meghívjuk a *match_Assign* metódust.

A *match_Assign* először a statementek listájában szereplő node-okra meghívja a *matcher_Assign* függvényt. Ha a *matcher_Assign* sikeresen mintailleszt a node-ra, akkor a listában az utána szereplő statementeken kell mintaillleszteni a *matcher_For* segítségével. Ha a *matcher_For* is sikeresen mintaillesztett akkor az átalakítást el lehet végezni a felismert *Assign* és *For* node-okon. Ilyenkor az átalakítást a felismert *Assign* és *For* párokat tartalmazó *matches* listához adjuk.

A *For* átalakításokat felismerésének pszeudokódját a 1. algoritmuson láthatjuk.

1. algoritmus A *For* átalakítások felismerésének algoritmusa

method ForRule::match_Assign(*statements* : *list*[*stmt*])

```
1: n := length(statements) − 1
2: for i in 0 . . . n do
3:     assign := self.matcher_Assign(statements[i])
4:     if assign is not None then
5:         self.match_For(statements[i + 1 : n], assign)
6:     end if
7: end for
```

method ForRule::match_For(*statements* : *list*[*stmt*], *assign* : *Assign*)

```
1: name := assign.target.id
2: for statement in satements do
3:     matched := self.matcher_For(statement, name)
4:     if matched is not None then
5:         self.matches.append(matched)
6:         return
7:     end if
8:     if set(ids(statement)) ∩ set(ids(assign)) ≠ ∅ then
9:         return
10:    end if
11: end for
```

Ha az 1. algoritmus lefutott, akkor a megfelelő *Assign* és *For* node párokat a szabály *matches* listájában találhatjuk meg.

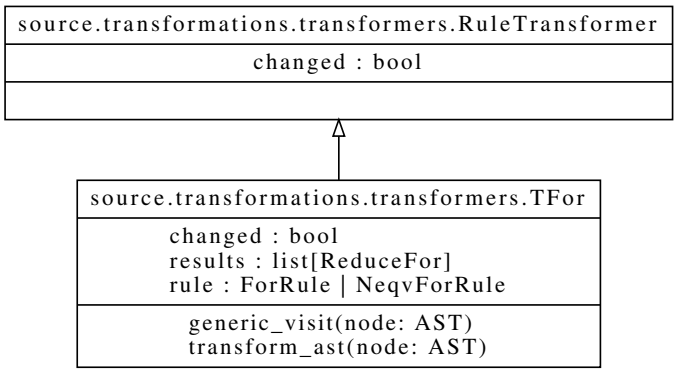
A *For* szabályok *change* függvénye először előállítja a *matches* listát. Ha a *matches* nem üres akkor az elemeiből létrehozza és visszaadja a változtatásokat egy listában ami *ReduceFor* példányokat tartalmaz (lásd *change* metódus a 3.3. ábrán).

A *ReduceFor* osztály egy példánya tartalmazza a törlendő *Assign* node-ot, a generált comprehension node-ot és a *For* node-ot, amit helyettesíteni kell a generált node-al.

A *For* szabályokat a *TFor* átalakító osztály segítségével lehet alkalmazni. Az átalakítások alkalmazásához az átalakító osztálynak kétszer kell bejárnia az AST-t.

Az első bejárással összegyűjtük a lehetséges változtatásokat az AST node-jain. A változásokat a *TFor* osztály *results* listájában tároljuk (lásd 3.4. ábra). A lista a szabály *change* metódusa által visszaadott *ReduceFor* példányokkal bővül.

Az első bejárás után a lehetséges változtatások már a *results* listában vannak. Az AST-t újból bejárva a változtatásokat elvégezzük. A megjelölt *Assign* node-okat töröljük és megjelölt *For* node-okat a generált kifejezésre változtatjuk.



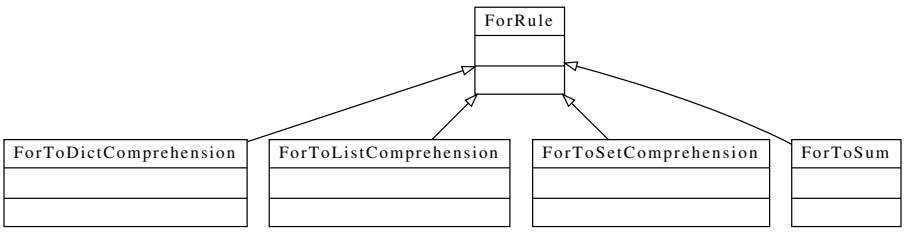
3.4. ábra. A *TFor* átalakító osztály

A *TFor* osztály *transform_ ast* metódusa kétszer hívja meg a *visit*-et a paraméterül kapott node-ra. A bejárást az ozstály a *generic_ visit* felülírásával végzi.

A *transformations* csomagban négy féle *For* szabályt implementáltam. Az első három szabály az üres dict, list vagy set -et inicializáló for ciklusokat, alakítja dict, list vagy set comprehensionné (a feltételes esetet is vizsgálva).

A negyedik szabály a nullával inicializált, for ciklus által számolt, összeget vagy feltételes összeget alakítja egy *sum* builtin függvény hívássá.

Mind a négy fajta *For* szabálynak van ekvivalens és nem ekvivalens változata is. A különböző szabályok fajtáit a 3.5. ábrán látható osztálydiagrammon láthatjuk.



3.5. ábra. A *For* szabályok UML diagramja

3.2.6. API az átalakítások alkalmazásához

Az átalakításokat a *transformation* modul segítségével alkalmazhatjuk. A modul célja egy olyan API létrehozása amivel könnyen átalakíthatunk kódokat vagy kódok AST-jét. Az API implementálásához az absztrakt gyár (*Abstract factory*) és az építő (*Builder*) tervezési mintákat használtam fel.

Ahogy azt az előző fejezetekben részleteztem egy szabály alkalmazásához két objektumot kell létrehozni: a szabályt és az annak megfelelő átalakító osztályt. Az absztrakt gyár az átalakító osztályok és szabályok példányosításában segít.

Az absztrakt gyár mintát a *create_rule* függvény implementálja. A függvény a szabálytípusok átalakító osztályait állítja párba a szabályokkal. Egy szabálynév alapján a *create_rule* létrehozza a szabálynak megfelelő átalakító osztály példányát. Ha a megadott nevű szabály nem létezik, akkor *ValueError* exceptiont dob.

Egy AST-t az átalakítás során gyakran több átalakító osztállyal is be szeretnénk járni. Például ha két szabályt szeretnénk alkalmazni, akkor az AST-t kétszer kell bejárni. A többszörös bejárások elvégzésében az építő tervezési mintát megvalósító *TransformationBuilder* osztály segít.

A *TransformationBuilder* osztály segítségével létrehozhatjuk átalakítók listáját, amivel egy AST-n több egymás utáni átalakítást (például szabályokat) hajthatunk végre. Ehhez definiáltam a *Transformer* interfészt, amit minden átalakítónak implementálnia kell. A *Transformer* interfészben csak egy metódus található, a *transform_ast*, ami az AST bejárását és átalakítását végzi. Ezt a metódust például a szabályok átalakító osztályai is implementálják.

Az is gyakran előfordul, hogy nem az eredeti AST-t akarjuk átalakítani, hanem annak egy másolatát, erre a célra a *CopyTransformer* osztályt használhatjuk.

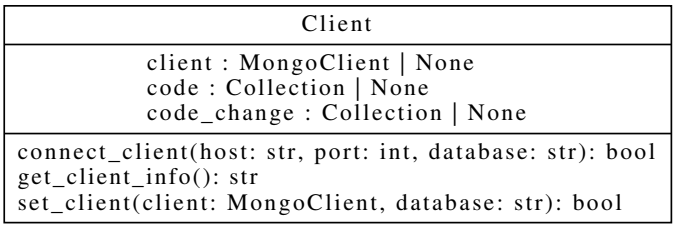
A *CopyTransformer* osztályt egy AST-t alapján példányosíthatjuk. A megadott AST-t a *CopyTransformer* lemásolja a *deepcopy* segítségével. Az átalakításokat már a másolaton végzi a egy *TransformationBuilder*-t felhasználva.

3.3. A *client* csomag

A *client* csomag feladata az adatbáziskapcsolat és az indexek létrehozása. A kliens a *pymongo* könyvtárat használja, implementációja a *Client* osztályban van. Ha szoftvernek az adatbázisra van szüksége azt ezen az osztályon keresztül érheti el. Például az adatbázis forráskódokat és forráskód-párokat tartalmazó kollekciói a kliensen keresztül elérhetők.

A *Client* osztály a Python-ban gyakori *monostate* [4] tervezési mintát használja, a minta a *singleton*-hoz hasonló, de több példány létrehozását is megengedi. Egy *monostate* osztálynak van egy belső (statikus) állapota, példányosításnál ezt a belső állapotot adja vissza. Ez hasznos, mert a példányok egy közös állapoton osztoznak, ami a program több részéről is elérhető.

Python-ban az objektumok állapota reprezentálható egy dict segítségével, ezért a *monostate* mintát nagyon egyszerű implementálni: a belső állapot egy dict lesz, példányosításnál a belső állapot dict-je alapján létrehozunk egy objektumot.



3.6. ábra. A *Client* osztály UML diagramja

Amikor először példányosítunk a *Client*-ből a *client*, *code* és *code_change* attribútumai *None* értékeket vesznek fel (ez a kezdelteges belső állapot).

Ha ezután meghívjuk a *connect_client* metódust az adatbázis paramétereivel és a kapcsolat 10 másodpercen belül létrejön, akkor a kapcsolat sikeres. Ekkor a *client* attribútum az adatbáziskliens, a *code* és *code_change* attribútumok pedig rendre a kódokat és kód-párokat tartalmazó kollekciók lesznek.

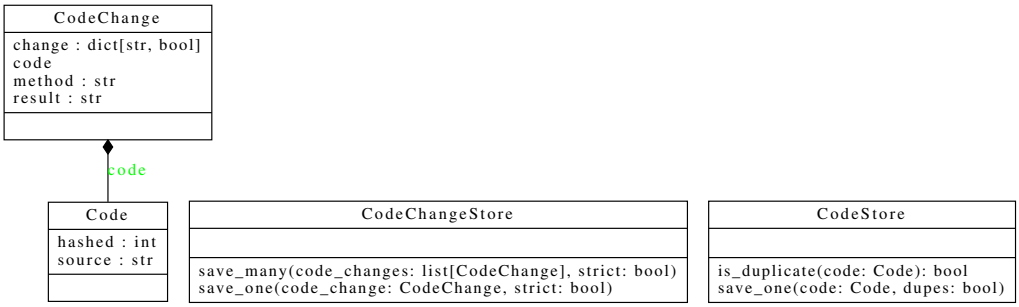
A kollekciók akkor is létrejönnek a kliens szintjén ha az adatbázisban még nem szerepelnek. Ebben az esetben a kollekció az adatbázisban akkor jön létre ha a kliensen keresztül elmentünk egy dokumentumot. A kliens a szükséges indexeket is definiálja.

3.4. A *model* csomag

A *model* csomag a feladata a forráskód-párok modellezése, három modulból áll:

- *datatypes* - az adattípusokat definiáló modul
- *serializers* - az adattípusokat szerializáló modul
- *stores* - az adattípusokat elmentő modul

A csomagban két adattípust definiálok a *Code* és *CodeChange*. A *Code* a forráskódok modellje, a *CodeChange* pedig a forrráskód-párokat modellezi.



3.7. ábra. A model csomag osztályainak UML diagramjai

A forráskód a duplikátumok kiszűrése miatt rendelkezik saját modellel. A duplikált forráskódok szűrése azért szükséges, mert GitHub-on a kódok jelentős része duplikátum [5], vagyis ha GitHub-ról bányászunk kódokat akkor a generált adathalmaz minőségén javíthatunk, a duplikátumok kiszűrésével.

A kódok modellje ezért a kód mellett a kód *sha256*-os hash-ét is tárolja. Mielőtt a kódot elmentjük az adatbázisba megnézzük, hogy a hash ütközik-e. A hash attribútum indexelve van a kódokat tartalmazó kollekcióban, ez biztosítja a gyors lekérdezést. Ha nincs ütközés, akkor a kódot egyből hozzáadhatjuk az adatbázishoz, ha vannak ütközések, akkor csak az ütköző kódokat kell összehasonlítani.

A duplikátumok kiszűrését a *CodeStore* osztály *save_one* metódusa végzi, a *CodeChangeStore* osztály segítségével pedig a kód-párokat tartalmazó kollekcióba szúrhatunk be dokumentumokat.

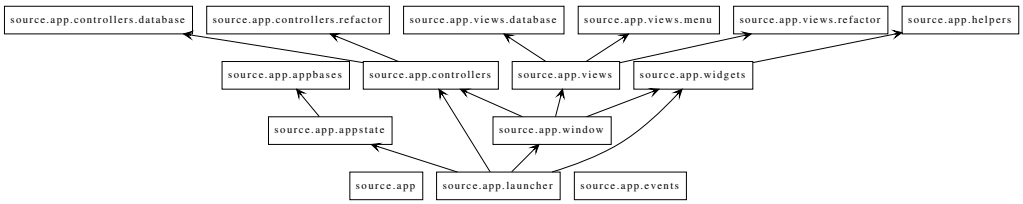
A *CodeStore.save_one* metódusában a duplikátumok szűrése opcionális, tehát duplikátumoktól mentes kódokból is generálhatunk adathamlazt ha szeretnénk.

3.5. Az *app* csomag

Az *app* csomag feladata az átalakításokat szemléltető GUI-s alkalmazás megvalósítása. Az alkalmazás architektúrája modell-nézet-kontroller (MVC) szerű. Egy nézet rendelkezik egy kontrollerrel és a kontroller pedig egy modellel.

A nézet feladata a GUI definiálása és frissítése, a modell feladata az adatelérés vagy az alkalmazás állapotának modellezése. A kontroller ezt a két réteget köti össze, így a nézet nem függ a modelltől és a modell sem a nézettől.

Az alkalmazás csomagjainak UML diagramját az alábbi ábrán láthatjuk.



3.8. ábra. Az alkalmazás csomagjai

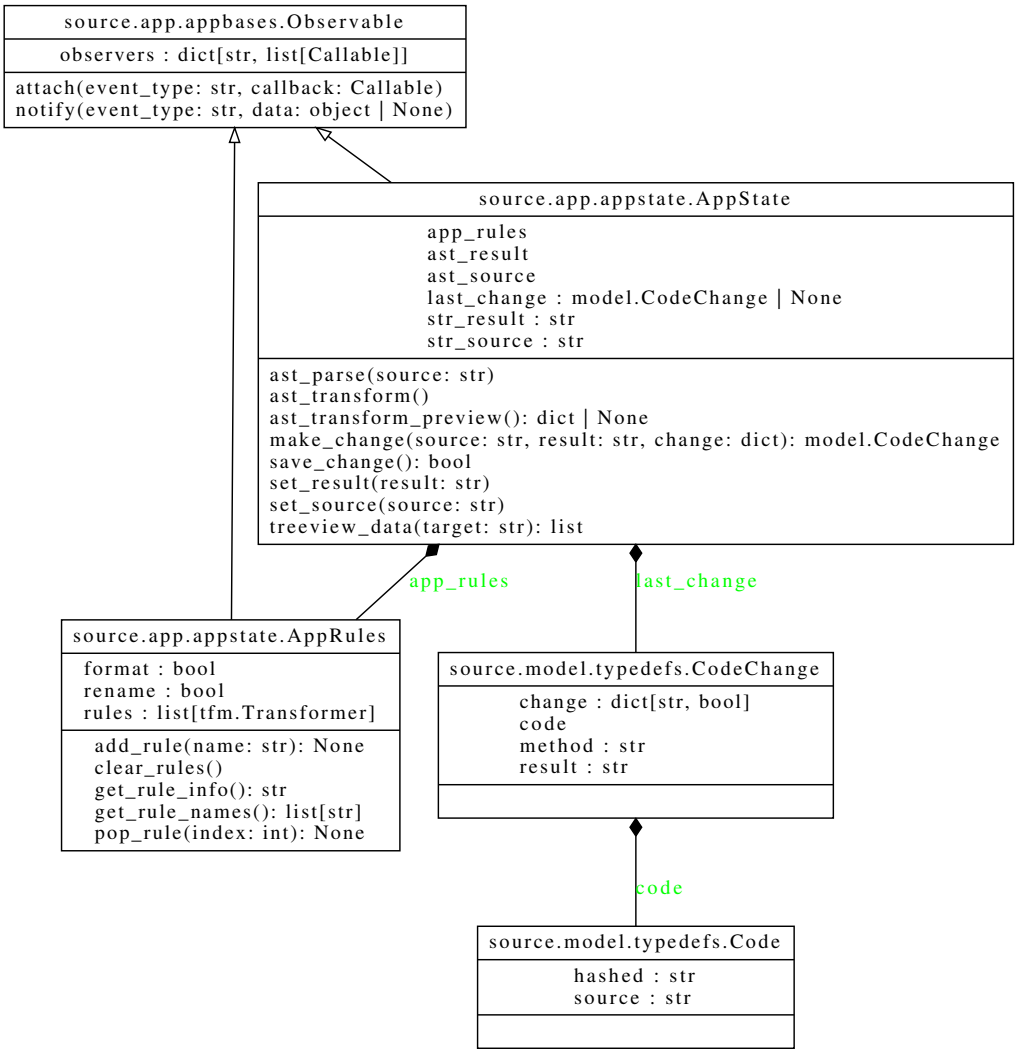
3.5.1. Állapotmodell

Az alkalmazásban kétféle modell különböztethető meg: az adatelérési modellek, amik működéséről az előző szekcióban olvashatunk és az alkalmzás állapotmodellje ami az *app.appstate* modulban található.

Az állapotmodell az *AppState* osztályban van definiálva, a megfigyelő (*observer*) tervezési mintát használja a nézetek frissítésére. Az *AppState* osztály az *Observable* osztályból származik, ezért rendelkezik megfigyelők (*observers*) egy listájával, ami esemény-eseménykezelő párok listája.

Ha a nézet valamelyik komponenesét az állapotmodell egy változásának hatására szeretnénk frissíteni, akkor azt az eseménykezelőt, ami frissíti, hozzárendelhetjük az állapotmodell egy eseményéhez az *app.events* modulból.

Hozzárendelni egy eseménykezelőt egy eseményhez az *Observable* osztály *attach* metódusával lehet. Ha az adott eseményt kiváltja egy változás a modellben, akkor a modell értesíti a nézeteket, vagyis az *observers*-ben az eseményéhez rendelt eseménykezelőket meghívja, a frissítéshez szükséges adatokat paraméterként továbbítva.



3.9. ábra. Állapotmodell és kapcsolódó osztályok UML diagramjai

Az alkalmazás egy állapotát a bemeneti és kimineti AST-k, az ezekből generált kódok, az átalakításért felelő szabályok listája, és az utolsó átalakítás írják le.

Az AST-k és a belőlük generált kódok az *AppState* osztály példány szintű változói. Ezeken kívül az állapotmodell az *AppRules* és a *CodeChange* osztályok egy-egy példányát is tartalmazza, ezek rendre az átalakításért felelő szabályokat és az utolsó átalakítást tárolják.

Az *AppRules* a szabályok állapotát modelezi, szintén az *Observable*-ből származik. Ez az osztály tartalmazza a kiválasztott szabályokat a *rules* listában, a *format* és *rename* bool-okkal pedig azt tárolja, hogy az átalakított kódot kell-e formatálni és az átnevezéseket végre kell-e hajtani.

Az utolsó validált átalakítást a *CodeChange* osztály egy példánya tárolja. Ahogy azt az előző szekcióban is említettem ezzel az osztállyal, lehet kimenteni az átalakítást az adatbázsiba.

Az *AppState* metódusai segítségével változtathatjuk a modellt. A metódusok az AST-k átalakítását és az utolsó átalakítás mentését valósítják meg, ezek a metódusok váltják ki a modellel kapcsolatos eseményeket is.

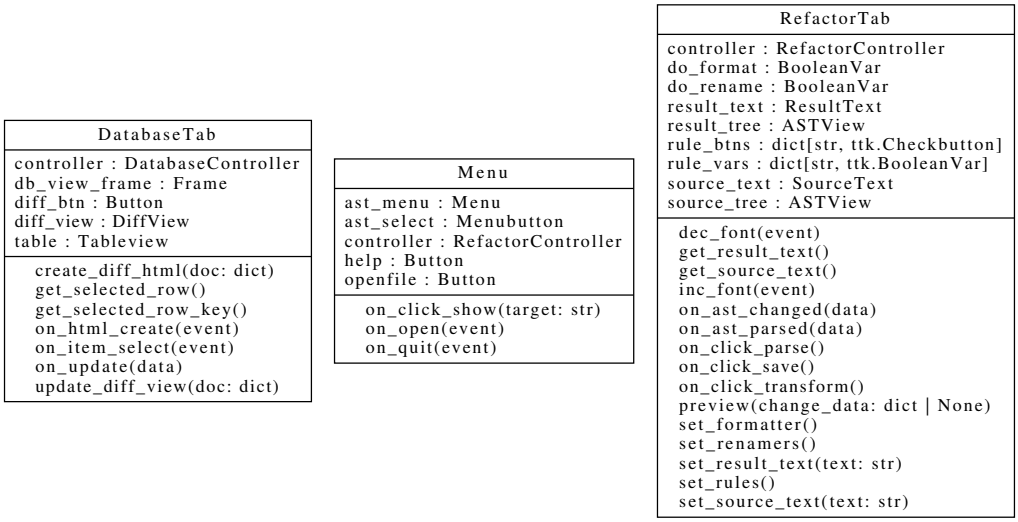
3.5.2. Nézetek

Az alkalmazás grafikus felhasználói felületének megvalósításához a Python-ban alapból megtalálható *tkinter* könyvtárt használom, amit az erre építő *ttkbootstrap* könyvtárral egészítek ki. A felhasználói felület forráskódja az *app.views* csomagban és az *app.widgets* modulban található.

A *tkinter* könyvtárban a GUI elemeket widgeteknek hívják. Az applikáció widgetei az *app.widgets* modulban vannak definiálja. Például az *app.widgets* modulban található a Python szintaxis kiemelését támogató szövegdoboz és az AST-ket ábrázoló fa-nézet definíciója is.

Az applikáció összetettebb nézeteit az *app.views* csomagban definiáltam. Ezek a nézetek szintén widgetek, de az *app.widgets* widgeteivel ellentétben rendelkeznek egy kontrollerel, amit a modellel való kommunikációhoz használnak (az *app.widgets* widgetei nem férnek hozzá a modellekhez).

Az *app.view* widgetei az *app.appbases.View* osztályból származnak. A *View* osztály egy *tkinter*-es frame, ami a nézethez tartozó kontroller egy példányával jön létre. Az alkalmazásban három ilyen kontrollerrel rendelkező nézet van, ezek osztálydiagrammjait a 3.10. ábrán láthatjuk.

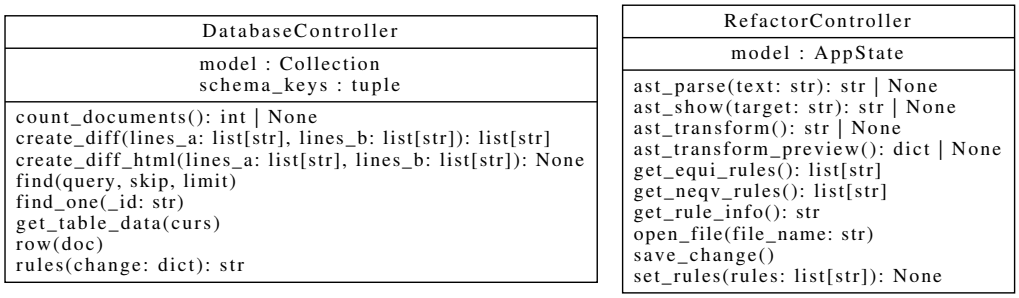


3.10. ábra. A nézetek osztálydiagrammjai

3.5.3. Kontrollerek

A kontrollerek feladata a kommunikáció a modellek és nézetek között. Ahogy azt a 3.10. ábrán láthatjuk csak az alkalmazás két fő nézete (*DatabaseTab* és *RefactorTab*) illetve a menü (*Menu*) rendelkeznek kontrollerrel.

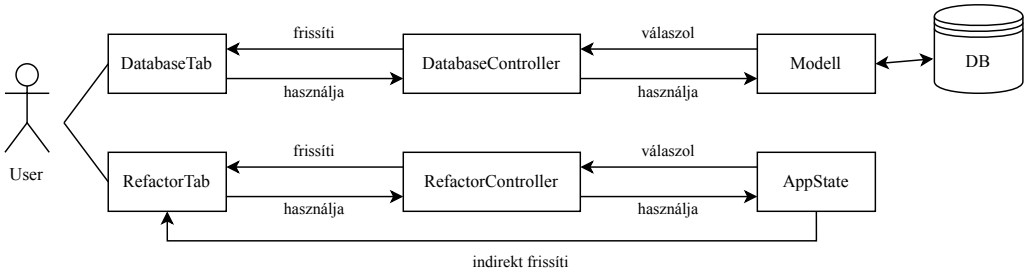
Az applikációban két különböző kontroller van. A *DatabaseController* az adatelérési modellekkel, a *RefactorController* az alkalmazás állapotmodelljével kommunikál.



3.11. ábra. A kontrollerek osztálydiagrammjai

Ha a nézeten olyan GUI esemény, történik aminek az eseménykezelője a modell vagy az állapotmodell használatát igényli, akkor az eseménykezelő a nézethez tartozó kontroller megfelelő metódusát hívja meg. Ha az eseményhez input is tartozik (pl. egy szöveges doboz tartalma) akkor azt is továbbítja a kontroller metódusának paraméterként.

A kontroller használja a modellt, lekérdezéseket vagy változtatásokat végez rajta, ha ezek megtörténtek a nézetet direk vagy indirekt módon frissíti. Direkt módon frissíti, ha a modelltől kapott adatokat a nézetnek továbbítja, ami azokkal frissül. Ha a kontroller egy eseményt vált ki a modellben, aminek hatására a nézet frissül, akkor indirekt frissíti. Ezt a működést az alábbi ábrán láthatjuk.



3.12. ábra. Egy esemény kezelése az alkalmazás architektúrájában

Az alkalmazásban csak a *RefactorController* végez indirekt frissítést a 3.5.1. alcím alatt részletezett *Observable* tervezési minta segítségével.

3.6. Modulok

A csomagok a *tests* csomagon kívül nem tartalmaznak futtatásra szánt fájlokat. A belépési pontok és segédfüggvények a 3.4. táblázatban látható modulokba vannak szervezve.

Modul	**Rövid leírás**
launch	GUI alkalmazás belépési pontjának modulja
persistor	CLI program modulja
tools	modul eszközök alkalmazására (pl. linterek)
utils	utility függvények modulja (pl. fájlok olvasásához)

3.4. táblázat. A szoftver fő moduljai

A GUI alkalmazást a *lanuch* modul main függvénye példányosítja. Az adathalmazt generáló program implementációja a *persistor* modulban található. A *tools* és *utils* modulokban pedig a forráskódban használt segédfüggvények definíciói találhatóak.

3.7. Tesztelés

TODO

4. fejezet

Összegzés

TODO

Köszönetnyilvánítás

TODO: Tamás köszönetnyilvánítás

A. függelék

Kiértékelés eredményei

TODO: kiértékelés eredményei

Irodalomjegyzék

[1] python docs. *ast - Abstract Syntax Trees*. URL: https://docs.python.org/3/library/ast.html.

[2] mongodb docs. *MongoDB Installation*. URL: https://www.mongodb.com/docs/manual/installation/.

[3] Verebics Péter. *API dokumentáció*. URL: http://szonyegxddd.web.elte.hu/szakdolgozat_api_doc/index.html.

[4] VasileAlaiba. *Monostate Pattern*. 2014. URL: https://wiki.c2.com/?MonostatePattern.

[5] „DéjàVu: a map of code duplicates on GitHub". *ACM Journal* (2017). URL: https://dl.acm.org/doi/10.1145/3133908.

Ábrák jegyzéke

Táblázatok jegyzéke

Forráskódjegyzék