

TDK-dolgozat

Csiszár Norbert István

Szegedi Tudományegyetem
Természettudományi és Informatikai Kar
Szeged, 2012

Programkód másolatokat elimináló refaktoring algoritmus megvalósítása.

Konzulensek:

Dr. Ferenc Rudolf adjunktus

Dr. Vidács László tudományos munkatárs

Tanszék:

Szoftverfejlesztés Tanszék

Készítette:

Csiszár Norbert István

Programtervező informatikus BSc

Szekció:

Szoftvertechnológia szekció

Tartalomjegyzék

1.	Bevezetés.....	4
1.1.	Problémák a Duplicate Code smell-el kapcsolatban.....	4
1.2.	Refaktoring	6
1.3.	Megoldás.....	7
2.	Kapcsolódó munkák	8
2.1.	Tudományos cikkek, közlemények.....	8
2.2.	Pár hasznos refaktoráló eszköz	11
3.	Elvi megközelítés	12
3.1.	Esetek, és kiküszöbölendő problémák	13
4.	Architektúra	14
5.	Algoritmus.....	16
5.1.	Szerkezeti felépítés	16
5.1.1.	A klón osztály beolvasása	17
5.1.2.	A megfelelő klónosztály kiválasztása.....	17
5.1.3.	A klón példányok analizálása, adatok begyűjtése	18
5.1.4.	A klónosztály több részre bontása	19
5.1.5.	Metódus helyének kijelölése	20
5.1.6.	A metódus felépítése	20
5.1.7.	A metódus visszatérési típusának kiválasztása.....	21
5.1.8.	A metódushoz a szükséges paraméterek hozzáadása	22
5.1.9.	A klón példányok átalakítása.....	22
5.2.	Egy egyszerű forráskód példa.....	24
5.3.	Egy komplexebb forráskód példa	26
5.4.	Kiértékelés	29
6.	Korlátozások.....	31
6.1.	A programom korlátai.....	31
6.2.	Esetek, amiket nem célszerű átalakítani	32
7.	Konklúzió	33
7.1.	Tervek további funkciókra.....	33
8.	Irodalomjegyzék	34

1. Bevezetés

A programok minősége manapság nagy szerepet kap az informatikában. Több szempontból is lehet vizsgálni, hogy az adott program minősége mennyire felel meg bizonyos kritériumoknak. Nem csak a funkcionális követelmények megoldásai számítanak, hanem az érthetőség, tagoltság, karbantarthatóság, illetve az objektum orientáltság kihasználtsága is. Ezen tulajdonságokat nehéz objektíven mérni. A széles körben elfogadott megoldás az, hogy a program alapvető tulajdonságait különböző szoftver metrikákkal (mérőszám [1, 24]) fejezzük ki, majd ezek alapján következtetünk a fenti, magasabb szintű tulajdonságokra. Például a metódusok, és adattagok tulajdonságai függhetnek a metódusok hosszától, változók és metódusok elnevezésétől, változó- és metódus-hivatkozásoktól, paraméterlista hosszától, programkód másolatok számától. A metrikáknak lehetnek olyan értékeik, melyek azt jelzik a programozónak, hogy valami nincs rendben a programkód implementálásával kapcsolatban, és bár a program funkcióját nem befolyásolják, mégis általában valamilyen mélyebb problémára utalnak. Ezeket a tulajdonságait a programnak nevezzük bad code smell-nek [2]. Ilyen smellek például a Long Method [3], amely a nagyon hosszú metódusokra jellemző, a Large Class, amely a hatalmas osztályokra jellemző, illetve a **Duplicate Code** [20], amely egy adott kódrészlet többszöri előfordulását jelenti.

Az én témaköröm az utóbbi problémára, az adott kódrészlet többszöri előfordulására keres megfelelő megoldást, egy olyan program megírásával, amely a kiválasztott kódrészleteket összeolvasztja, feltéve, hogy lehetséges. Azért ennek a smellnek a megszüntetését céloztam meg, mert míg a hosszú metódus csak programkód megértését nehezíti, a duplikált kódok komolyabb hibákat rejthetnek magukban, melyeket majd később tárgyalok.

1.1. Problémák a Duplicate Code smell-el kapcsolatban

Mint ahogy előzőleg láthattuk, ebben az esetben egy bizonyos kódrészlet több mint egy helyen van jelen a programunkban. A másolt és módosított program részek felfedezése és kezelése jelenleg is aktuális kutatási téma [21,22]. Hibák következhetnek be, ha a kérdéses kódrészlet átírásáról van szó, mivel egy változtatás során meg kell keresni az összes egyező kódrészletet, és mindet átírni. Nem beszélve arról, hogy kódunk méretét is megnövelik a kód-duplázások. Elég, ha egy duplázás következtében egy konstans értéket

nem írunk át a megfelelőre, vagy ha az eredeti kód, amelyet átmásoltunk hibás, így a hiba detektálása után az összes felhasznált helyen át kell írni a hibás utasítássorozatot.

Következőekben két példát fogok hozni, amely motivált, és amely demonstrálja a programkódok által okozott nehézségeket, és veszélyeket. A problémák nem csak ezen a két területen jelentkeznek a kódDuplikálások által, hiszen az informatika már szinte minden szakterületen jelen van. Azért ezt a két szakterületet emelném ki, mivel az első pénzügyi területen okoz problémákat, a másik pedig emberéleteket követelhet bizonyos esetekben.

Mint ahogy említettem, az informatika mára már nagyon szerteágazó szakma lett. Ez nincs máshogy a banki programok implementálásánál sem. Sokkal egyszerűbb egy programra bízni magát az embernek ilyen ügyekben, mint az emberi hibalehetőségnek kitennie magát, és programok nélkül megoldani egyes átutalásokat, pénzváltásokat, kifizetéseket. Azonban nem hagyhatjuk figyelmen kívül, hogy a banki rendszerek programjaiban is benne lehetnek emberi hibák, mivel a programokat emberek írják. Ha nem történnek meg a megfelelő tesztelések, akkor maradhatnak rejtett hibák a programban. Történt egy olyan eset, hogy a Szoftverfejlesztés tanszék által készített kódDuplikálásokat kereső, és kimutató szoftverét mutatták be egy banki szoftveren, amely a valuták átváltásáért volt felelős. Ekkor tűnt csak fel, hogy az egyes pénznemek közötti átváltást végző függvények mind egymás másolatai voltak. Ez alapjába véve nem probléma, hiszen ha az átmásolás során a kellő átalakítások, mint például a konstansok átírása, megtörténik, akkor a függvény a megfelelő feladatot látja el. Bár még így is nagy a veszély, hiszen ha valami olyan szerkezeti átírást kellene végrehajtanunk az összes másolaton, akkor nem lenne egyszerű dolgunk, ráadásul nőne a hibák eshetősége. Ebben az esetben a hiba, amit ki szeretnék emelni az volt, hogy az egyik másolásnál egy konstans érték nem lett átírva két valuta átváltásánál. Ez azt eredményezte, hogy vagy az ügyfél, vagy a bank veszteséget szenvedett minden egyes átváltásnál a két valuta között. Ráadásul ez egy olyan hiba, amely nem olyan könnyen detektálható, hisz egy kis értékváltozást is kevés észrevenni, ha nem nézünk bele a programkódba.

Egy másik terület, amiről szót szeretnék ejteni, az orvostudomány. Régen a vizsgálatokat olyan eszközökkel végezték az orvosok, amelyeknél nem volt szükség programokra. Akkoriban nem lehetett az emberi szervezetet teljes körűen átvizsgálni, és sokkal nehezebb volt az egyes betegségeket felfedezni. Manapság már a beavatkozások

egy részét is próbálják gépiesíteni, amely ismét az előbb említett emberi hibalehetőséget hivatott megakadályozni. Azonban ezekben a programokban nagyon pontos számokkal kell dolgozni, és már egy kis eltérés is sok bajt okozhat például egy lézeres szemműtétnél. A hibák számos okból bekövetkezhetnek, ezek között vannak a kódDuplikálások. Ha az adott másolatokat meg tudjuk szüntetni, és az értékeket jól megadni minden egyes használatnál, akkor elkerülhetőek lehetnek ezek a problémák, és egy jól működő, biztonságos rendszert adhat ki a programozó a kezei közül.

Sajnos ezek a problémák még valóságok, és megoldásra várnak. Tervem az, hogy a programozók dolgát megkönnyítve, és a hibák csökkentése érdekében megbízható program kódDuplikálásokat megszüntető programot készítek. Ehhez azonban szükséges, hogy az átstrukturálás olyan kódot eredményezzen, amely működésében megegyezik a bemeneti kód működésével. Nagyon fontos szempont továbbá, hogy könnyen olvasható, és értelmezhető kódot gyártsunk. Ha ezt a két szempontot tudjuk biztosítani a programozó számára, akkor elérhetjük, hogy megbízzon a másolatot megszüntető algoritmusban, és használja, amikor szükséges.

1.2. Refaktoring

A már meglévő programkód minőségét javító módszereket gyűjtő néven **refaktoring** [3, 7] módszereknek nevezzük. „A refaktoring egy technika egy már meglévő forráskód átstrukturálására, megváltoztatva saját belső struktúráját, a külső viselkedés megváltoztatása nélkül.” Ezen eljárások a fentebb említett kód minőségére hatással levő tulajdonságait küszöbölik ki a programnak. Sok módszer létezik, néhányat ezek közül be is mutatok. Ezeket az eljárásokat több csoportra lehet osztani. Ilyenek a metódusok átformálására szolgálók, az egyes osztályok közötti metódus, belső osztály, változó átmozgatások, adatokat egységesítő, és könnyen kezelhetővé tevő eljárások, a feltételek átláthatóságára szolgáló összevonások, továbbá a metódus-hívásokat egyszerűbbé tevő, végül pedig az osztályok öröklődési-hierarchiájával kapcsolatos mozgatók, és átszervezések. Minden egyes csoporthoz több kisebb módszer tartozik, és minden ilyen kisebb módszer egy-egy problémára van kialakítva. Azonban nagyon fontos szempont, hogy e módszerek alkalmazása csak a programkódra van hatással, viszont a program működésére nem szabad, hogy kihasson. Ezért nem árt, ha a minőség javítása közben, és utána is, a programot folyamatos tesztelésnek vetjük alá, hiszen könnyű a program futására

hatással levő változtatást a kódon végrehajtani. Ilyen változtatás lehet, ha például változó illetve függvény átnevezésekor nem nevezünk át egy hivatkozást, vagy névütközés történik, és ez okoz rendellenes működést. A refaktoringról átfogóbban **Mens és Tourwe** [11] tanulmányában olvashatunk.

Mivel nagy a hibalehetőség, és van esély a figyelmetlenségre, meg kellett fontolni, hogy lehet-e a kód javítását **automatizálni** [9, 10, 12, 13], az emberi hibalehetőséget kiküszöbölni. Viszont mielőtt az automatizáló programot megírnánk, szükségünk van segéd programokra, amelyek az aktuális programkódot más, könnyebben kezelhető formátumba alakítják át, illetve elemző programokra, melyek segítik a munkánkat, és felfedik az olyan részeket a kódban, melyeket refaktorálnunk kell, illetve a kódgeneráló programok, amelyek a refaktorált kódot tartalmazó gráfból újra előállítják a programot. A visszagenerálásnál ügyelni kell arra, hogy a felhasználó kódjának stílusán, behúzásain és tagolásán ne változtassunk. Nem minden refaktoring-ot lehet automatizálni.

A leggyakoribb automatizált refaktoring-ok a rename [3, 9], amely a metódus, változó, osztály, csomag, névtér átnevezése úgy, hogy minden hivatkozásnál is végrehajtja a kellő névmódosításokat, az Extract Method, amely a programkód tagoltságát segíti, melynél egy nagyobb metódus egy utasítássorozatot alakítjuk külön metódussá, majd az eredeti kódból kitörölve a metódussá átalakított utasításokat, meghívjuk az újonnan létrehozott metódust.

1.3. Megoldás

Ebben a dolgozatban a **Replace Method Code Duplicates** [3] eljárást járom körbe jobban, hiszen ez az eljárás alkalmas a programkód duplikálások megszüntetésére. Továbbiakban két fogalmat kell tisztázni: **klón példánynak** nevezzük az egyező programkód részleteket külön-külön. Ezek közül az egyező klón példányokat közös **klón osztályba** soroljuk.

Munkám során meghatároztam, hogy milyen feltételeknek kell teljesülni ahhoz, hogy egy klón osztály példányai átszervezhetőek legyenek. Az erre kidolgozott módszert implementálva olyan eszközt készítettem, mely egy adott klón osztály megszüntetését végzi el: a szükséges feltételek teljesülése esetén a klón példányokat egy új metódusba emeli ki a program viselkedésének megváltoztatása nélkül. Az eszközt egy valós, nyílt forráskódú rendszeren teszteltem biztató eredménnyel.

Ha megkapja a szükséges klónosztályok listáját, és a megszüntetni kívánt klónosztály azonosítóját, akkor megvizsgálva a példányokat, végrehajtja a klónpéldányok kiemelését a környezetükből, készít egy metódust, amelyet a klónok helyén meghívva elértem, hogy az átstrukturálás megtörténjen a funkció megtartása mellett. Az átalakítás után a forráskód egy külső programmal kinyerhető, és ellenőrizhető a kapott eredmény.

2. Kapcsolódó munkák

Ebben a témakörben kifejtem, hogy milyen, a refaktoringgal kapcsolatos cikkek, illetve munkák születtek. A munkákkal kapcsolatban nem korlátozódok a program klónok eliminálására, hiszen a refaktoring programot fejlesztőknek sok különböző módszer áll a rendelkezésükre, különböző bad smellek eltüntetésére. A jelenleg elérhető refaktoráló eszközök közül csak a legismertebbekkel foglalkozom, azok közül is inkább a Java nyelvű forráskódot átformálókkal. Minden eszköznél felsorolom, hogy mely refaktorálások érhetőek el rajtuk. Lesznek olyan funkciók, melyekkel az alább említett programok rendelkeznek, azonban nem sorolom fel az implementált refaktoringok között, mivel nem kapcsolódik a smellekkel kapcsolatos refaktoringhoz. Ilyenek például az a funkció, melynél egyik csomagból átemeljük a másikba az osztályunkat, illetve az összes olyan hivatkozást, mely erre az osztályra mutat, kicseréljük az átmozgatás utáni útvonalra.

2.1. Tudományos cikkek, közlemények

Martin Fowler sokat idézett könyvével népszerűsítette az akkor még gyerek cipőben járó módszert, a refaktorálást [7], mely munka 1999-es megjelenése óta a téma alapművének számít. A szintén hozzá köthető Refactoring honlapon megtalálható katalógus jó kiindulópont a téma iránt érdeklődőknek [23].

A gyökerek a 90-es évek elejére nyúlnak vissza, a téma egyik úttörő tanulmánya William F. Opdyke doktori disszertációja [8]. Motivációja, hogy nehéz újrahasználató, és könnyen fejleszthető programot írni. Megemlíti, hogy az objektum orientált programok átalakítása nehezebb munka, mint az elsőnek tűnik, hiszen ha mozgattunk osztályok között adattagokat, vagy megváltoztatjuk két osztály kapcsolatát, akkor az kihathat a program összes olyan részére, amely használja az adott osztályokat vagy az átmozgatott metódusokat, esetleges láthatósági problémákat is eredményezhet. Felveti az ötletet, hogy ezen átalakítás hibái kizárhatóak, ha automatizáljuk ezeket a műveleteket. Hangsúlyozza, hogy a program működésére nem szabad hatással lennie az átalakításnak, tehát ha adott inputokkal futtatjuk a programot az átalakítás előtt és után, akkor a program kimenetének

mind a két esetben ugyan annak kell lennie. Kifejt további motivációkat, hogy miért szükséges a refaktoring, és milyen tulajdonságain javíthatunk vele. Ilyen például a függvények érthetősége, újrafelhasználhatósága, karbantarthatósága. Bemutat sok refaktoráló módszert, mint például a Creating An Abstract Super Class, mely szükséges, ha egy meglévő öröklődési hierarchiát bővítünk egy új osztállyal, mely osztály metódusainak a többségét az egyik testvérosztályból másoljuk át, majd átírjuk a függvények törzsét. Ilyenkor célszerű egy köztes absztrakt osztályt létrehozni, és ebben létrehozni az átmásolt függvényeket, majd ezt az eredeti ősosztályból származtatjuk, illetve bekötjük alá azokat a gyerekosztályokat, amelyekben ezek a metódusok megtalálhatóak. Ezekre a módszerekre mind megoldásokat is talál, és kifejti ezeket részletesen, illetve ábrákkal segíti az ott leírtak megértését.

Többen foglalkoztak már közvetlenül a program másolatok megszüntetésével. Tairas Phd dolgozatában leírja a veszélyeit a kód másolatoknak, ír olyan programokról, melyek a kódmásolatokat automatikusan megtalálják a programkódban, illetve a megtalált másolatok vizsgálatáról, és a vizsgálat után a klónok megszüntetéséről. Először röviden említi ezeket az eljárásokat, majd a dolgozata következő fejezeteiben részletesebben tárgyalja minden eljárásnak a hátterét, azaz, hogy a program hogyan lett implementálva. Részletesen kifejti a program klón keresését, a megtalált klónok megjelenítését, illetve a klóntípusokat. Továbbá a klón analízist és a programban megtalált klón megszüntetését is kifejti.

A Duploc nevű program [26] a másolt kódrészletek megtalálásában segít, és nem alakítja át a forráskódot más adatstruktúrává, hanem lexikálisan, a white space-k (szóköz, tabulátor, újsor) figyelembe vétele nélkül. Az eszköz FAMIX sémát használ, mely egy nyelv-független objektum orientált metamodell. Lényege, hogy elég csak az osztályok, és azok adattagjai közötti kapcsolatot reprezentálni. Ahhoz, hogy a refaktorálás megfelelő legyen, sok esetet figyelembe kell vennie a másolatok kapcsolatáról, például a két osztály, melyben a másolatok találhatóak, milyen hierarchiai viszonyban állnak egymással, illetve a másolatokban milyen változók fordulnak elő. Javasolja, hogy a refaktoring után manuálisan is legyen átnézve a refaktorált kód.

Higo, Kusumoto és Inoue szintén a programklónok eliminálását tűzték ki célul [10]. Munkájukban kiemelik a főbb eseteket, amik befolyásolhatják a refaktorálás menetét.

Kezdetben megemlíti a hat fokozatát az eliminálásnak, hivatkozva Mens & Tourve, 2004-es munkájára [11], majd mind a hat fokozatot egyesével ki is fejtik. Például a kóddektálásnál kifejtik, hogy több szempontból is másolatnak mondható két szoftver. Ezek között van a sorok szerinti egyezés, a tokenekre bontással kapott egyezések, a programrész absztrakt szintakt fájának egyezése más programrész absztrakt szintakt fájával, illetve a metrika szerinti összehasonlítás. Ők az utolsó szempontot, a metrika szerinti összehasonlítás módszerét használták fel. Ezután felsorolják a klónok eltüntetésére során használt refaktoringokat, amelyek egy halmazát hajtják majd végre az adott klónokon, attól függően, hogy a klónok milyen tulajdonsággal rendelkeznek, illetve milyen kapcsolat áll fenn köztük (például közös osztályban vannak-e, van-e közös őse az őket tartalmazó osztályoknak). Ezeket öt csoportra osztják, melyek mind különböző eseteket fednek le. Leírják részletesen az esetekhez a megoldásokat, és feltűntetik, hogy melyik esetben kell paramétereket, return utasítást, illetve egyéb átalakításokat alkalmazni. Bemutatja az ARIES nevű szoftvert, amely segít a felhasználónak a másolatok megszüntetésében, és használja a fentebb említett eseteket, és az alapján végzi el a szükséges strukturálásokat.

Meg kell említenem továbbá, hogy kódmásolatok megítélése bizonyos esetekben pozitív is lehet. Kapser és Godfrey szerint a nagy projektek akár 10-15 százalékát is kitehetik kód másolatok, melyek között létezhetnek olyan esetek, amikor a másolatok nem tekintendők rossznak [13].

Munkám során Java nyelvi sémára alapuló program reprezentáción dolgoztam, melyhez hasonló nyelvi sémákat már korábban is felhasználtak refaktorálással kapcsolatos kutatásokra. A refaktorálás leírására és végrehajtására kézenfekvő infrastruktúrát nyújtanak a gráf transzformációk [4]. A C++ nyelven bemutatott tanulmány bemutatja az elvi és a konkrétan végrehajtható refaktorálás közti különbségeket. A gráf transzformációk mellett OCL kifejezéseket is használnak a szerzők, így teszik könnyebbé a gráfban való munkát. Az OCL (Object Constraint Language) segít a gráfban való navigálásban, illetve szabályok létrehozásában. A cikk példaként egy Extract Class refaktoráló eljárást említi példaként, ahol egy osztály összetartozó adatait összevonják egy közös osztályba. A szerzők hasonló eszközöket használnak fel egy későbbi tanulmányban, melynek tárgya a C/C++ preprocesszor direktívák átalakítása [5].

2.2. Pár hasznos refaktoráló eszköz

IntelliJ IDEA [14]: Az egyik legelső Java IDE, amiben megvalósították a refaktoringokat. Létezik JSP programkódot refaktoráló része is, de mi most a Java programnyelvre koncentrálunk. Az IDE letölthető az internetről. Implementált refaktoringok:

Extract Method Object, Type Migration, Extract Class, Inline Superclass, Remove Middleman, Wrap Return Value, Move Instance Method, Extract Subclass, Replace Method Code Duplicates, Extract Method, Introduce Constant, Extract Interface, Extract Superclass, Pull Members Up, Push Members Down, Inline Method, Encapsulate Fields, Replace Temp With Query, Replace Constructor With Factory Method.

Eclipse's Java Development Toolkit (JDT) [15]: Az Eclipse Java nyelvre kifejlesztett refaktoráló eszköze. Ez a rész a menüsorban egy külön legördülő menüz is kapott. Bizonyos refaktoring műveletek külön billentyű kombinációt kaptak, mint például a rename, amely Alt+Shift+R billentyűkombinációra jön elő. Másik érdekesség, hogy ez a fejlesztőkörnyezet rendelkezik úgynevezett „Refactoring script”-el, mely arra szolgál, hogy naplózni lehessen a kódunkon elvégzett refaktoringokat. Implementációk:

Rename, Move, Extract Local Variable, Extract Constant, Extract Interface, Extract Superclass, Extract Method, Inline Superclass.

NetBeans [16]: Ahogy az ismertebb fejlesztőkörnyezetekben, úgy a NetBeans fejlesztőkörnyezetében is elérhető a refaktoring. Elérhető még egy olyan modul is hozzá, amellyel a felhasználó saját refaktorálási szabályokat írhat. Implementációk:

Rename, Extract Method, Encapsulate Fields, Move Class, Change Method Parameters, Extract Interface, Extract Superclass, Move Inner to Outer Level.

Oracle JDeveloper (Java nyelvre) [17]: Az Oracle JDeveloper nevű fejlesztőkörnyezete is rendelkezik refaktorálással Java nyelvre. Különlegessége, hogy nem csak szimpla, Java forráskódokra, hanem webes JSF, cpx, és jpx fájlokat is kezelhetünk vele. Implementációk:

Rename, Move, Refactoring pagedef.xml Binding Objects, Refactoring ADF Business Compinents, Refactoring ADF Business Component Object Attributes.

Embarcadero Delphi [18]: Ezt a példát azért hoztam fel, mert meg szeretném mutatni, hogy nem csak Java nyelvre léteznek refaktoráló programok, hanem ez egy olyan terület, amely programozási nyelvektől függetlenül jelen van a programozók körében. Azonban ez nem azt jelenti, hogy egy bizonyos problémát mindig ugyan azzal az algoritmussal oldunk meg. Vannak olyan programozási nyelvek, amelyek elegánsabb megoldást kínálnak egyes problémákra, míg mások megnehezíthetik a dolgunkat. Vegyük példának a kimenteni kívánt paramétereket. Java nyelvnél könnyű dolgunk van, hiszen minden referenciaként van átadva, de például C++-nál külön jelezni kell, hogy referencia vagy érték szerint szeretnénk átadni. Az előbb említett program Delphi nyelvre lett készítve, és a megszokott refaktorálásokat hajtja végre. Implementáció:

Rename, Change Parameters, Extract Method.

3. Elvi megközelítés

Mint egyéb más kódjavító programoknak, az én programomnak is az alapelve, hogy a meglévő kódon olyan átalakítást végezzen, amely a kód minőségén javít, és a létrejött kód futtatható legyen, illetve funkcióját ne veszítse el. Azért fontos bizonyos metrika értékek javítása (mint például a Duplikált kódelemek száma a kódban), mert ezek az értékek egyfajta mérőszámként viselkednek a programunk minőségére vonatkozóan. Természetesen van olyan érthetőséget befolyásoló tényező a forráskódban, amelyet nem lehet programmal vizsgálni, és észlelni, ha nem megfelelő. Ilyen Tulajdonság például az, hogy az osztályok, változók, paraméterek nevei megfelelnek-e a funkciójuknak, hiszen ennek eldöntésére először is a programunknak értelmeznie kéne az adott nevet, majd rá kéne jönnie a funkcióra, és e két dolgot összekapcsolnia. Ennek ellenére, amely tulajdonságokat meg lehet mérni metrikák segítségével, és ki lehet mutatni egyfajta rossz értéket, azokat érdemes megszüntetni.

Adott egy programkód, melyben elszórva találhatóak olyan kódrészletek, amelyek ha a táblázatokban, szövegekben, változónevekben, konstansokban el is térnek, de az egymás után végrehajtott utasítások megegyeznek. Ha sok ilyen kódrészlet található elszórva a kódban, az nagy veszélyt jelent a kód megbízhatóságára. Egy nagy projektnél pedig nagyon elszaporodhatnak a kódmásolatok, így a hibagyakoriság is megnőhet.

A megoldásra több módszer is adott a refaktoráláson kívül, viszont mégis a legkézenfekvőbb megoldás, ha a kód átírásával szüntetjük meg a problémát, hiszen a

klónok megléte is arra utal, hogy a kódrészlet gyakran van használva, és valószínűleg még több helyen használva lesz, és a refaktoring által egy helyen elég átszerkeszteni a programot, ha változtatni kell. Azonban, mint a legtöbb refaktorálásnál, itt is sok a hibalehetőség.

A megszüntetésnek több része van. Először vizsgálatokat kell végezni, hogy a megtalált kód példányok valóban egymás másolatai, vagy mind más feladatot látnak el, viszont struktúrájukban hasonlítanak (például egy függvényhívásnál nem ugyan azt a függvényt hívják meg). Utána a környezetből való kiemelhetőségre kerül a hangsúly, hiszen lehetnek olyan esetek, amikor ezt lehetetlen megtenni. Végül létrehozunk egy metódust, amiben a kód példányok tulajdonságaitól függően beállítjuk a bemeneti paramétereket, illetve kimentjük a megfelelő változókat. Ehhez az eljáráshoz több különböző módszer is tartozik, ezek közül választottam ki egyet. A módszer kiválasztásánál három esetet kell jobban figyelembe venni. Ezek közül az egyik, amikor a példányok egy osztályon belül vannak, a másik, amikor a példányok nincsenek egy osztályon belül, illetve ha a kódrészletünkből készült metódusnak több visszatérési értékkel kell, hogy rendelkezzen a másolat környezetéből kifolyólag.

3.1. Esetek, és kiküszöbölendő problémák

Az első, és legfontosabb a változó-hivatkozások, illetve változó-módosítások. Fontos megnézni, hogy az egyes klónok milyen változókat használnak, hiszen ha kiemeljük a kódrészletet egy metódusba, akkor a használt változókat paraméterként át kell adni a hívás helyén. Ha olyan műveleteket hajtunk végre a változókon, amelyek bizonyos osztályváltozók értékére hatással lehetnek, nem árt referenciaként átadni. Bár Java programozási nyelv esetén a változók alapértelmezetten referenciaként adódnak át, de vannak nyelvek, ahol ezt a szándékot jelezni kell. Azonban ez sem megoldás, hogyha nem a referencia által mutatott objektum osztályváltozóinak az értékét módosítjuk, hanem magát az objektumot, amelyre a változó mutat. Ilyenkor az eredeti objektumra mutat a változó a függvény végrehajtása után is, még akkor is, hogyha egy új objektumra cserélődne az eredeti működés szerint.

Egy másik probléma a kifelé mutató élekkel kapcsolatos. Lehet olyan eset, amikor a klón egy vagy több cikluson belül vannak. Ez is csak abban az esetben probléma, hogyha a klónok tartalmazznak olyan break, vagy continue utasítást, amihez a klónon belül nem

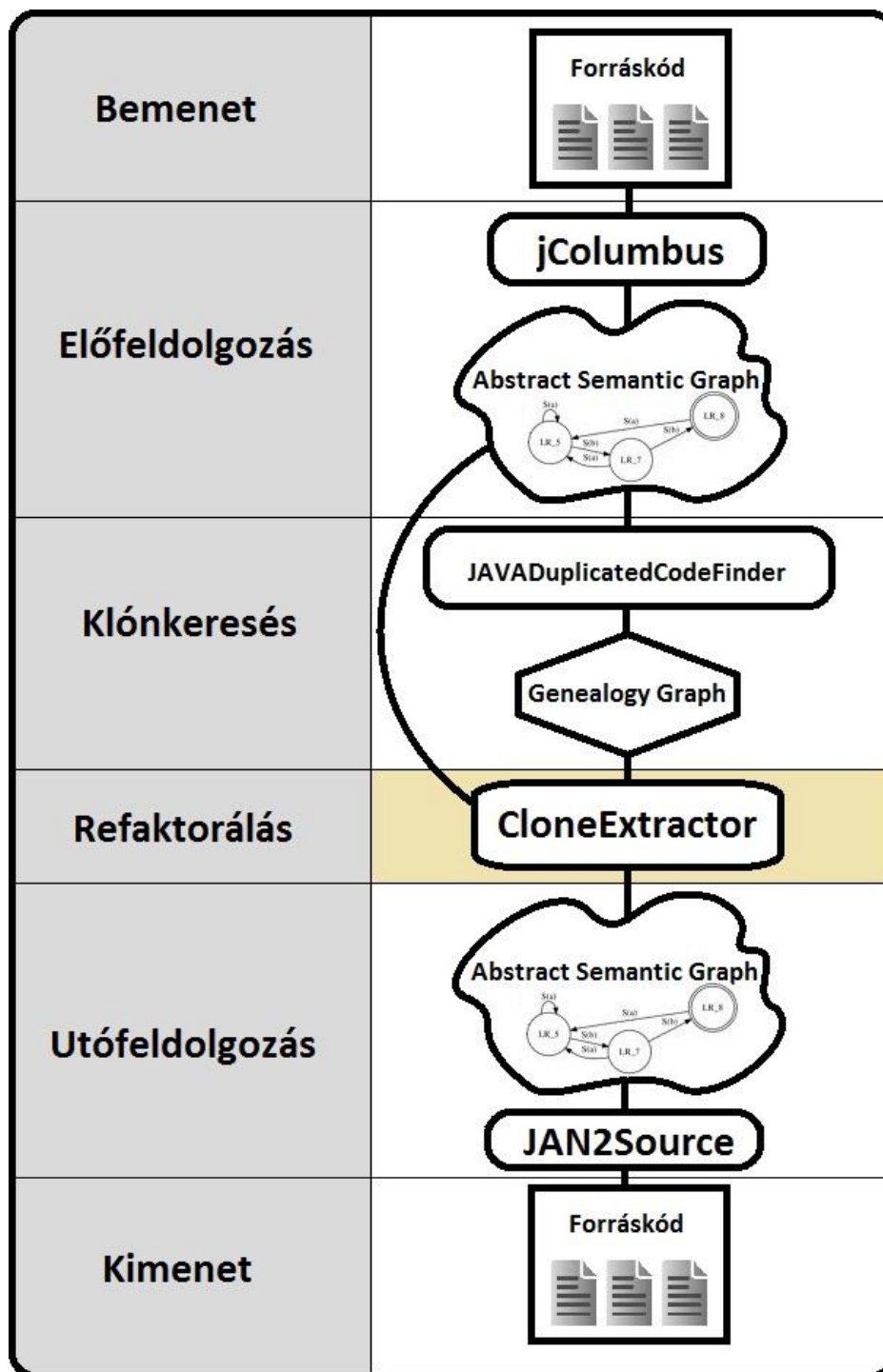
tartozik ciklus, mert ilyen esetben a fő ciklusból ki kellene lépnie, de ha ezt nem kezeljük le, akkor akár egy végtelen ciklusba is belemehetünk. Az is problémát jelenthet, ha return utasítás van a kódban, és ez két okból is. Egyrészt ha benne hagyjuk, akkor az általunk létrehozott metódusból fog kilépni, és nem abból, amelyikben eredetileg a klón található volt. Illetve ha az eredeti metódusnak volt visszatérési értéke, azt is át kell vinni az őt hívó metódusba. Ezen problémák megoldására olyan összetett adatszerkezete van szükség, amely tudja kezelni a megfelelő ugrásokat, és a hívott helyen meg is tudja őket hívni.

4. Architektúra

A rendszer, mellyel a klónok refaktorálását végzem, nem csupán egyetlen programból áll, hanem részt vesznek benne bizonyos segédprogramok. Erre azért van szükség, mert a kódban nagy változtatásokat kell végeznünk, és célszerű, ha ezt nem magán a forráskódon tesszük meg, hanem a forráskódot valamely könnyebben kezelhető adatstruktúrába visszük át, és ott hajtjuk végre a változtatásokat. Erre egy **jColumbus** nevezetű programot használok, amely a paraméterben kapott programkód fájlokat egy **ASG (Abstract Semantic Graph)** gráffá alakítja. Ez a gráf reprezentálja a forráskódot. Minden, Java programnyelvben előforduló utasításra, hivatkozásra, struktúrára van benne egy külön csúcs, illetve a forráskódban összetartozó részek a gráfban a megfelelő csúcsok közötti élekkel vannak reprezentálva. Ilyenek például az osztály és adattagjai, függvényei közti kapcsolat, a változó és függvény-hivatkozások, a ciklusok, szelekciós vezérlések és feltételeik közti kapcsolatok. Ebben a gráfban egy **DuplicatedCodeFinder** nevezetű program keres egyező gráf részeket úgy, hogy a gráf pontok típusainak meg kell egyezniük, de bizonyos tulajdonságaikban eltérhetnek, mint például a változó- és metódushivatkozások, a bináris operátorok típusai, változók és metódusok nevei. Ez a program a megtalált klónokat egy **genealogy**-nak nevezett gráfban tárolja el. Többször lefuttatva a programot a későbbi verziókon, a genealogy fájl folyamatosan bővül, verzióként egy **system** csomóponttal. Nekünk ebben a systemek közül majd csak az utolsóra lesz szükségünk, mivel az reprezentálja az aktuális verziót. Minden system rendelkezik klónosztályokkal, és klónpéldányokkal az aktuális verziónak megfelelően. Ezt egy gráfban binárisan eltárolva megkapja az általam írt, **CloneExtractor**-nak nevezett refaktoráló program. A genealogy gráfban lekérdezhető az adott ASG fájl útvonala, így azokat be tudjuk olvasni, és azon elvégezni az átalakításokat. A CloneExtractor által

véghezvitt, megváltozott ASG-t lementjük, majd egy **JAN2Source**-nak nevezett program az ASG-ből a kért részeket átalakítja forráskóddá.

Ezt a sok programot felparaméterezni és futtatni a megfelelő sorrendben sok munkát venne igénybe, mind a fejlesztés során, amikor tesztelni kellene a programot, mind a használatnál. Ezért jött az ötlet, hogy ezeket a programokat egyesítsük egy Eclipse plug-in-ban, és így a klónosztályok megjelenítése, és kiválasztása is pár kattintással megoldható. Így analízáláskor a plug-in lefuttatja a JColumbus, és DuplicatedCodeFinder programot, és a kimenetet (a megtalált klónokat) meg is jeleníti egy ablakban. Ekkor a refaktorálandó klónra kattintva jobb gombbal felnyílik egy menü, ahol kiválaszthatjuk a refaktorálás menüpontot, azon belül az ExtractMethod műveletet. Ekkor a programom megkezdí a futását, majd a kiadott ASG-n elindítja a JAN2Source programot, így az eredeti fájlok átírásra kerülnek.



5. Algoritmus

5.1. Szerkezeti felépítés

Mint ahogy említettem, a refaktorálást a CloneExtractor-nak nevezett programom végzi, és a klónkereső program által megtalált klónok közül a paraméterben átadott azonosítóval ellátott klónosztályon végzi el. A program több kisebb részre bontható szét,

melyek mind elengedhetetlenek a refaktorálás szempontjából. Ezek nem mások, mint a klón osztály beolvasása, megfelelő klónosztály kiválasztása, a klón példányok analízálása, adatok összegyűjtése, a begyűjtött adatok alapján a klónosztály több részre bontása (ha szükséges), a metódus helyének kiválasztása, a metódus felépítése, visszatérési érték kiválasztása, a metódushoz a szükséges paraméterek hozzáadása, klón példányok átalakítása, metódus hívásának paraméterezése. A refaktorálás egyes esetei megkövetelik, hogy létrehozzunk új segédosztályokat a refactoring csomagba. Ezeknek két fajtája van, az egyik a változók visszamentéséről gondoskodik, és a **NewSupClass** prefixummal rendelkeznek. Ebből az osztályból a refaktorálás során mindig készül egy, hogyha éppen szükség van rá, illetve általában hozzáteszünk egy számot a nevéhez, hogy egyedi legyen a csomagban. A másik csoport, amely a kifelé mutató return, break, és continue utasításokat kicseréli a megfelelő beépített függvényeire. Ezt **Refactored** osztálynak hívjuk, és ebből mindig csak egy van.

5.1.1. A klón osztály beolvasása

Az első, amit ebben a részben végrehajtottunk az egész genealogy gráf beolvasása. Ezt csak akkor tehetjük meg, ha paraméterként a program megkapta már ennek a fájlnek az elérhetőségét. Azonban nincs szükségünk az összes információra ebből a gráfból, így második lépésként egy függvénnyel kiválasztjuk az utolsó systemet (verziót), és annak a klón osztályait, és klónpéldányait olvassuk be. Saját osztályaim vannak ezek tárolására, melyek csak azokat az információkat fogják tartalmazni, amelyekre a programomnak szüksége van. Ilyenek például a systemhez tartozó ASG gráfok, a klónosztály azonosítója, és a hozzá tartozó klónpéldányok, a klónpéldányokhoz tartozó ASG gráf (mindegyikhez pontosan 1 tartozhat), a hozzá tartozó csúcsok azonosítója az ASG gráfban. A klónosztályokat a State nevű osztályom tárolja.

Main függvény

```
A genealogy fájl létezésének ellenőrzése;  
A genealogy fájl beolvasása;  
State létrehozása;  
Utolsó system kiválasztása, és betöltése a state-be;  
A refaktorálást kontrolláló osztály létrehozása;  
A refaktorálás elindítása;  
State bezárása;
```

5.1.2. A megfelelő klónosztály kiválasztása

A megfelelő klónosztály kiválasztása azt jelenti, hogy a paraméterként megkapott azonosítóval rendelkezőt refaktoráljuk. Eredetileg úgy lett a program elkezdve, hogy a megkapott system összes klónosztályát refaktorálja. Ez lett átalakítva úgy, hogy egyenként

lehesen refaktorálni, mivel lehet olyan eset a klónok között, hogy az egyik klónosztály megszüntetése megváltoztatja egy másik klónosztálybeli példány struktúráját, és így hibákat eredményezhet. A refaktorálandó klónosztály azonosítóját a CCid változó tárolja el. Ezt a kiválasztást a refaktorálást kontrolláló osztály végzi el.

```
For (az első klónosztálytól az utolsóig)
    If (CCid==0, vagy CCid==klónosztály.azonosító)
        For (Az összes klónpéldányra a klónosztályon belül)
            Klónpéldány.analizálás();
        For vége;

    A klónosztály szétválasztása;
    For (Összes kisebb klónosztályra a klónosztályon belül)
        Refaktorálás példányosítása;
        Refaktorálás elindítása adott klónosztályra;
    For vége;
If vége;
For vége;
```

5.1.3. A klón példányok analízálása, adatok begyűjtése

A klónpéldányokon sok vizsgálatot kell elvégezni két okból. Első, hogy megtudjuk, hogy a klónosztályban levő klónpéldányok refaktorálhatóak-e. Ehhez összegyűjtjük a változó és függvény-hivatkozásokat, az előforduló bináris és unáris műveletek típusait. A második, hogy az összeolvasztható klónpéldányoktól összegyűjtsük az összeolvasztáshoz szükséges adatokat, mint például a használt változók, az eltérő konstansok, közös metódusban vannak-e, tartalmazznak-e kifelé mutató éleket, melyik példány után mely változókat kell kimenteni (ez attól függ, hogy a kódban melyik változó van használva a példány után). Az adatok begyűjtését a klónpéldányok analízáló metódusa hajtja végre, és a klónpéldányokon belül külön-külön el van tárolva. Az adatok begyűjtését úgynevezett **vizitorok** végzik el. Ezek olyan absztrakt osztályokból általam származtatott osztályok, amelyekkel be tudjuk járni az ASG gráfot adott csúcstól kiindulva. Minden származtatott osztály más és más feladatot lát el, és más csúcsokat járnak be. A csúcsok azonosítóját tároljuk el egy vektorban, a bejárás sorrendjében, és a szétválogatásnál ezeket használjuk fel. Itt tároljuk el továbbá azt is, hogy mely változókat kell majd kimenteni. Ezt onnan kapjuk meg, hogy vesszük a használt lokális változókat, és megnézzük, hogy ezekből mennyi az, amelyre a klónpéldány után is hivatkozunk, illetve az összes globális változót is használnak tekintjük.

```
Analízáló függvény
    A klónpéldány adatainak törlése;
    Változó-hivatkozásokat begyűjtő vizitor példányosítása;
    A klónpéldány csúcsainak bejárása a vizitorral;
```

```

Függvény-hivatkozásokat begyűjtő vizitor példányosítása;
A klónpéldány csúcsainak bejárása a vizitorral;
Unáris és binárisokat bejáró vizitor példányosítása;
A klónpéldány csúcsainak bejárása a vizitorral;
Konstansokat bejáró vizitor példányosítása;
A klónpéldány csúcsainak bejárása a vizitorral;

A használt lokális változók kigyűjtése;
A használt globális változók kigyűjtése;
A klónpéldány után használt változók;
Analizáló függvény vége;

```

5.1.4. A klónosztály több részre bontása

A klónosztály több részre bontása azért fontos, mert ha egy klónosztály több olyan klónpéldányt tartalmaz, amelyek különböznek olyan tulajdonságokban, amelyek miatt nem lehet őket refaktorálni, akkor szét tudjuk bontani több kisebb klónosztályra, amelyek már refaktorálhatóak. Ekkor jöhetnek létre olyan klónosztályok, melyeknek csak egy klónpéldányuk van. Ebben az esetben ezt a klónosztályt elvetjük, és nem refaktorálhatónak tekintjük. Az olyan klónosztályokat, melyek több klónpéldányt is tartalmaznak, refaktoráljuk. A következő két kódrészlet a klónosztályok szétbontását mutatja be. Az első a függvényhívásokat mutatja be, a másik a függvény megvalósítását.

```

Szétválogató függvény()
  Vágás (ASG összehasonlító);
  Vágás (Referencia összehasonlító);
  Vágás (Metódus-hivatkozás összehasonlító);
  Vágás (Operátor összehasonlító);
Szétválogató függvény vége;

Vágás függvény (összehasonlító);
  Méret=klónrész.méret();
  For(i=0-tól i< Méret)
    If (klónrész[i].példanyszam() > 1)
      While (klónrész[i] > 1)
        Egy f kiválasztása a klónrész[i]-ből;
        Átesszük f-et egy új klónrészbe;
        Összes g példányra klónrész[i]-ből;
        If(Összehasonlító.hasonló(f, g))
          Átesszük g-t f mellé;
        If vége;
      While vége;
    If vége;
  For vége;
  Töröljük a klónészt 0-tól Méret-1-ig;
Vágás függvény vége;

```

A szétválogató függvény meghívja a vágás függvényt annyiszor, ahány szempont szerint szeretnénk vágni, és minden összehasonlító függvény más szempontból hasonlítja össze a klónpéldányokat. Minden vágásnál a kezdetben van valamennyi klónosztály darabunk, és ezeket daraboljuk fel még jobban, és a kezdeti klónosztály darabokat töröljük. A végén már csak azok a klónpéldányok lesznek egy klónosztály darabban, amelyek minden kritérium szerint megegyeznek.

5.1.5. Metódus helyének kijelölése

A következőkben már nem az egész klónosztályról beszélünk, hanem a szétdarabolás után keletkezett darabjairól. Egyenként megy rajtuk végig, és ami rendelkezik elegendő számú példánnyal, azt refaktorálja. A metódus helyének meghatározása azt jelenti, hogy a készülő metódust olyan helyre kell rakni, ahol minden klónpéldány el tudja érni. Legkézenfekvőbb az az eset, amikor mindegyik klónpéldány egy osztályon belül van. Ekkor a közös osztályban kell elhelyezni a metódust, és ebben az esetben nem kell, hogy statikussá tegyük a függvényt. Ha azonban már egy példány is más helyen van, akkor kénytelenek leszünk statikussá módosítani a függvényt. Ha nincs közös osztály, akkor annak a példánynak az osztályába teszi bele a metódust, amelyik a klónpéldány listában az első.

5.1.6. A metódus felépítése

A metódus felépítése nagyon fontos. Azzal kezdjük, hogy a klónpéldányok közül az elsőt kiválasztjuk, és készítünk belőle egy másolatot. Itt két esetet kell elkülöníteni. Az első, amikor a példányok mindegyike metódus, a másik, amikor a példányok egyike sem metódus. Ha metódusokról van szó, akkor a lemásolást követően törölni kell a metódusból a paraméterlistát, a függvény nevét, illetve a static és final módosítókat. A metódus felépítése tartalmazza még a megfelelő osztályba való elhelyezést is. A függvényt statikussá tesszük, ha nincs közös osztály. Azonban hogyha a klónok nem függvények, akkor létre kell még hozni a függvényt, mely magába foglalja, az előbb elkészített példány másolatot. Ilyenkor még utasításokat kell végrehajtani, hogy csúcsról csúcsra létrehozzuk a függvényt, illetve a block-ot, ami a függvény törzseként szolgál majd. Ezután egyenként be kell másolnunk a függvény törzsébe a másolat gyökereit. Az esetleges változtatásokat is a törzsön itt kell végrehajtani, mint például a kifelé mutató return, break, continue utasítások lecserélését a megfelelő, Refactored osztálybeli metódusra, illetve a kimentendő változókra való összes hivatkozás átírása a klónosztályhoz létrehozott SupClass megfelelő osztályváltozóra. Továbbá ha használnunk kell a Refactored objektumot, akkor a metódus elején példányosítanunk kell, illetve a legvégén egy return utasítással ezt az objektumot

kell majd visszaadnunk (mivel ekkor a metódus visszatérési típusa maga a Refactored generikus osztály, megfelelő paraméterekkel ellátva).

Metódus elkészítése függvény ()

```
If (klónpéldány.típus()==metódus)
    A metódus átnevezése extractedMethod-ra;
    A metódus paraméterlistájának törlése;
    A metódus láthatóságát publikusra állítjuk;
    A metódushoz visszatérési érték rendelése;
    Az eltérő konstansokhoz paraméterek rendelése;
    Return;
If vége;
A metódus létrehozása;
A metódushoz block létrehozása;
A metódusnak beállítjuk törzséül a block-ot;
A blockhoz hozzáadjuk a másolatot;
A metódusnak nevet adunk;
A metódus láthatóságát átállítjuk public-ra;
A metódushoz visszatérési értéket rendelünk;
A kifelé mutató return break continue-k kicserélése;
Az eltérő konstansokhoz paramétereket rendelünk;
A bemeneti paraméterek megadása;
Metódus elkészítése függvény vége;
```

5.1.7. A metódus visszatérési típusának kiválasztása

A visszatérési érték megválasztása fontos. Itt több szempontot is figyelembe kell venni, mind például a visszamenteni kívánt változók számát, van-e kifelé mutató return, break vagy continue utasítás, és a klónokat tartalmazó metódusnak mi a visszatérési értéke. A módszer a következő. Ha a klónpéldány egy metódus, és nem kell kimenteni változó értékeket, akkor simán megadhatjuk a visszatérési típusnak a klónpéldányok visszatérési értékét. Ha van kifelé mutató el, vagy kell visszamenteni változóértéket, akkor az általam kidolgozott visszatérési értéket kell használni. Ekkor el kell készíteni egy új SupClass osztályt, amelyben eltároljuk ideiglenesen minden kimenteni kívánt változó értékét. Illetve létre kell hozni, vagy (ha már létezik) elő kell keresni a Refactored osztályt, és annak generikus paramétereként át kell adni. A másik paramétere a típusnak a klónpéldányt tartalmazó metódus visszatérési értéke. Ha ez primitív típus, akkor megkeressük a hozzájuk tartozó beépített osztályt. Ha ez void, akkor egyszerűen Object-nek állítjuk. Ha ezek egyike sem teljesül, akkor biztos, hogy olyan nem metódus klónpéldányokról van szó, melyeknek nincs kifelé mutató utasításaik. Ez esetben void típust kell beállítani, hiszen a függvény ekkor eljárásként funkcionál, és meghívás után nincs más teendőnk.

Visszatérési érték beállítása ()

```
If (kiemelklónpéldány.metódus() és mentett_változók==0)
    Metódus.visszatérési_érték=tartalmazó_metódus.visszérték;
```

```

ElseIf (van_kimutató_él() vagy mentett_változó>0)
    Új típus létrehozása Refactored Objektumból;
    A típus paraméterezése;
    Metódus.visszatérési_érték=elkészített_típus;
Else kezd
    Új void típus létrehozása;
    Metódus.visszatérési_érték=void_típus;
Else vége;
Visszatérési érték beállítása vége;

```

5.1.8. A metódushoz a szükséges paraméterek hozzáadása

A metódus paraméterezésének két része van. Először akkora adunk hozzá paramétereket, amikor az eltérő konstansok problémáját akarjuk megoldani. Ilyenkor csak össze kell gyűjteni az összes klónpéldánynál a páronként eltérő konstansok helyének az unióját, majd mindegyik értéket helyettesíteni kell egy paraméterrel, mégpedig olyan típusú paraméterrel, mint amilyen a konstans típusa. Ezután a konstansokat ki kell cserélni egy hivatkozásra az éppen létrehozott paraméterre. Ha ezzel megvagyunk, jöhet a következő rész, amikor a bemenő paraméterekkel ruházzuk fel a metódust. Azoknál a változóknál kell paramétert alkalmazni, melyek lokálisak, vagy páronként vizsgálva a klónpéldányokat nem egyeznek meg. A konstans paraméterek nevei „literal” prefixummal kezdődnek, és egy számmal végződnek, illetve teljesen egyediek. A változókhoz rendelt paraméterek prefixumai megegyeznek a változó nevével, és ezekhez is számot rendelünk, amely teljesen egyedivé teszi. Ezek a paraméterek fontosak, hiszen az összes klónpéldány esetén a függvény paraméterezése egyedi lesz, és tudnunk kell, hogy melyik klónpéldányra a függvényt mely paraméterekkel hívjuk meg. Ezért szükségünk van egy listára minden példány esetén, amely tartalmazza sorba véve a paramétereket, amiket majd át kell adnia.

5.1.9. A klón példányok átalakítása

Ez a legnehezebb része a refaktorálásnak. Itt szinte minden eddig összegyűjtött adatunkra szükségünk lesz. Két részletben zajlik az átalakítás. Az első részletben a klónpéldányok után a megfelelő utasításokat beszúrni, majd a klónpéldányhoz tartozó utasításokat kell törölni. Ez a sorrend azért fontos, mert ha először lennének törölve az utasítások, akkor fontos adatokat veszítenénk az adott példány összetételéről, illetve a klónpéldány remek támpontot nyújt abból a szempontból, hogy hova kell beszúrni az új utasításokat. Itt is több esetet kell figyelembe vennünk. Egy függvény esetén, ahol nem kell kimenteni változóértékeket, elég csak return-al visszaadni az újonnan létrehozott metódus hívásának eredményét (kivéve void esetén, ahol a return elhagyható, és csak a függvényt hívjuk meg). Ha Refactored objektumot használunk, akkor egy megfelelően paraméterezett

generikus Refactored típusú változót kell példányosítani, és ebbe kell a metódushívás visszatérési értékét kimenteni. Ez tartalmazni fog információkat a kimenteni kívánt változók értékeiről, illetve arról, hogy végrehajtódott-e a return, break, continue utasítások közül valamelyik. Először a változók kimentéséről gondoskodunk, mert a megváltozott értékeket mindenképp el kell menteni a megfelelő helyre. Ez után vizsgáljuk, hogy történt-e valamilyen kifelé irányuló ugró utasítás. Ha történt, akkor a példány helyén végre kell azt hajtani, így garantálva a program helyes működését.

Példányok refaktorálása függvény ();

```
For (Minden klónpéldányra az klónosztályban)
    If (Refactored osztály lett használva)
        Refactored importálása a klónpéldány osztályához;
        NewSupClass importálása a klónpéldány osztályához;
    If vége;
    A példányt tartalmazó block kiválasztása;
    A metódushívás létrehozása;
    If (Nem egy osztályon belüliek a klónok)
        A metódushívás statikus hívássá alakítása;
    If vége;

    Mutató példányosítása;
    If (klónpéldány.metódus())
        Mutató az aktuális példány törzsének a végére;
    Else
        Mutató az aktuális példány utolsó utasítására;
    Else vége;

    If (klónpéldány.metódus() és mentett_változók==0)
        If (klónpéldány.vissztípus==void)
            Mutató által mutatott helyre beszúrjuk a hívást;
        Else
            Return utasítás létrehozása;
            Return.vissz_értéke=metódus_hívás;
            Mutató által mutatott helyre beszúrjuk a Returnt;
        Else vége;
    ElseIf (van_kimutató_él() vagy mentett_változó>0)
        Refactored típusú Ref változó létrehozása;
        Értékadó operátor létrehozása;
        Operátor.bal_oldal=Ref;
        Operátor.jobb_oldal=metódus_hívás;
        Mutató által mutatott helyre beszúrjuk az Operátort;
        Mutató növelése;

    For(összes kimentendő változóra)
        Értékadó operátor létrehozása;
        Operátor.bal_oldal=használt_változó;
        Operátor.jobb_oldal=mentett_változó;
        Mutató helyére beszúrjuk az operátort;
```

```

        Mutató növelése;
    For vége;

    A kellő if utasítás hozzáadása lekezelte ugrásokra;
Else vége;

For (Minden eltérő konstansra)
    Metódus_hívás.addParaméter(konstans_értéke);
For vége;
For (Minden használt változóra)
    Metódus_hívás.addParaméter(változó_hivatkozás);
For vége;

Példány utasításainak törlése;
For vége;
Példányok refaktorálása vége;

```

5.2. Egy egyszerű forráskód példa

Követezőkben a programomat bemutatom egy nyílt forráskódú rendszeren talált klónon. A felhasznált forráskód a log4j[19] programból származik, a két klón a LogBrokerMonitor.java fájlban található, a LogBrokerMonitor osztályban. Az aktuális klón két függvény, melyek csak különböző típusú konstansokban térnek el. A példa a refactoring hasznosságát is hivatott bemutatni, hiszen olyan klónról van szó, amely nem eredményez a klónok helyén, illetve a létrehozott metódussal komplexebb forráskódot azzal, hogy végrehajtjuk rajta a klónok kiemelését.

```

920
921  protected JMenuItem createNoLogTableColumnsMenuItem() {
922      JMenuItem result = new JMenuItem("Hide all Columns");
923      result.setMnemonic('h');
924      result.addActionListener(new ActionListener() {
925          public void actionPerformed(ActionEvent e) {
926              selectAllLogTableColumns(false);
927              // update list of columns and reset the view
928              List selectedColumns = updateView();
929              _table.setView(selectedColumns);
930          }
931      });
932      return result;
933  }
934

```

CloneInstance~134

```

906
907 protected JMenuItem createAllLogTableColumnsMenuItem() {
908     JMenuItem result = new JMenuItem("Show all Columns");
909     result.setMnemonic('s');
910     result.addActionListener(new ActionListener() {
911         public void actionPerformed(ActionEvent e) {
912             selectAllLogTableColumns(true);
913             // update list of columns and reset the view
914             List selectedColumns = updateView();
915             _table.setView(selectedColumns);
916         }
917     });
918     return result;
919 }
920

```

CloneInstance~135

Az első szembetűnő különbség a két függvénynek a neve. A névkülönbség könnyen kiküszöbölhető, hiszen a metódusok belsejét cseréljük ki ebben az esetben egy függvényhívással. A másik különbség a konstansok, melyekből 3 is különbözik. A String konstansnál a „Hide” és „Show” szavak, a karakternél a 'h' és az 's', a logikai konstansnál az értéke az egyik függvényben igaz, a másik függvényben viszont hamis. Ezek a különbségek könnyen feloldhatóak paraméterek bevezetésével, illetve az, hogy a metódusok ugyan abban az osztályban találhatóak tovább könnyíti a klónok eltüntetését. Az eredményt a következő kép illusztrálja.

```

709 protected JMenuItem createAllLogTableColumnsMenuItem()
710 {
711     return extractedMethod("Show all Columns", 's', true);
712 }
713 protected JMenuItem createNoLogTableColumnsMenuItem()
714 {
715     return extractedMethod("Hide all Columns", 'h', false);
716 }

```

A klónok helye.

```

1280 protected JMenuItem extractedMethod(final String literal, final char literal1, final boolean literal2)
1281 {
1282     JMenuItem result = new JMenuItem(literal);
1283     result.setMnemonic(literal1);
1284     result.addActionListener(new ActionListener() {
1285         public void actionPerformed(ActionEvent e)
1286         {
1287             selectAllLogTableColumns(literal2);
1288             List selectedColumns = updateView();
1289             _table.setView(selectedColumns);
1290         }
1291     });
1292     return result;
1293 }
1294 }
1295

```

Az elkészült metódus.

Amint látható, a két klón-t elég volt két függvényhívásra cserélni. Amint látható, a különbséggént említett konstansok bekerültek a metódushívások paraméterlistájába. A létrehozott metódus paraméterlistájában 3 paraméter látható, melyek típusai megegyeznek az eltérő konstansok típusával. Ezen paraméterek mindegyike final tulajdonsággal is rendelkezik, mivel konstansokról van szó. A paraméterekre a megfelelő helyeken hivatkozás is található.

5.3. Egy komplexebb forráskód példa

A következő példában a klón már nem két függvény, hanem két while ciklus, illetve használ 4 olyan lokális változót, amelyek nem a klónokon belül lettek létrehozva, viszont használva vannak bennük, és az értéküket kívül is használni szeretnénk. Azonban ha sok olyan értékünk van, amit kívül is szeretnénk használni, akkor a sok visszatöltögetés nagyban megnehezítheti a kód átláthatóságát, és összezavarhatja a programozót. A példa ezt hivatott megmutatni, és azt a tényt, hogy még ekkor is lehetséges a kód átírása a működés megtartása mellett.

```
203     while (children.hasMoreElements()) {
204         CategoryNode node = (CategoryNode) children.nextElement();
205         String title = node.getTitle().toLowerCase();
206
207         String pathLC = element.getTitle().toLowerCase();
208         if (title.equals(pathLC)) {
209             categoryAlreadyExists = true;
210             // This is now the new parent node.
211             parent = node;
212             break;
213         }
214     }
```

CloneInstance~197

Adott két while ciklus, amik karakterről karakterre megegyeznek egymással. Ez nagyban megkönnyíti a dolgunkat, ám a klónokban előfordulnak olyan változók, amelyek visszatöltéséről gondoskodnunk kell. Ebben a példában kettő ilyen is van, a „parent”, amely egy „CategoryNode” típusú változó, illetve egy „categoryAlreadyExists” változó, amely pedig logikai (boolean) típusú. Mindkettő értékére hatással van az adott klón, hiszen értékadások szerepelnek, melyek bal oldalán ez a két változó fordul elő. A két kódrészlet a log4j nevű program forráskódjában a CategoryExplorerMode osztályban található.

```

119         while (children.hasMoreElements()) {
120             CategoryNode node = (CategoryNode) children.nextElement();
121             String title = node.getTitle().toLowerCase();
122
123             String pathLC = element.getTitle().toLowerCase();
124             if (title.equals(pathLC)) {
125                 categoryAlreadyExists = true;
126                 // This is now the new parent node.
127                 parent = node;
128                 break; // out of the while, and back to the for().
129             }
130         }

```

CloneInstance~198

A refaktoráló programom azonban érzékel még két előforduló változót, amelyek értékén esetlegesen változtathat egy függvény-hívás. Igaz, Java nyelvben ezek a függvény-hívások nem változtatnak a referencia által mutatott helyen, de a másik két változó visszatöltéséről is gondoskodni fogunk. Ebben az esetben létre kell hozni egy osztályt a visszatérési értékek kimentésének, fel kell venni adattagként ezt a 4 változót, majd a létrehozott új függvényen belül ezekkel hivatkozni a megváltozott értékekre. Létrehozunk még egy másik függvényt is, amely tudja kezelni a kifelé mutató „return”, „break”, és „continue” utasításokat. Egy adott break utasítást például a setBreak() függvénnyel kell kicserélni. Return utasítás esetén a setReturnValue() függvényt kell meghívnia a return helyén. Ennek egy paramétere a visszatérési érték. Ha void függvényről van szó, akkor ez az érték a null lesz. A két klón összeolvasztásából kapott új függvény hívása után a visszatérési értékeket visszatöltve meg kell vizsgálni, hogy a kifelé mutató utasítások közül meghívódott-e valamelyik, és ha igen, akkor melyik. Erre csak egy if utasítás kell, amely leellenőrzi a isReturn, isBreak, IsContinue utasításokat egyesével, de csak ha volt ilyen utasítás. Igaz, itt nem lesz használva ez a funkció, de a programunk ebben az esetben is használni fogja az osztályt.

```

1 package refactoring;
2 public class Refactored <T, M> {
3     public boolean isReturn;
4     public boolean isBreak;
5     public boolean isContinue;
6     public T return_value;
7     public M obj;
8     public Refactored(M newObj)
9     {
10         isReturn = false;
11         isBreak = false;
12         isContinue = false;
13         return_value = null;
14         obj = newObj;
15     }
16     public Refactored<T, M> setReturnValue(T ret_val)
17     {
18         isReturn = true;
19         return_value = ret_val;
20         return this;
21     }
22     public Refactored<T, M> setBreak()
23     {
24         isBreak = true;
25         return this;
26     }
27     public Refactored<T, M> setContinue()
28     {
29         isContinue = true;
30         return this;
31     }
32 }

```

Refactored osztály

A Refactored osztály segít a kimenő élek kezelésében. Ez egy generikus típus két paraméterrel: az első paramétere a visszatérési érték típusa, míg a másik paramétere a Segéd osztálynak a típusa. Mindkettőt meg kell adni neki, hiszen a visszatérési érték típusa a return utasításnál kell, a segéd osztály (NewSupClass) pedig a változók visszatöltésénél lesz használva. Minden egyes változóra van egy adattagja, illetve ezek közül csak azokat tölti fel konstruktorral, amelyek nem a klónon belül jöttek létre, hanem már rendelkeztek kezdeti rétékkal. A programom gondoskodik arról is, hogy azok a típusok, amelyeket a kódrészlet használ, be legyenek importálva, hiszen nélküle nem érné el őket. A Refactored és a NewSupClass osztály is a refactoring csomagba kerül.

```

1 package refactoring;
2 import java.util.Enumeration;
3 import org.apache.log4j.lf5.viewer.categoryexplorer.CategoryElement;
4 import org.apache.log4j.lf5.viewer.categoryexplorer.CategoryNode;
5 public class NewSupClass {
6     public boolean categoryAlreadyExists;
7     public CategoryNode parent;
8     public CategoryElement element;
9     public Enumeration children;
10    public NewSupClass(Enumeration children,
11                      CategoryElement element,
12                      CategoryNode parent,
13                      boolean categoryAlreadyExists)
14    {
15        this.children = children;
16        this.element = element;
17        this.parent = parent;
18        this.categoryAlreadyExists = categoryAlreadyExists;
19    }
20 }

```

NewSupClass osztály

Végül ejtenék egy pár szót a visszatöltésről is. Ebben az esetben csak az egyik klón példány helyét fogom bemutatni, mivel mind a két generált kód megegyezik ebben az esetben.

```

61 Refactored<CategoryNode, NewSupClass> ref1 = extractedMethod(children, element, categoryAlreadyExists, parent);
62 children = ref1.obj.children;
63 element = ref1.obj.element;
64 categoryAlreadyExists = ref1.obj.categoryAlreadyExists;
65 parent = ref1.obj.parent;

```

A klón példányok helye

Itt a függvény hívás visszatérési értéke a Refactored osztály, megfelelően felparaméterezve a tartalmazó függvény visszatérési értékének típusával, illetve a változókat tartalmazó új osztály típusával. Átadjuk mindegyik olyan paramétert, amelyre szüksége lesz a függvényünknek, a függvény hívása után pedig visszatöltjük a refaktorált osztályból a változók értékét. Abban az esetben, ha egy változó mindkét klónon belül egy osztályváltozóra mutat, akkor természetesen nem kell visszatöltéssel bajlódni, hiszen a függvény közvetlen változtatja az osztályváltozó értékét.

5.4. Kiértékelés

Készítettem 64 tesztet, melyeken tesztelni tudtam a programomat. Mindegyik tesztetem egy-egy konkrét esetre lett kiélezve. Összesen 7 darab tesztet nem futott le, mert ezek olyan eseteket fedtek le, melyekkel még nem foglalkoztam. A tesztesetek több csoportra oszthatóak. A return, break, continue lekezelését tesztelők, az egy, vagy annál több visszatérési értékkel rendelkezők. Vizsgálom azt is, hogy a program működjön akkor

is, ha a kódrészlet egy metóduson belül van, illetve akkor is, ha a klónok metódusok. Kell még vizsgálat arra is, hogy a klónok egy osztályon belül vannak-e, vagy több külön osztályban. Az eltérő értékű konstansokat is vizsgálni kell, mert arra is lett egy elegáns megoldás. Továbbá a készítettem teszteteket olyan esetekre, amikor a klónban található hibadobás (**throw**), illetve hibaelkapás (**catch**), bár erre még nem sikerült a megoldást elkészíteni. Továbbá kellenek még új tesztetek, amiknél a klónokban található egyes típusok eltérnek, viszont összevonhatók, mert létezik közös ősök, így megoldható az összeolvasztás egy generikus metódusban. Ezek az eseteken felül még belekerültek olyan tesztetek, melyek azt hivatottak bemutatni, hogy a program mennyire működik, ha ezen esetek kombinációi találhatók meg a klónokban.

A kiértékelés második fele egy nyílt forráskódú programon való tesztelése volt, ahol minden megtalált klónosztályra végigfuttattam a programomat. A projekt, amelyen teszteltem, a log4j[19] nevet viseli. A klónkereső program 34 klónosztályt talált meg, ezeket hat részre osztottam, programom refaktorálási képességei szerint. Első szint, az olyan klónosztályok voltak, amelyeket nem lehetett megszüntetni, mivel olyan tulajdonságokkal rendelkeztek, amelyek lehetetlenné tették refaktorálásukat. Ilyen például, ha különböző metódusokra hivatkoztak a klónosztályon belüli klónpéldányokban. Ebbe a csoportba 11 klónosztály került bele. A második csoport azok a klónosztályok, amelyeket lehetséges volt eliminálni, és a programom meg is birkózott vele gond nélkül. Ebből volt 9 darab. A harmadik az olyan klónosztályok, amelyek refaktorálhatóak lehetnének, de az analízisnél hiba lépett fel. Ez akkor fordulhat elő, amikor egy olyan változót keresek a gráfban, amely nincs benne. Ennek oka, hogy beépített függvényre vagy adattagra hivatkozunk, és ezeket az elemző nem mindig találja meg. Ebbe a csoportba 3 klónosztály tartozott. A negyedik az olyan osztályok csoportja, melyeknél az öröklődési hierarchiát kellett volna átformálni, hogy refaktorálható lehessen. Ez akkor fordul elő például, hogyha két külön osztályban vannak a klónok, amelyeknek létezik közös ősosztályuk, és annak egy felülírt változatát kellett volna meghívni, illetve az elkészített metódust a közös ősben elhelyezni (ez nem lehetséges, ha a közös ős beépített osztály). Ide tartozott öt klónosztály. Az ötödik, amikor a típusok tértek el az egyes változókból, vagy paraméterekben. Ezek akkor refaktorálhatóak, ha létezik közös ősosztályuk, és a klónon belül csak a közös ősosztály adattagjait használja. Itt is a közös ősben kell a metódust létrehozni. Ebből is található négy klónosztály a listában. Az utolsó csoport, amikor lehet refaktorálni, de a programom hiányosságai miatt nem tudta. Ez 2 klónosztálynál jött elő.

Összességében az általam implementált algoritmus 9 klón osztályban 18 klón példányt szervezett ki a 34 lehetséges klón osztályból. A fennmaradó klón osztályokból még 6 osztály automatikus refaktorálása az implementáció javításával elérhetővé válik. A változtatásokat manuálisan átvizsgálva megállapítottam, hogy a szükséges változók kimentése megtörtént, illetve az utasítások megfelelő sorrendben végrehajtásra kerülnek.

6. Korlátozások

A korlátozásoknak két fajtáját szeretném kiemelni. Létezik olyan korlátozás, ami azért korlátozza le a programomat, mert a hozzá tartozó rész még nem lett implementálva, de megvalósítható lehetne, míg a másik azért, mert nem lehet a refaktorálást úgy végrehajtani, hogy a program alap működése ne változzon meg a struktúra átformálásával. Ebben a fejezetben erről a két esetről ejtek szót, illetve a megvalósíthatók közül a megvalósítást is leírom. Azonban meg kell említeni, hogy míg nagyon sok eset megvalósítható, nem biztos, hogy érdemes is megvalósítani, mert csak még bonyolultabbá, kuszábbá alakítaná át a kódot. Ilyen eset például, ha túl sok változó módosul a klónon belül, és az összes ilyen változó kimentése megnehezítheti a kód olvashatóságát, nem beszélve arról, hogy amennyi helyet megtakarítanánk a példányok eltüntetésével, annyi vissza is jön a változók visszamentésével a függvény hívása után.

6.1. A programom korlátai

Az első, amit meg szeretnék említeni a programom hiányosságai között a kivétel dobásának a kezelése. Itt arra kell gondolni, hogy mikor a példányokon belül egy kivétel dobódhat, akkor két eset lehetséges: vagy le lett kezelve még a példányon belül egy catch utasítással, mely paraméterének típusa ősosztálya a dobott kivétel típusának, vagy a klónpéldányon kívül lesz majd kezelve valahol. Utóbbi esetben hibát jelezhet a fordító, hogyha refaktorálás után nem jelezzük a létrehozott függvény esetében, hogy egy bizonyos típusú kivétel dobása lehetséges (a függvény fejléce mellé, egy throws utasítás után fel kell sorolni, hogy milyen kivételek dobódhatnak a függvényen belül). Ez alapján véve nem lenne nagy feladat, de ismernünk kell hozzá az egész öröklődési hierarchiáját a használt típusoknak. Ha ez meg van, akkor a klónból kifelé haladba megnézhetjük, hogy el lesz-e kapva még a klónon belül a kivétel, és ha elérünk egy try blockhoz, a hozzá tartozó catch utasítások paramétereinek a típusát összehasonlítva a dobott kivétel típusával, már meg tudjuk határozni, hogy fel kell-e sorolni, vagy nem.

A másik hiányosság, amely kicsit talán megvalósíthatóságban átlóg a másik kategóriába (a megvalósíthatatlanba), az az eset, amikor a klónpéldányok bizonyos változói típusban eltérnek egymástól. Ezt egyszerűen úgy is vehetnénk, hogy lehetetlen végrehajtani rajtuk a megfelelő átalakítást, de léteznek olyan esetek, amikor lehetséges megoldást találni a problémára. Ehhez a problémához is ismernünk kell azonban az osztályok öröklődési hierarchiáját. Ha a két típusnak létezik közös őse, és a klónosztályban csak olyan metódusokat használnak, amelyek megtalálhatóak a közös ősből, akkor meg lehet oldani a refaktorálást. Ha azonban használ olyan függvényt, amely név szempontjából megegyezik ugyan a másik függvénnyel, de nem a közös ősből lett létrehozva, vagy példányosítás történik a klónon belül, akkor már nem lehet megoldani. Tételezzük most fel, hogy meg lehet oldani a refaktorálását egy ilyen példánynak. Ekkor szükségünk lesz arra, hogy az általunk létrehozott új metódust generikussá alakítsuk. Ehhez annyi generikus paramétert kell felvenni, ahány helyen különböznek a típusok. Meg kell adnunk még minden ilyen paraméterhez, hogy milyen típusokkal lehet példányosítani. Itt adhatunk egy alsó korlátot a megadható típusokra azzal, hogy a generikus paraméter neve mellé `extends` kulcsszóval megadjuk a közös ősöknek a típusát. Így az adott metódust csak olyan generikus paraméterrel lehet meghívni, amely a közös ősből lett származtatva, és az összes olyan metódust meg lehet rá hívni, amely a közös ősből elérhető. Amire ez nem megoldható, azt ki kell szűrni.

6.2. Esetek, amiket nem célszerű átalakítani

Ebbe a kategóriába tartoznak azok az esetek, amikor megvalósítható ugyan az összeolvasztás, de a végeredmény egy átláthatatlan forráskód lenne. Ide tartozik az előbb említett eset, amikor létezik olyan változó a klónpéldányokban, amelyek típusai különböznek klónpéldányonként, és vagy példányosítást tartalmaz az adott típusban, vagy olyan függvényre hivatkoznak, amely nem a közös ősből található. A következő eset, amikor a példányokban egy vagy több helyen eltér a függvény, amelyet meghívunk. Egy másik megvalósíthatatlan eset, amikor nem tudunk megfeleltetni minden klónpéldányban előforduló változót egy másik klónpéldányban levő változónak. A megfeleltetés azt jelenti ebben az esetben, hogy két külön klónpéldányban egy-egy változóra ugyan azokon a vezérlési szerkezeteken keresztül hivatkozunk.

7. Konklúzió

Dolgozatomban a szoftverfejlesztés egy aktuális megoldatlan problémájával, a kód másolatok kezelésének kérdésével foglalkoztam. A felvázolt problémákra egy automatikus refaktoráló eszközzel adtam választ.

A forráskód részletek több különböző helyre való átmásolása, habár látszólag növeli a produktivitást, komoly hibalehetőségeket rejt magában. Előfordulhat, hogy a másolt kód hibás, és ha észre is vesszük azt, megeshet, hogy nem írjuk át mindenhol. Megoldásként a fejlesztő manuálisan kiemelheti a másolt kódrészleteket egy metódusba, viszont ez a módszer sokáig tart, kimaradhatnak másolatok, illetve meg van a kockázat, hogy az átstrukturálás nem várt működést eredményezhet, mivel a fejlesztő hibázhat.

Kifejlesztettem egy eszközt, mely a hasonló kódrészleteket elemezve ellenőrzi a kiszervezéshez szükséges feltételeket, és azok teljesülése esetén automatikusan refaktorálja a programot. A másolatokból egy metódust készít, majd gondoskodik a metódus megfelelő meghívásáról az összes másolat helyéről, ezzel megtartva a program eredeti működését, de javítva annak minőségét. A teljes működéshez külső statikus program elemzőt, klón-detektáló és forráskód generáló modult is felhasználtam, így a rendszer teljes forráskód-forráskód átalakítást valósít meg. Az eszközt Java nyelvre készítettem el, de a módszer alkalmazható más objektum-orientált nyelvekre is. A kiértékelést a saját tesztesetek mellett egy nyílt forráskódú programon is elvégeztem, ahol a refaktorálható klónok egy nagy halmazát jól, a program működését nem befolyásoló átalakításokkal sikerült eliminálni. Ezt a refaktorálás előtti és utáni állapot manuális átnézésével állapítottam meg.

7.1. *Tervek további funkciókra*

Sok olyan ötlet van, amely a hatékonyabb programklón eliminálását teszi lehetővé, és ott van még a többi refactoring, amelyet lehet implementálni a programomba. Először a mostani programom fejlesztése a cél, illetve olyan átalakításra van szükség, amely lehetővé teszi a többi refaktoráló algoritmus beépítését a kódba. Ekkor a program bővíthetővé válik, és a refaktorálandó problémák megtalálásával párhuzamosan a módszerek is elkészíthetőek.

A programkód másolatok közül a változók típusban való eltérése, és a kivételkezeléssel kapcsolatos problémák megoldása a cél. A másik cél az, hogy az osztályok öröklődési

hierarchiájának figyelembevételével az eliminálás során létrehozott új metódust a lehető legjobb helyre tegyük, és lehetőségünk legyen a külön osztályban levő, ám a tartalmazó osztályok közös őssel való rendelkezése esetén a közös ősbe felmozgatni a metódust.

A többi refactoring közül két módszer implementálását tűztem még ki célul. Az egyik az úgynevezett **Switch Statement**[3] bad smellt eltüntető eljárás, amiről akkor beszélünk, hogyha a programkódban több helyen is előfordul olyan switch utasítás, melynek címkéi a többi switch utasítás címkéivel megegyeznek. Ekkor valószínűleg tervezési hibáról lehet szó. A problémát megoldani egy olyan komplex eljárással kell, melyben minden címkéhez létrehozunk egy új osztályt, és ennek egy közös ősosztálya az az osztály lesz, melynek metódusai tartalmazzák az refaktorálandó switch utasításokat. Az ősosztály példányosításakor a megfelelő gyerekosztályt példányosítjuk. Ezután minden switch utasítást kicseréljük egy absztrakt függvényhívásra, amely az ősosztályban lett létrehozva, de a gyerekosztályok felüldefiniálták, és minden gyerekosztály a függvényhez a switch adott címkéjének megfelelő utasításokat hajtja végre. Így minden címkéhez a megfelelő utasítások fognak végrehajtódni, és kihasználjuk az objektum orientáltság lehetőségeit.

A másik módszer az úgy nevezett **Data Clumps**-ok[3] refactoringja. Data Clumps-okról akkor beszélünk, hogyha a projekten belül több helyen is felfedezhetünk olyan változókat, amelyek együtt vannak használva. Ekkor felmerül a kérdés, hogy miért nem készítünk egy olyan osztályt, melynek az adattagjai típusban és névben megegyeznek az együtt használt változókkal. Létre kell hoznunk egy ilyen osztályt, és minden előfordulásukat egy ilyen típusú objektummal helyettesíteni.

Ha ezek a refactoringok megvannak, akkor további műveleteket is leimplementálhatunk. Azért pont ezt a két eljárást választottam, mert az ezekhez tartozó smellek jelenlétét viszonylag egyszerűen fel lehet fedezni.

8. Irodalomjegyzék

[1] Wikipédia

http://en.wikipedia.org/wiki/Software_metric (2012)

[2] Source Making

<http://sourcemaking.com/refactoring/bad-smells-in-code> (2012)

- [3] Source Making
<http://sourcemaking.com/refactoring> (2012)
- [4] Vidács László, Gogolla Martin és Rudolf Ferenc: From C++ Refactorings to Graph Transformations. In Electronic Communications of the EASST (ICGT 2006 Workshop Software Evolution through Transformations), volume 3, 2006.
- [5] Vidács László: Refactoring of C/C++ Preprocesszor Constructs at the Model Level: In Proceedings of ICSOFT 2009, 4th International Conference on Software and Data Technologies, 2009
- [6] Matthias Rieger, Stéphane Ducasse, Georges Golomingi: Tool Support for Refactoring Duplicated OO Code. ECOOP Workshops 1999: 177-178
<http://scg.unibe.ch/archive/papers/Rieg99aToolSuppRefacOOCode.pdf>
- [7] Martin Fowler: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
- [8] William F. Opdyke: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana–Champaign (1992)
- [9] Filip van Rysselberghe and Serge Demeyer - Evaluating Clone Detection Techniques from a Refactoring Perspective. [ASE 2004](#): 336-339
- [10] Yoshiki Higo, Shinji Kusumoto, Katsuro Inoue: A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system. Journal of Software Maintenance 20(6): 435-461 (2008)
- [11] Tom Mens and Tom Tourw: A survey of software refactoring. IEEE Transactions on Software Engineering, 30(2), 126–139. (2004)
- [12] Robert Ambrosius Tairas: Representation, Analysis, And Refactorings Techniques To Support Code Clone Maintenance, PhD thesis, University of Alabama at Birmingham, 2010
- [13] Cory Kapser & Michael W. Godfrey (2006). “Cloning Considered Harmful” Considered Harmful. In Proc. of the 13th Working Conference on Reverse Engineering (pp. 19–28).
- [14] IntelliJ IDEA
<http://www.jetbrains.com/idea/features/refactoring.html> (2012)
- [15] Eclipse JDT
http://www.ibm.com/developerworks/opensource/library/os-eclipse-refactoring/index.html?ca=dgr-lnxw07Refactoringdth-OS&S_TACT=105AGX59&S_CMP=grlnxw07 (2012)

- [16] Netbeans Refactoring
<http://wiki.netbeans.org/Refactoring> (2012)
- [17] Oracle JDeveloper
http://docs.oracle.com/cd/E25054_01/web.1111/b31974/refactoring.htm (2012)
- [18] Delphi
http://docwiki.embarcadero.com/RADStudio/en/What%27s_New_in_Delphi_and_C%2B%2BBuilder_2010 (2012)
- [19] log4j Projekt, 1999-2010 Apache Software Foundation
<http://logging.apache.org/log4j/1.2/> (2012)
- [20] Tibor Bakota, Rudolf Ferenc, and Tibor Gyimóthy. Clone Smells in Software Evolution. In Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM'07), Paris, France, October 2007.
- [21] Chanchal Kumar Roy, James R. Cordy, Rainer Koschke: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci. Comput. Program. 74(7): 470-495 (2009)
- [22] Rainer Koschke: Identifying and Removing Software Clones. Software Evolution 2008: 15-36
- [23] Refactoring Catalog, <http://www.refactoring.com> (2012)
- [24] Tibor Gyimóthy. To Use or Not to Use? The Metrics to Measure Software Quality (Developers' View). In Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09), 2009. keynote presentation.
- [25] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, Ettore Merlo: Comparison and Evaluation of Clone Detection Tools. IEEE Trans. Software Eng. 33(9): 577-591 (2007)
- [26] Duploc
<http://c2.com/cgi/wiki?DupLoc> (2012)

Köszönetnyilvánítás

Ez a TDK nem jöhetett volna létre a két témavezetőm, Dr. Ferenc Rudolf adjunktus, illetve Dr. Vidács László tudományos munkatárs közreműködése nélkül. Sokat segítettek a dolgozat megformálásában, és kérdéseimmel bármikor fordulhattam hozzájuk.

Köszönöm továbbá projektvezetőmnek, Dr. Fülöp Lajos Jenőnek, hogy ellátott ötleteivel, és biztosította számomra a megfelelő erőforrásokat, hogy a programomat működőképessé tehessem.

Megemlíteném még Muhi Kornélt, akinek a programom könnyen kezelhetőségéhez hozzájáruló eclipse plug-in fejlesztését köszönhetem.

Szót ejtenék még a Szoftverfejlesztés Tanszék, azon belül a Columbus csapat tagjairól, akik a segédprogramokat fejlesztették, és az ezzel kapcsolatos kérdéseimet megvitathattam velük.

Köszönöm csoporttársamnak, Almási Dénesnek, hogy megosztotta velem programját, amely jó alapul szolgált a későbbi fejlesztéseimnek.