Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Nejc Kišek

Večkratno razpošiljanje v Javi z obdelavo anotacij

MAGISTRSKO DELO

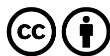
MAGISTRSKI PROGRAM DRUGE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2018

Avtorske pravice.

Naloga je na voljo pod pogoji licence *Creative Commons Attribution 4.0 International* (CC BY 4.0). To pomeni, da je uporabnikom dovoljeno reproduciranje, deljenje in predelovanje, pod pogojem, da navedejo avtorja izvirnega dela. Podrobnosti o licenci so na voljo na naslovu `https://creativecommons.org/licenses/by/4.0/`.



Pripadajoča izvorna koda je na voljo na `https://github.com/thenejcar/AnnotationDispatch`, pod pogoji licence *BSD 2-Clause*.

©2018 Nejc Kišek

Zahvala

Zahvaljujem se mentorju Juriju Miheliču, ki mi je z idejami, nasveti, popravki in komentarji pomagal pri izdelavi naloge.

Nejc Kišek, 2018

Kazalo

Povzetek

Abstract

1 Uvod in cilji naloge **1**

2 Razpošiljanje **3**

2.1 Osnovni pojmi 3

2.2 Razpošiljanje v Javi 7

2.3 Podobne rešitve 8

2.3.1 Multijava 8

2.3.2 Java MultiMethod Framework 9

2.3.3 JPred 9

2.3.4 Clojure 10

2.3.5 Julia 12

3 Anotacije in njihovo obdelovanje **15**

3.1 Anotacije 15

3.2 Obdelovanje anotacij 16

3.3 Inicializacija 17

3.4 Običajni mehanizmi obdelave anotacij 17

3.5 Abstraktno sintaksno drevo 18

3.6 Spreminjanje obstoječe kode 19

3.7 Simbolne tabele 21

3.8 Knjižnica Project Lombok 21

4 Implementacija **23**

4.1 Simuliranje večkratnega razpošiljanja 23

4.2 Uporaba knjižnice 24

4.3 Osnovni obdelovalec anotacij 26

4.4 Odločitveno drevo . . . 26
4.5 Odsevnost . . . 28
4.6 Večnivojski obiskovalec . . . 30
4.6.1 Uporaba obiskovalca za razpošiljanje . . . 31
4.6.2 Nivoji obiskovalca . . . 32
4.6.3 Iskanje po drevesu . . . 34
4.6.4 Generiranje kode za obiskovalca . . . 36

5 Primerjava različic knjižnice **39**
5.1 Generiranje testnih primerov . . . 39
5.2 Vrste in izvedba eksperimentov . . . 40
5.2.1 Vpliv števila parametrov . . . 41
5.2.2 Vpliv oblike razredne hierarhije . . . 41
5.2.3 Vpliv števila modelov metod v programu . . . 42
5.2.4 Vpliv števila primerkov modela . . . 42
5.3 Hitrost izvajanja . . . 43
5.4 Hitrost prevajanja . . . 44
5.5 Velikost prevedenih programov . . . 47
5.6 Rezultati primerjanja . . . 49

6 Zaključek in nadaljne delo **53**

Seznam uporabljenih kratic

kratica	angleško	slovensko
JVM	Java Virtual Machine	javanski navidezni stroj
JDK	Java Development Kit	zbirka orodij za razvoj v Javi
JMMF	Java Multi-Method Framework	ime knjižnice, predstavljene v razdelku 2.3.2

Povzetek

Naslov: Večkratno razpošiljanje v Javi z obdelavo anotacij

Razpošiljanje je mehanizem, s katerim se v objektno usmerjenih programskih jezikih razlikuje med metodami, ki imajo enako ime in število parametrov. Deluje tako, da ob klicu metode na podlagi dinamičnih tipov podanih parametrov izbere najustreznejšo metodo. Enojno razpošiljanje med izvajanjem za to uporabi le tip enega parametra (prejemnika), medtem ko se tipi ostalih ugotovijo med prevajanjem. Pri večkratnem razpošiljanju se tipe vseh parametrov ugotovi šele med izvajanjem, kar je počasnejše, a bolj fleksibilno. V Javi in mnogih drugih objektno usmerjenih programskih jezikih je podprto le enojno razpošiljanje, večkratnega pa lahko na različne načine simuliramo.

Cilj naloge je bil izdelati knjižnico, s pomočjo katere se na podlagi anotacij, ki jih dodamo v običajen javanski program, generira koda za simuliranje večkratnega razpošiljanja. Logika za razpošiljanje se generira šele med prevajanjem, zato je lahko poljubno zapletena, ne da bi vplivala na zapletenost izvorne kode. Pri uporabi te knjižnice so edina sprememba med pisanjem programa dodatne anotacije, kar je enostavnejše za uporabo kot podobne obstoječe rešitve, ki potrebujejo posebne prevajalnike ali pa zahtevajo klicanje metod na povsem drugačen način.

Knjižnica je izvedena v treh različicah, ki uporabljajo različne načine za simuliranje večkratnega razpošiljanja: s pomočjo odločitvenega drevesa z neposrednim preverjanjem tipov, z uporabo odsevnosti in z razširjenim načrtovalskim vzorcem obiskovalec. Od teh sta prva dva podobna načinom, ki so uporabljeni v obstoječih rešitvah, tretji pa je manj znan in o njem do sedaj ni mogoče najti podrobne literature. V sklopu naloge so predstavljene implementacije, pa tudi rezultati eksperimentalne primerjave teh treh načinov, kjer primerjamo hitrost izvajanja, hitrost prevajanja in velikost prevedene kode.

Ključne besede

razpošiljanje, večkratno razpošiljanje, anotacije, obdelava anotacij, generiranje kode, Java, javac

Abstract

Title: Multiple dispatch in Java using annotation processing

Dispatch is a mechanism in object-oriented programming languages used for distinguishing between methods with the same name and number of parameters. It works by examining the runtime types of parameters passed to a method call and selecting the most suitable method for them. Single dispatch selects a method based on the runtime type of just one of the parameters (the receiver), while the types of the rest of them are determined during compilation. Multiple dispatch determines the types of all the parameters during program execution, which is slower but more flexible. Java along with many other object-oriented languages supports single dispatch but not multiple dispatch. In such languages multiple dispatch can be simulated in different ways.

The goal of this thesis was to create a library that uses annotations added to an ordinary Java program in order to generate code for simulating multiple dispatch. Because the dispatch logic is generated during compilation, it can be very complex without complicating the source code. The only change when writing a program that uses our library are additional annotations, which is much simpler than existing similar solutions that use special compilers or change the way methods can be used.

There are three different versions of the library with three different mechanisms for simulating multiple dispatch: a decision tree with direct type inspection, a mechanism that uses reflection and an extended version of the visitor design pattern. The first two are similar to mechanisms used in other solutions while the third one is less known and is not found in the existing literature. In this thesis we show implementations of the three versions of our library and present the results of the experiments, where we compare them based on execution time, compilation time and size of the generated code.

Keywords

dispatch, multiple dispatch, annotations, annotation processing, generating code, Java, javac

Poglavje 1

Uvod in cilji naloge

Večkratno razpošiljanje je eden od načinov izvajanja metod, pri katerem se za izbiro ustrezne metode med izvajanjem preverjajo vrednosti oziroma tipi vseh njenih parametrov. Večina priljubljenih objektno usmerjenih programskih jezikov, kot so C++, Java in C#, večkratnega razpošiljanja ne podpira in se namesto tega zanaša na enojno razpošiljanje oziroma navidezne metode, kjer se med izvajanjem preverja le vrednost prejemnika klica metode. Če programer želi uporabiti večkratno razpošiljanje, ga mora tako simulirati znotraj samega programa.

Obstaja več načinov, s katerimi lahko večkratno razpošiljanje v Javi simuliramo, v ta namen pa je bilo izdelanih že več knjižnic in razširitev jezika. Dve težavi, ki se pojavljata pri obstoječih rešitvah, sta, da zanje potrebujemo drug prevajalnik ali pa da je treba spremeniti način deklariranja metod ali njihovega klicanja.

Eden od ciljev te naloge je razvoj knjižnice, s katero lahko v Javo z uporabo anotacij dodamo večkratno razpošiljanje. Glavna prednost takega pristopa je, da so pri njeni uporabi spremembe v izvorni kodi minimalne, vsa potrebna koda za razpošiljanje pa se generira med prevajanjem z mehanizmi za obdelovanje anotacij, ki so podprti v prevajalniku `javac`. Tako je lahko logika za razpošiljanje zelo zapletena, ne da bi vplivala na enostavnost uporabe knjižnice.

Drugi cilj te naloge je predstavitev in primerjava različnih pristopov za izvedbo razpošiljanja, zaradi česar bo knjižnica izvedena v treh različicah z različnimi načini simuliranja razpošiljanja, ki jih lahko nato eksperimentalno primerjamo med seboj.

V naslednjem poglavju so predstavljeni koncepti razpošiljanja in razlike med enojnim in večkratnim razpošiljanjem. Poleg tega je opisana tudi izvedba enojnega razpošiljanja v JVM in izvedbe večkratnega razpošiljanja v jeziku MultiJava, razširitvi JPred, knjižnici JMMF ter jezikih Clojure in Julia.

Tretje poglavje opisuje anotacije v Javi in mehanizme za njihovo obdelavo. Najprej je opisano delovanje obdelovalcev anotacij in običajni mehanizmi, ki se v njih uporabljajo na primer za generiranje dokumentacije ali testov. Nato so opisani tudi manj uporabljeni mehanizmi, s kate-

rimi lahko spreminjamo abstraktno sintaksno drevo programa med prevajanjem. Predstavljenih je nekaj razredov in metod, ki jih za to potrebujemo in ki so uporabljeni tudi v implementaciji naše knjižnice za večkratno razpošiljanje.

Četrto poglavje vsebuje opise treh načinov za simuliranje večkratnega razpošiljanja, ki so uporabljeni v treh različicah izdelane knjižnice: prvi temelji na odločitvenem drevesu z neposrednim preverjanjem tipov parametrov, drugi uporablja mehanizme odsevnosti (angl. *reflection*), tretji pa je razširjena oblika načrtovalskega vzorca obiskovalec. Za vse tri načine je opisana tudi implementacija obdelovalca anotacij, ki generira njihovo kodo v naši knjižnici.

V petem poglavju so prikazani rezultati eksperimentalnega primerjanja treh različic knjižnice, kjer upoštevamo hitrost razpošiljanja, hitrost prevajanja in velikost generirane kode. Za potrebe tega testiranja je bil izdelan tudi generator primerov, s katerim lahko opazujemo vpliv različnih lastnosti, kot so na primer število parametrov metode, število metod ali oblika razredne hierarhije parametrov, na vsakega od treh testnih indikatorjev. Ob vseh rezultatih so podane tudi njihove razlage in komentarji.

Poglavje 2

Razpošiljanje

V tem poglavju si bomo pogledali nekaj osnovnih konceptov, ki jih bomo potrebovali v nadaljevanju naloge. Najprej bomo opisali, kaj je razpošiljanje, kakšna je razlika med enojnim in večkratnim razpošiljanjem in kako deluje enojno razpošiljanje v jeziku Java. Nato si bomo pogledali, kakšni so obstoječi načini, da v Javo dodamo večkratno razpošiljanje in kako je to implementirano v nekaterih drugih jezikih.

2.1 Osnovni pojmi

Izraz razpošiljanje (angl. *dispatch*) prihaja iz koncepta izvajanja operacij preko pošiljanja sporočil z imenom operacije in dodatnimi parametri (angl. *message passing*). Tak pristop se v objektno usmerjenih jezikih uporablja namesto starejšega načina, pri katerem ob klicu funkcije podamo le naslov v pomnilniku, kjer se nahaja njena koda. Razpošiljanje je postopek, s katerim poskrbimo, da poslano sporočilo pride do najustreznejše operacije, ki se nato lahko izvede. V jeziku Java, na katerega se bomo osredotočali v tej nalogi, je pošiljanje sporočil izvedeno v obliki klicanja metod, razpošiljanje pa je način izbiranja prave metode. Tako v Javi pravega pošiljanja sporočil v obliki, kot ga najdemo na primer pri jezikih Smalltalk ali Objective C, ni, vendar so postopki, ki se izvedejo v ozadju zelo podobni – zato tudi pri Javi in podobnih jezikih uporabljamo izraz razpošiljanje.

Pri statičnem razpošiljanju (angl. *static dispatch*) klic metode ali funkcije razreši že prevajalnik. To lahko stori le, če obstaja samo ena možna operacija, na katero se sporočilo nanaša, ne glede na stanje preostalih delov programa ali katerekoli druge dejavnike. Do pravega razpošiljanja tako pravzaprav ne pride, saj se zgodi implicitno med prevajanjem, zato se tudi sam izraz statično razpošiljanje redko uporablja.

Dinamično razpošiljanje (angl. *dynamic dispatch*) se izvaja takrat, ko na izbiro ustrezne

operacije poleg podanega imena vplivajo tudi dodatni parametri, katerih vrednosti pa običajno ni mogoče predvideti pred izvajanjem. Vnaprejšnje razreševanje ni mogoče zaradi uporabe polimorfizma s podtipi (angl. *subtype polymorphism*), zaradi katerega lahko vrednosti nekega tipa vsebujejo tudi objekte vseh njegovih podtipov.

Za primer vzemimo program, kjer ustvarimo spremenljivko z določenim tipom, nato pa tekom programa vanjo shranimo neko vrednost. Tip spremenljivke ob definiciji imenujemo tudi *statični* tip, tip vrednosti, ki je v spremenljivki shranjena, pa *dinamični* tip. Statični tip se določi ob definiciji spremenljivke in ostane ves čas enak, dinamični pa se določi vsakič, ko v spremenljivko shranimo novo vrednost in se torej lahko med izvajanjem spreminja. Zaradi polimorfizma sta ta dva tipa lahko različna, če je dinamični izpeljan iz statičnega, saj objekte lahko shranjujemo v spremenljivke njihovega nadtipa. Dinamičnih tipov tako prevajalnik ne pozna, saj jih lahko določimo šele med samim izvajanjem.

Nepoznavanje dinamičnega tipa neke vrednosti lahko privede do dveh vrst dvoumnosti pri klicanju metod, ki jih posledično ni mogoče razrešiti med prevajanjem:

- Metodo, ki je definirana znotraj nekega tipa, lahko na novo definiramo v tipu, ki je iz njega izpeljan, kar imenujemo prekrivanje (anlg. *overriding*). Če tako večkrat definirano metodo pokličemo na spremenljivki, za katero prevajalnik ne ve, ali vsebuje objekt z osnovnim ali izpeljanim tipom, prav tako ne more vedeti, ali je treba izvesti osnovno ali izpeljano metodo. Podobno se zgodi, če je tip spremenljivke vmesnik (angl. *interface*), le da osnovne metode tam sploh ni – prevajalnik ve, katere metode so na voljo, ne ve pa, katera od implementacij se bo izvedla.

- V programu lahko definiramo več operacij z istim imenom, kar imenujemo preoblaganje (angl. *overloading*). Kadar imata dve taki operaciji enako število parametrov in se torej razlikujeta le po njihovih tipih, bomo med njima razlikovali glede na to, kakšni parametri so bili ob klicu podani. Če za enega izmed podanih parametrov prevajalnik ne pozna dinamičnega tipa, ne more z gotovostjo vedeti, katera od dveh možnih operacij naj se izvede.

Tako je treba vsakič, ko pride do klica, preveriti trenutne tipe oziroma vrednosti uporabljenih parametrov in na podlagi njih izbrati ustrezno operacijo. Ta postopek, ki se izvede tekom programa, imenujemo dinamično razpošiljanje ali pa kar samo razpošiljanje.

Izraza enojno in večkratno razpošiljanje (angl. *single*, *multiple dispatch*) se nanašata na število parametrov, ki se pri dinamičnem razpošiljanju uporabijo za iskanje ustrezne operacije. Večina priljubljenih objektno usmerjenih jezikov (npr. Java in C#) podpira le enojno razpošiljanje, kjer pri dinamičnem razpošiljanju med izvajanjem programa sodeluje le eden izmed parametrov. Ta parameter se imenuje prejemnik (angl. *receiver*) in predstavlja razred, v katerem se operacija nahaja ter ga običajno pišemo pred ime metode, medtem ko so ostali parametri v oklepajih (na primer `prejemnik.metoda(parametri)`), včasih pa se ga lahko tudi izpusti.

Primer uporabe enojnega razpošiljanja je prikazan v izseku javanske kode 2.1, kjer je najprej definiran osnovni razred `O`, ki vsebuje metodo `metoda`, iz njega pa sta nato izpeljana dva razreda `A` in `B`, ki osnovno metodo prekrijeta. Ko s klicem `O obj = new A()` ustvarimo spremenljivko `obj`, je njen statični tip osnovni razred `O`, njen dinamični tipa pa razred `A`. Ob klicu `obj.metoda()` enojno razpošiljanje ugotovi dinamični tip prejemika `obj` in zato izvede metodo iz razreda `A`. Če ne bi izvajali enkratnega razpošiljanja, bi se prevajalnik lahko zanašal le na statične tipe in tako ob klicu `obj.metoda()` določil, da gre za klic osnovne metode.

```
class Primer1 {
    static class O {
        String metoda(){ return "Osnovna metoda"; }
    }
    static class A extends O {
        @Override
        String metoda(){ return "Metoda iz A"; }
    }
    static class B extends O {
        @Override
        String metoda(){ return "Metoda iz B"; }
    }

    public static void main(String[] args) {
        O obj = new A();
        obj.metoda(); // izvede se metoda iz A
    }
}
```

Izsek kode 2.1: Primer, ki uporablja enojno razpošiljanje.

Pri večkratnem razpošiljanju se poleg prejemnika za določitev ustrezne metode uporabi tudi ostale parametre. V Javi in mnogih drugih programskih jezikih to ni podprto, kar lahko ilustriramo z izsekom kode 2.2, kjer bi se z večkratnim razpošiljanjem program izvedel drugače. Na začetku programa je definirana enaka razredna hierarhija kot pri prejšnjem primeru, nato pa tri statične metode z enakim imenom in različnimi kombinacijami tipov parametrov.

```
class Primer2 {
    static class O {}
    static class A extends O {}
    static class B extends O {}

    static String metoda(O x, O y) { return "Osnovna metoda"; }
    static String metoda(A x, B y) { return "Metoda 2"; }
    static String metoda(B x, A y) { return "Metoda 3"; }

    public static void main(String[] args) {
        O arg0 = new A();
        O arg1 = new B();
        metoda(arg0, arg1); // izvede se osnovna metoda
    }
}
```

Izsek kode 2.2: Primer, kjer bi potrebovali večkratno razpošiljanje.

V prikazanem programu ustvarimo dve spremenljivki tipa `O` in vanju shranimo objekta tipa

`A` in `B`. Če bi Java podpirala večkratno razpošiljanje, bi se ob sledečem klicu sprožil postopek iskanja metode, ki se najbolj ujema z dinamičnimi tipi podanih parametrov in izvedla bi se druga metoda. Ker večkratnega razpošiljanja ni, se ta klic v celoti razreši že med prevajanjem, pri čemer prevajalnik uporabi statične tipe. V prikazanem primeru imata obe spremenljivki statični tip `O`, zato prevajalnik izbere osnovno metodo.

Posebna oblika večkratnega razpošiljanja, kjer poleg prejemnika upoštevamo samo še en dodaten parameter, se imenuje dvojno razpošiljanje (angl. *double dispatch*). Poseben je zato, ker običajno zadostuje potrebam po večkratnem razpošiljanju, njegova izvedba pa je veliko enostavnejša kot razpošiljanje glede na poljubno število parametrov. Zanj se pogosto uporablja načrtovalski vzorec obiskovalec (angl. *visitor design pattern*), pri katerem obiskovalec obiskuje spremenljivke osnovnega tipa, ob vsakem obisku pa glede na dinamični tip vrednosti izvede eno izmed vnaprej določenih operacij. Primer uporabe so obdelovanje drevesnih struktur, odzivanje na različne vrste dogodkov in podobni primeri, kjer imajo metode le po en parameter. Več podrobnosti o vzorcu obiskovalec je opisanih v razdelku 4.6 in v sorodni literaturi [4, 14].

Danes večina programskih jezikov večkratnega razpošiljanja ne podpira, kar je posledica kombinacije redkih primerov uporabe, kjer ne zadostuje vzorec obiskovalec, ter dodatnih zapletov in posledičnih upočasnitev, ki jih večkratno razpošiljanje povzroči ob izvajanju metod. Najdemo ga predvsem v starejših funkcijskih jezikih, kot je Common Lisp, spet pa se pojavlja v nekaterih novih jezikih, kot sta Clojure in Julia, ki si ju bomo ogledali v nadaljevanju. Za mnogo primerov, kjer bi potrebovali večkratno razpošiljanje, v nekaterih programskih jezikih lahko uporabimo tudi ujemanje vzorcev (angl. *pattern matching*), ki prav tako spremeni potek programa na podlagi trenutnih vrednosti med izvajanjem, vendar je sama uporaba bolj podobna stikalu (angl. *switch*).

Operacije, pri katerih je večkratno razpošiljanje koristno, so na primer interakcije med različnimi vrstami objektov. Taki primeri so prekrivanje grafičnih elementov ali trki različnih teles, kjer so objekti podani z osnovnim tipom, glede na podtip pa nato uporabimo različne algoritme.

```
interface Oblika {...}
class Pravokotnik implements Oblika {...}
class Krog implements Oblika {...}
class Elipsa implements Oblika {...}

class PovrsinaPreseka {
    public int izracunajPresek(Oblika a, Oblika b, Oblika c) {
        // veckratno razposiljanje izbere ustrezno metodo presek
        return presek(a, b, c);
    }

    private int presek(Pravokotnik a, Pravokotnik b, Pravokotnik c) {...}
    private int presek(Krog a, Krog b, Krog c) {...}
    private int presek(Pravokotnik a, Krog b, Elipsa c) {...}
}
```

Izsek kode 2.3: Primer uporabe večkratnega razpošiljanja.

V izseku kode 2.3 vidimo primer programa, pri katerem z metodo `izracunajPresek` želimo

izračunati površino preseka med tremi oblikami, ki so podani kot objekti tipa `Oblika`. Glede na podane oblike bo treba uporabiti različne algoritme, ki se med seboj precej razlikujejo, zato je smiselno, da vsakega zapišemo kot svojo metodo. Večkratno razpošiljanje bi ob klicu metode `presek` glede na podane parametre izbralo ustrezno implementacijo, v programskem jeziku, ki tega ne podpira, pa mora preverjanje tipov in iskanje ustrezne metode napisati programer.

2.2 Razpošiljanje v Javi

Vsako metodo v Javi je mogoče prepoznati po njenem imenu, razredu, v katerem se nahaja, številu parametrov in tipih teh parametrov. Dve metodi sta torej različni, če se razlikujeta v vsaj eni izmed teh lastnosti, jezik pa mora poskrbeti, da se bo ob vsakem klicu izvedla prava izmed njih. Java podpira enojno razpošiljanje, kar pomeni, da se med izvajanjem metode izbira le glede na dinamični tip prejemnika, za ostale parametre pa se uporabi statične tipe, ki jih je ugotovil prevajalnik. Klic take metode se v JVM izvede s pomočjo ukaza `invokevirtual` [1], ki mu je treba podati referenco na razred in referenco na metodo. Referenca na metodo se določi med prevajanjem, pri čemer se upošteva vse, kar prevajalnik lahko izve iz klica metode – njeno ime ter statični tipi parametrov. Referenca na razred se lahko določi šele med izvajanjem, saj jo je treba prebrati iz trenutne vrednosti prejemnika. Izvajalno okolje lahko s tema dvema referencama poišče pravi razred, nato pa znotraj njega še ustrezno metodo.

Eden od načinov za izvedbo tega postopka, ki ga uporablja navidezni stroj HotSpot [2], je preko tako imenovane razpošiljevalne tabele ali tabele navideznih metod (angl. *dispatch table* ali *virtual method table*), v kateri vsak razred hrani reference na svoje metode. Ko razred izpeljemo iz nekega obstoječega razreda, od svojega starša dobi tudi razpošiljevalno tabelo, kjer nato nekatere od starševih vnosov prepiše s svojimi na novo implementiranimi metodami. Tako so vsi vnosi, ki so prisotni v starševi tabeli, prisotni tudi v izpeljani, le da nekateri kažejo na starševe metode, nekatere pa na novo definirane. Referenca metode, ki jo prevajalnik poda ukazu `invokevirtual`, označuje eno izmed polj v tabeli razreda, ki je statični tip prejemnika. To polje bo v tabeli prisotno tudi, če je dinamični tip prejemnika eden od njegovih podtipov, vendar bo morda kazalo na njegovo prekrito metodo. Referenca razreda, podana ukazu `invokevirtual`, določa, kateri je pravi (dinamični) tip prejemnika, s čimer pove, katero izmed razpošiljevalnih tabel je potrebno upoštevati. Enojno razpošiljanje v Javi torej poteka tako, da se iz trenutnega tipa prejemnika razbere prava razpošiljavalna tabela, nato pa se znotraj nje s pomočjo reference metode, ki je bila razrešena že med prevajanjem, izbere pravi vnos.

Pravega večkratnega razpošiljanja, kjer razrešimo prejemnika in vse ostale parametre, Java tako kot mnogi ostali popularni objektno usmerjeni programski jeziki ne podpira. Če ga želimo uporabljati, ga moramo implementirati sami, na primer z mehanizmi opisanimi v poglavju 4, lahko pa uporabimo katero od obstoječih razširitev ali knjižnic, ki so opisane v nadaljevanju.

2.3 Podobne rešitve

V tem razdelku bomo opisali nekaj obstoječih rešitev, ki v Javo dodajo večkratno razpošiljanje, in jezike, v katerih je to že vgrajeno. Pogledali si bomo, kako je večkratno razpošiljanje pri vsaki našteti rešitvi izvedeno, te koncepte primerjali s knjižnico izdelano v sklopu te naloge in našteli njihove morebitne prednosti ter slabosti.

2.3.1 Multijava

MultiJava [5, 6] je razširitev Jave, ki v jezik doda multimetode – metode, ki uporabljajo večkratno razpošiljanje. Preostala koda je enaka kot pri običajni Javi (vsaj do verzije 1.3) in tudi metode definirane na običajen način se še vedno obnašajo, kot bi se sicer. MultiJava ima svoj prevajalnik, ki kodo prevede v datoteke `.class`, ki se lahko izvajajo na običajnem javanskem navideznem stroju. Implementacija večkratnega razpošiljanja, ki se zgodi v ozadju, je zelo podobna načinu z odločitvenim drevesom z neposrednim preverjanjem tipov, ki je opisan v poglavju 4.

V primerjavi z rešitvijo, opisano v tej nalogi, je prednost MultiJave, da poleg razpošiljanja glede na tipe parametrov omogoča tudi upoštevanje njihovih vrednosti. Tako lahko na primer napišemo eno metodo, ki kot parameter sprejema celo število, in drugo metodo, ki se izvede namesto nje, če je to število 5. Slabost MultiJave je, da ne podpira konstruktov iz novejših verzij Jave in da je popolnoma nov jezik, kar oteži vključitev v obstoječe projekte in delo z orodji, ki so sicer namenjena za Javo.

```
class MultiJavaRazred {

    static class O {}
    static class A extends O {}
    static class B extends O {}

    static String metoda(O x, O y, int i) { return "Osnovna metoda"; }
    static String metoda(O@A x, O@B y, int i) { return "Metoda 2"; }
    static String metoda(O@A x, O@B y, int@@5 i) { return "Metoda 3"; }

    public static void main(String[] args) {
        O arg0 = new A();
        O arg1 = new B();
        metoda(arg0, arg1, 15); // izvede se metoda 2
        metoda(arg0, arg1, 5);  // izvede se metoda 3
    }
}
```

Izsek kode 2.4: Primer programa v jeziku MultiJava.

Izsek kode 2.4 prikazuje primer multimetode s tremi primerki. Od običajne javanske kode jo ločita operatorja `@` in `@@`, s katerima ob definiciji metode označimo zahtevan tip oziroma vrednost parametra.

2.3.2 Java MultiMethod Framework

Java MultiMethod Framework (krajše JMMF) [7, 8] je knjižnica, s katero lahko v javanski program dodamo večkratno razpošiljanje. Knjižnico uporabimo tako, da najprej ustvarimo razred `MultiMethod`, ki mu podamo razred, kjer se naše metode nahajajo, ime metod in število njihovih parametrov. Knjižnica bo poiskala vse metode, ki ustrezajo opisu in si jih zapomnila. S klicem metode `invoke` na razredu `MultiMethod`, ki ji podamo prejemnika in ostale parametre, lahko nato sprožimo razpošiljanje – to bo glede na parametre poiskalo in poklicalo najustreznejšo od prej najdenih metod.

```
class JMMFRazred {
    static class O {}
    static class A extends O {}
    static class B extends O {}

    static String metoda(O x, O y) { return "Osnovna metoda"; }
    static String metoda(A x, B y) { return "Metoda 2"; }
    static String metoda(B x, A y) { return "Metoda 3"; }

    public static void main(String[] args) {
        Object arg0 = new A();
        Object arg1 = new B();

        MultiMethod mm = MultiMethod.create(JMMFRazred.class, "metoda", 2);
        mm.invoke(null, new Object[]{arg0, arg1}) // izvede se metoda 2
    }
}
```

Izsek kode 2.5: Primer uporabe knjižnice JMMF.

Program 2.5 prikazuje primer iz izseka kode 2.2, izveden s knjižnico JMMF, kjer je klic metode izveden preko zgoraj omenjenega razreda `MultiMethod`.

V ozadju se iskanje metod in razpošiljanje izvede s pomočjo odsevnosti na način, ki je zelo podoben naši implementaciji razpošiljanja z odsevnostjo iz poglavja 4. Pri obeh pristopih se pred začetkom razpošiljanja s pomočjo odsevnosti poišče vse ustrezne metode, ki se shranijo v tabelo. Ob vsakem klicu se v tabeli metod poišče najprimernejšo, ki se jo z mehanizmi odsevnosti nato tudi izvede. Prednost naše rešitve je v tem, da se koda za grajenje tabele z metodami, njihovo iskanje in izvajanje programu doda šele med prevajanjem, pri knjižnici JMMF pa je treba spremeniti način programiranja in se ves čas zavedati večkratnega razpošiljanja.

2.3.3 JPred

JPred [9, 10] je razširitev jezika Java, ki podpira tako imenovano predikatno razpošiljanje (angl. *predicate dispatch*). To pomeni, da podobno kot MultiJava poleg tipov pri razpošiljanju upošteva tudi vrednosti parametrov, dodatno pa omogoča razpošiljanje glede na kakršenkoli pogoj, ki ga k metodi dodamo s ključno besedo `when`. V pogoju lahko preverjanje tipov parametrov izvedemo z operatorjem `@`, lahko pa vključuje poljuben izraz, ki vrača vrednost tipa `boolean`. Primer

programa 2.6 je izveden s pomočjo razširitve JPred. Delovanje programa je navzven videti enako kot v primeru 2.5 s knjižnico JMMF.

```
public class JPredRazred {
    static class O {}
    static class A extends O {}
    static class B extends O {}

    static String metoda(O arg0, O arg1) {
        return "Osnovna metoda";
    }
    static String metoda(O arg0, O arg1) when arg0@A && arg1@B {
        return "Metoda 2";
    }
    static String metoda(O arg0, O arg1) when arg0@B && arg1@A {
        return "Metoda 3";
    }

    public static void main(String[] args) {
        O arg0 = new A();
        O arg1 = new B();

        metoda(arg0, arg1); // poklicala se bo metoda 2
    }
}
```

Izsek kode 2.6: Primer programa z JPred razširitvami.

JPred uporablja svoj poseben prevajalnik, ki program najprej prevede v datoteko `.java`, ki jo je treba nato prevesti z običajnim javanskim prevajalnikom, preden lahko program poženemo. Prednost takega pristopa je, da kot rezultat dobimo javanski program s predikatnim razpošiljanjem, ni pa nam treba kode za razpošiljanje pisati ročno. Poleg tega, da si s tem prihranimo čas, nam JPred med prevajanjem tudi zazna napake, ki izhajajo iz neujemanja tipov ali napačne uporabe multimetod, ki bi jih z ročno implementacijo razpošiljanja težko zaznali. Ker kot izhod iz prevajalnika dobimo javansko kodo, lahko vidimo, kako je razpošiljanje implementirano – deluje zelo podobno, kot razpošiljanje z odločitvenim drevesom, opisano v poglavju 4, oziroma razpošiljanje, ki ga uporablja MultiJava.

Od knjižnice, implementirane v sklopu te naloge, se JPred razlikuje v tem, da uporablja drugačno sintakso in ga zato ni možno vstaviti v obstoječe javanske programe. To sicer lahko storimo v dveh korakih – del kode napišemo v jeziku JPred, ki ga prevedemo v Javo in nato vstavimo v javanski program.

2.3.4 Clojure

Clojure [11] je novejši funkcijski programski jezik, ki izhaja iz jezika Lisp. Z njim lahko delamo v ukazni lupini, kjer se prevajanje izvaja za vsak ukaz sproti, lahko pa ga prevedemo v `.class` datoteke za izvajanje na JVM, kot bi bil napisan v Javi. Obstajajo tudi različice, ki se prevajajo v Javascript ali v format za izvajanje v okolju *Common Language Runtime*. Clojure je strogo

in dinamično tipiziran jezik, do neke mere pa podpira tudi statično tipiziranje, da omogoča kompatibilnost z obstoječimi javanskimi knjižnicami.

Za nas je zanimiv, ker podpira večkratno razpošiljanje preko multimetod. Kriterij za razpošiljanje definiramo pri vsaki multimetodi posebej in ni nujno samo tip parametrov, ampak je lahko na primer tudi vrednost, element objekta ali vnos v slovarju.

```
(defn -main []
    (derive ::A ::O)
    (derive ::B ::O)

    (defmulti metoda (fn [arg0 arg1] [(:tip arg0) (:tip arg1)]))

    (defmethod metoda [::O ::O] [arg0 arg1] (str "Osnovna metoda"))
    (defmethod metoda [::A ::B] [arg0 arg1] (str "Metoda 2"))
    (defmethod metoda [::B ::A] [arg0 arg1] (str "Metoda 3"))

    (def a { :tip ::A })
    (def b { :tip ::B })

    (metoda a b)  ; izvede se Metoda 2
    (metoda b a)  ; izvede se Metoda 3
    (metoda a a)  ; izvede se Osnovna metoda
)
```

Izsek kode 2.7: Primer multimetode v jeziku Clojure.

V programu, ki je prikazan v izseku kode 2.7, smo definirali dedovanje med vrednostmi `::A`, `::B` in `::O`, ki so bile nato uporabljene v slovarjih pod ključem `:tip`. Definirali smo multimetodo `metoda`, ki smo ji podali pravilo, na podlagi katerega se bo izvajalo razpošiljanje: iz dveh podanih argumentov se prebere vrednost `:tip`. Nato smo definirali tri primerke te multimetode na podoben način kot pri prejšnih primerih. Ustvarili smo tudi dva slovarja `a` in `b` z različnima vrednostma za ključ `:tip`, ki smo ju multimetodi s klicem podali v različnih vrstnih redih, razpošiljanje pa je poskrbelo, da se je vsakič poklical pravi izmed primerkov.

Ker se Clojure prevaja v `.class` datoteke, ga lahko z obratnim prevajanjem (angl. *decompile*) pretvorimo v javansko kodo in tako vidimo, kako bi isti program izgledal implementiran v Javi. Iz programa 2.7 na tak način dobimo kodo, sestavljeno iz več razredov:

- Razredi za zagon programa, ki naložijo knjižnice iz paketa `clojure.lang`, vzpostavijo okolje in nato pokličejo funkcijo `main`. Ti za opis razpošiljanja niso zanimivi.
- Razred, ki predstavlja funkcijo `main` in vsebuje vso kodo, ki v njej nastopa. Na začetku razreda so konstante, ki predstavljajo vrednosti `::O`, `::A` in `::B`, konstanta za ključno besedo `:tip`, za vse definirane spremenljivke, njihove vrednosti in tako dalje.

  Konstantam sledi telo metode, v kateri se najprej dvakrat pokliče funkcija `derive`, s čimer se v globalno hierarhijo tipov dodajo povezave med `::A`, `::B` in `::O`. Nato se ustvari multimetoda, v katero se dodajo vsi trije primerki. Sledi inicializacija slovarjev `a` in `b` ter trije klici prej ustvarjenih metod.

- Razred za razpošiljevalno funkcijo `(fn [arg0 arg1] [(:tip arg0)(:tip arg1)])`, ki smo jo podali definiciji multimetode. Ta vsebuje zelo enostaven algoritem, ki iz dveh podanih parametrov zgradi vektor njunih vrednosti `:tip`.
- Trije razredi, od katerih vsak predstavlja enega od primerkov metode in vsebujejo kodo, ki se bo izvedla ob njihovem klicu.

Ob izvedbi programa se nato ob vsakem klicu multimetode, ki smo ji podali oba parametra, razpošiljanje zgodi v treh korakih:

1. Multimetoda pokliče razpošiljevalno funkcijo, ki iz parametrov zgradi vektor njihovih tipov. Za klice iz našega primera so to vektorji `[::A, ::B]`, `[::B, ::A]` in `[::A, ::A]`.
2. Multimetoda na podlagi vektorja poišče najprimernejšo metodo. To stori tako, da se sprehodi čez vnose v tabeli metod, za vsako pa primernost ocenjuje na podlagi hierarhije, ki smo jo opisali z ukazi `derive`. Kombinacija najdene metode in vektorja se shrani v predpomnilnik, da se ob naslednjem klicu z enakim vektorjem ta korak lahko izpusti.
3. Najdena metoda se izvede s podanimi parametri.

Način izvajanja multimetod je precej podoben kot pri knjižnici JMMF ali pri načinu z uporabo odsevnosti, ki je opisan v poglavju 4. Razlikujeta se po tem, da v JMMF in naši implementaciji razpošiljamo samo glede na tip, v Clojure pa z razpošiljevalno funkcijo, ki dovoljuje več svobode. Poleg tega je nekaj razlik v učinkovitosti – Clojure predpomni najdene metode za pohitritev izvajanja, pa tudi sam mehanizem za izvedbo metod je drugačen. Pri uporabi odsevnosti v Javi je klic preko metode `invoke()` neprimerno počasnejši od običajnega klica, medtem ko se v jeziku Clojure najdena metoda izvede na enak način kot vsaka druga. S tem je razlika med uporabo običajnih metod in metod z večkratnim razpošiljanjem v Clojure precej manjša kot razlika med običajnimi metodami in razpošiljanjem preko odsevnih mehanizmov v Javi.

Za konec še poudarimo, da je neposredno primerjanje večkratnega razpošiljanja v Clojure in Javi nekoliko nerodno, saj njuni tipi, dedovanje in izvajanje metod delujejo na zelo drugačen način. Zato nekaj, kar dobro deluje v Clojure, ne bo nujno dobra rešitev za Javo ali pa vsaj ne v vseh primerih.

2.3.5 Julia

Eden izmed jezikov, ki podpira večkratno razpošiljanje, je tudi Julia [13, 12]. Jezik je dinamično tipiziran in se lahko prevede v strojno kodo ali pa se izvaja v ukazni lupini, kjer se kodo prevaja sproti. Pogosto se uporablja za numerično računanje kot hitrejša alternativa interpretiranim jezikom, kot sta na primer MATLAB ali Python, uporaben pa je seveda še za marsikaj drugega.

V programu 2.8 vidimo primer razpošiljanja v jeziku Julia, kjer smo definirali štiri metode in štiri različne tipe, ki nastopajo kot parametri. Za razliko od prejšnjih primerov nam ob definiciji

```
abstract type O end
struct A <: O end
struct B <: O end
struct C end


function dispatch_test()
    metoda(arg0::O, arg1::O) = "Osnovna metoda"
    metoda(arg0::A, arg1::B) = "Metoda 2"
    metoda(arg0::B, arg1::A) = "Metoda 3"
    metoda(arg0::O, arg1) = "Metoda 4"

    a::O = A()
    b::O = B()
    c = C()

    metoda(a, b)  # Metoda 2
    metoda(b, a)  # Metoda 3
    metoda(a, a)  # Osnovna metoda
    metoda(a, c)  # Metoda 4
end

dispatch_test()
```

Izsek kode 2.8: Primer večkratnega razpošiljanja v jeziku Julia.

metode vseh tipov ni treba določiti (npr. drugi parameter pri četrti metodi). Razpošiljanje bo neko metodo sprejelo ne glede na tip tako definiranega parametra, razen če najde drugo metodo, ki se parametrom bolj tesno prilega. To omogoča veliko svobodo pri pisanju metod, vendar lahko privede tudi do napak: če bi pri zgornjem primeru definirali še metodo `metoda(arg0, arg1::C)= "Metoda 5"`, bi razpošiljanje pri zadnjem klicu našlo dve enakovredno ujemajoči metodi in zato vrnilo napako.

Julia podpira tudi uporabo parametrov tipa (angl. `type parameters`) v definicijah metod, kar deluje podobno kot generične metode v Javi. Parametre, ki predstavljajo tipe, v definicijo metode dodamo s ključno besedo `where`, ob klicu pa se bodo te vrednosti zamenjale z dejanskimi tipi. Tako lahko definiramo metode, ki sprejmejo poljubne tipe, pod pogojem, da ima vsak parameter tipa le eno vrednost in da so vsi morebitni dodatni pogoji, ki smo jih dodali z besedo `where`, izpolnjeni.

Še ena dodatna posebnost, ki jo lahko uporabimo ob definiciji metod, so unije tipov. Z njimi lahko definiramo nov tip, ki deluje kot skupni starš tipov v uniji, tudi če so ti sicer med seboj povsem nepovezani.

Primer obeh opisanih načinov definicije metod je prikazan v programu 2.9. S parametri tipa smo definirali drugo metodo, ki sprejme dva poljubna parametra, vendar morata biti oba enakega tipa – edini parameter tipa je `T`, ki ne more imeti dveh različnih vrednosti. Tretja metoda je zelo podobna, vendar vsebuje še dodaten pogoj, s katerim zahtevamo, da je tip `T` izpeljan iz osnovnega tipa `O`. Definirali smo tudi nov tip `BaliD`, ki je unija tipov `B` in `D` in ki ga v definiciji metod lahko uporabimo, kot bi bil njun skupni starš.

V jeziku Julia razpošiljanje v ozadju deluje v principu podobno kot pri jeziku Clojure. Vsaka

```
abstract type O end
struct A <: O end
struct B <: O end
struct C end
struct D end
BaliD = Union{B,D}

function dispatch_test()

    metoda(x, y) = "Poljubna tipa"
    metoda(x::T, y::T) where {T} = "Katerakoli enaka tipa"
    metoda(x::T, y::T) where {T<:O} = "Enaka tipa, izpeljana iz O"
    metoda(x::BaliD, y::BaliD) = "Oba tipa sta B ali D"

    a = A(); b = B(); c = C(); d = D()

    metoda(a, b)  # Poljubna tipa
    metoda(a, a)  # Enaka tipa, izpeljana iz O
    metoda(c, c)  # Katerakoli enaka tipa
    metoda(b, d)  # Oba tipa sta B ali D
end

dispatch_test()
```

Izsek kode 2.9: Primer parametrov tipa in unij v jeziku Julia.

metoda ima svoj podpis, ki je zgrajen kot terka tipov, ki smo jih podali ob definiciji. Ob klicu metode se iz podanih parametrov prav tako zgradi terka tipov, razpošiljanje pa na podlagi nje v tabeli metod nato poišče najbolj ujemajočo metodo. Pri iskanju se vedno upošteva najbolj specifično najdeno metodo, kjer se za specifičnost poleg globine v hierarhiji upošteva še vrsta pravil. Eno tako pravilo je na primer, da so unije tipov manj specifične kot enostavni tipi, ki v uniji nastopajo. Najdena ujemajoča metoda se, tako kot pri jeziku Clojure, shrani v predpomnilnik, da je kasneje za enako kombinacijo tipov parametrov ni treba ponovno iskati.

Jezik Julia je spet težko neposredno primerjati z Javo oziroma z našo implementacijo iz poglavja 4, saj podpira dinamično tipiziranje in je zelo fleksibilen pri definiciji metod. Sicer je razpošiljanje od naših implementacij še najbolj podobno implementaciji z odsevnostjo, ki razpošilja z iskanjem po tabeli metod.

Poglavje 3

Anotacije in njihovo obdelovanje

To poglavje opisuje anotacije v Javi, mehanizme za obdelavo anotacij (angl. *annotation processing*) in njihovo uporabo. Anotacije lahko obdelujemo iz veliko različnih razlogov, na primer za avtomatsko pisanje dokumentacije in testnih programov, za generiranje nove izvorne kode oziroma za kakršenkoli postopek, ki ga želimo opraviti med prevajanjem. Manj pogosta uporaba obdelovanja anotacij je tudi spreminjanje kode v sintaksnem drevesu prevajalnika, kar nam omogoča spreminjanje samega programa, medtem ko se prevaja, ne da bi generirali nove izvorne `.java` datoteke. Ta način spreminjanja delovanja programa uradno ni podprt in se zato tudi redko uporablja – poleg knjižnice Lombok [16] skorajda ni mogoče najti primerov njegove uporabe. V tem poglavju bomo opisali tako mehanizme za običajno obdelovanje anotacij kot tudi mehanizme, s katerimi lahko spreminjamo prevajano kodo in ki so v tej nalogi uporabljeni za implementacijo večkratnega razpošiljanja. Tako lahko to poglavje služi tudi za izhodišče nekomu, ki bi rad implementiral podobno knjižnico.

V nadaljevanju naloge bomo uporabljali javansko kodo in prevajalnik `javac` različice 1.8. Ker spreminjanje kode med prevajanjem na način, kot ga bomo opisali tukaj, uradno ni podprt, obstaja verjetnost, da se bodo v kasnejših izdajah prevajalnika opisani programski vmesniki spremenili.

3.1 Anotacije

Anotacije [3] so oznake, ki jih v Javi lahko uporabimo za dodajanje metapodatkov v program. Na voljo so od Jave 1.5 naprej, v kasnejših izdajah pa so se še nekoliko nadgrajevale oz. spreminjale.

Ločimo jih lahko v tri skupine glede na to, v kateri fazi v življenju programa se uporabljajo:

- Anotacije, namenjene za delo z izvorno kodo (angl. *source retention*), ki se ohranjajo do prevajanja. Namenjene so prevajalniku in lahko kodo dopolnjujejo ali pa pomagajo

izpostaviti napake med prevajanjem. Ko jih prevajalnik uporabi, jih odstrani, zato v kasnejših fazah niso več na voljo. Dva primera sta `@Override`, ki označuje metodo, ki prekriva metodo iz nadrazreda – prevajalnik bo javil napako, če taka metoda v nadrazredu ne obstaja – ali pa `@SuppressWarnings`, s katero lahko prevajalniku povemo, naj ignorira zaznane napake.

- Anotacije, ki ostanejo v prevedeni datoteki, niso pa na voljo med izvajanjem (angl. *class retention*). Uporabimo jih lahko med prevajanjem z mehanizmi za obdelovanje anotacij ali po prevajanju s pomočjo zunanjih orodij. Tak tip anotacij je uporabljen v knjižnici, razviti v sklopu te naloge, njihovo obdelovanje pa je podrobneje opisano v nadaljevanju tega poglavja.
- Anotacije, ki ostanejo v izvajalni kodi (angl. *runtime retention*) in so namenjene procesiranju med izvajanjem programa, do njih pa lahko dostopamo na primer z uporabo odsevnosti. Primer takih anotacij so ukazi različnim ogrodjem za izvajanje spletnih aplikacij, ki na podlagi njih ugotovijo, čemu so označeni elementi programa namenjeni in kakšna je njihova naloga pri izvajanju aplikacije. Razširjena specifikacija Jave, Java EE (*Enterprise Edition*) definira veliko število takih anotacij.

Anotacije lahko dodajamo na vse deklaracije – razrede, metode, spremenljivke, itd., v kasnejših verzijah Jave pa tudi na tipe ob kreiranju novih objektov, implementaciji vmesnikov ali spreminjanju tipa vrednosti. Nekaj anotacij je v Javi že vgrajenih, na primer `@Override` ali `@Deprecated`, definiramo pa seveda lahko tudi svoje, ki jih kasneje uporabimo med prevajanjem z obdelovalcem anotacij (angl. `annotation processor`) ali med izvajanjem z odsevnimi mehanizmi.

3.2 Obdelovanje anotacij

Za obdelovanje anotacij potrebujemo obdelovalec anotacij (angl. *annotation processor*) – to je razred, ki razširja abstraktni razred `AbstractProcessor` in implementira njegovo metodo `process`. Vsakemu obdelovalcu moramo določiti seznam anotacij, ki jih bo obdeloval, in verzijo Jave, ki ji je namenjen. To določimo z anotacijama `@SupportedAnnotationTypes` in `@SupportedSourceVersion` na vrhu njegovega pripadajočega razreda.

Prevajalniku povemo, naj med prevajanjem uporabi obdelovalca anotacij tako, da klicu dodamo zastavico `-processor`. Po prvem delu prevajanja, kjer se zgradi abstrakno sintaksno drevo programa, bo prevajalnik zahtevanega obdelovalca zagnal, obdelovanje pa nato lahko poteka v več krogih. Če se med obdelavo anotacij ustvarijo nove datoteke, jih najprej obdela prevajalnik, nato pa se izvede naslednji krog obdelovanja anotacij. Med posameznimi krogi obdelovalec ohranja stanje, zato se zaveda tudi vseh prejšnjih krogov. Ko po izvedbi enega kroga ni nobenih novih datotek, se obdelava anotacij zaključi, prevajanje pa preide v naslednjo fazo, kjer se abstraktno sintaksno drevo dokončno prevede v izvajalno kodo.

3.3 Inicializacija

Ko se obdelovalec zažene, najprej pokliče metodo `init` s parametrom tipa `ProcessingEnvironment` (v primeru dela s prevajalnikom `javac` tipa `JavacProcessingEnvironment`), ki vsebuje vse informacije in pomožne metode, povezane s prevajalnikom. S pomočjo teh informacij lahko inicializiramo nekaj razredov, ki se uporabljajo v kasnejših fazah obdelave anotacij:

- razred `Trees` vsebuje informacije o abstraktnem sintaksnem drevesu prevajanega programa. Drevo se je ustvarilo v prejšnjih fazah prevajanja, iz njega pa se bo v kasnejših fazah generirala izvajalna koda v obliki `.class` datotek.
- razred `TreeMaker` vsebuje pomožne metode za delo z abstraktnim sintaksnim drevesom, predvsem za generiranje novih poddreves. Namenjen je za uporabo znotraj samega prevajalnika, v nadaljevanju pa ga bomo uporabljali za generiranje nove kode.
- razred `Elements` vsebuje pomožne metode za spremljanje informacij o elementih programa. Uporabili ga bomo predvsem za definiranje in sklicevanje na imena spremenljivk, pa tudi za generiranje novih tipov, kjer bo to potrebno.
- razred `Filer` nam omogoča kreiranje novih datotek z izvorno kodo, ki si jih bo prevajalnik zapomnil in jih prevedel po koncu te faze obdelave anotacij.
- razred `Types` vsebuje pomožne metode za delo s tipi. Uporabili ga bomo za ugotavljanje razrednih hierarhij med tipi, ki nastopajo v prevajanem programu.
- razred `Symtab` vsebuje podatke o konstantah, operatorjih in tipih, ki so del samega jezika, na primer razred `Object`. Potrebovali ga bomo za definiranje simbolov za nove metode in razrede.

Tudi če obdelovanje anotacij poteka v več krogih, se instanca obdelovalca med njimi ohranja, zato se inicializacija izvede samo enkrat in kasnejši klici metode `process` lahko dostopajo do spremenljivk iz prejšnjih krogov.

3.4 Običajni mehanizmi obdelave anotacij

V vsakem krogu obdelave anotacij se pokliče metoda `process`, ki kot parametra prejme množico anotacij, ki so bile najdene v obdelovani kodi, in spremenljivko tipa `RoundEnvironment`. Slednja vsebuje podatke, ki se navezujejo na prevajano kodo in so relevantni za ta krog, s pomočjo nje pa bomo izvajali celotno obdelovanje. Uporabimo jo lahko za pridobivanje referenc na elemente programa (razrede, metode, spremenljivke), ki so označeni z določeno anotacijo, nato pa preko objekta `Trees` vsakega od teh elementov poiščemo v abstraktnem sintaksnem drevesu. S tem

dobimo vse informacije o strukturi tega dela kode, kar nam pomaga pri izvedbi cilja obdelave anotacij, na primer generiranja dokumentacije.

Generiranje nove izvorne kode v obliki datotek `.java` je prav tako pogost namen uporabe anotacij. Tako lahko na primer generiramo povsem nove programe, velike datoteke, ki bi jih bilo prezahtevno napisati ročno, ali pa razrede, ki razširjajo razred, ki se ravnokar prevaja. Za ustvarjanje novih datotek uporabimo razred `Filer`, ki nam vrne referenco na tekstovno datoteko, v katero potem lahko pišemo z običajnimi mehanizmi za delo z datotekami. Če smo z razredom `Filer` ustvarili nove datoteke, bo prevajalnik ob koncu izvajanja metode `process` začel nov krog prevajanja, kjer bodo poleg razredov obdelanih v prejšnjem krogu vključene tudi nove generirane datoteke. Če smo v novi datoteki uporabili katero izmed anotacij, ki jih naš obdelovalec podpira, bodo te vključene tudi v nov krog obdelave anotacij.

3.5 Abstraktno sintaksno drevo

Vsi elementi abstraktnega sintaksnega drevesa razširjajo razred `JCTree`. Takih elementov je zelo veliko, tu bomo našteli le nekaj pomembnejših:

`JCCompilationUnit` je najvišje v drevesu in se nanaša na eno datoteko, ki jo prevajamo. Vsebuje seznam definicij (običajno definicija razreda in stavki za uvoz zunanjih paketov), pa tudi podatke o izvorni datoteki in imenu paketa.

`JCExpression` predstavlja javanski izraz, torej del kode, ki vrača neko vrednost. Razširjajo ga razredi, od katerih vsak predstavlja vrsto izraza, na primer klic metode (`JCMethodInvocation`), enočlenske in dvočlenske operacije, kot sta negacija in seštevanje (`JCUnary`, `JCBinary`), dostopanje do spremenljivk oz. ostalih simbolov (`JCIdent`), dostopanje do polja v razredu ali elementa v tabeli (`JCFieldAccess`, `JCArrayAccess`), sprememba tipa (`JCTypeCast`), prirejanje vrednosti (`JCAssign`), instanciranje novega razreda (`JCNewClass`) in tako dalje.

`JCStatement` predstavlja javanski stavek oziroma del kode, ki se izvede in ne vrača vrednosti. Primeri stavkov v obliki razredov, ki so izpeljani iz `JCStatement`, so `if` stavek (`JCIf`), definicija spremenljivke ali razreda (`JCVariableDecl`, `JCClassDecl`), blok z zaporedjem več stavkov (`JCBlock`), zanke `for` in `while` (`JCForLoop`, `JCWhileLoop`), `try` stavek (`JCTry`), ključne besede za kontrolo toka programa (`JCBreak`, `JCContinue`, `JCReturn`, `JCThrow`), stavek za izvedbo izraza brez upoštevanja vrnjene vrednosti (`JCExpressionStatement`) in drugi.

`JCModifiers` vsebuje vse modifikatorje, ki jih lahko naštejemo ob deklaraciji razreda, metode ali spremenljivke. Poleg seznama anotacij so to še zastavice, ki se nanašajo na vidnost, statičnost, abstraktnost in podobne lastnosti, ki jih deklaracije lahko imajo.

JCClassDecl predstavlja deklaracijo razreda. Vsebuje seznam z definicijami vsebovanih metod in polj, seznam razredov, ki jih razširja ali implementira, deklaracijske modifikatorje, podatke za identifikacijo (ime in simbol) in tako dalje.

JCMethodDecl predstavlja deklaracijo metode. Vsebuje tip in telo metode, seznam parametrov, morebitne izjeme, ki jih metoda lahko proži, deklaracijske modifikatorje, ime in simbol metode, itd.

Vsak element abstraktnega sintaksnega drevesa je zgrajen neposredno iz enega elementa izvorne kode, zato je mogoče postopek tudi obrniti – z metodo `toString()` iz kateregakoli elementa v drevesu dobimo pripadajočo javansko kodo. Če to metodo pokličemo na korenu drevesa, nazaj dobimo celotno izvorno kodo prevajane datoteke, kar je zelo uporabno pri iskanju napak, ki smo jih morda povzročili med spreminjanjem drevesa v fazi obdelave anotacij.

3.6 Spreminjanje obstoječe kode

Prevajalnik nam med obdelavo anotacij poda referenco na abstraktno sintaksno drevo programa, ki se trenutno prevaja in s katerim se prevajanje nadaljuje po obdelavi anotacij. Zaradi tega ga lahko z uporabo enakih orodij, ki so bila uporabljena pri njegovi gradnji, tudi spremenimo, spremembe pa se bodo poznale v končnem prevedenem programu. Pri tem moramo biti pazljivi, saj lahko hitro pridemo v stanje, ki ga prevajalnik ni pričakoval, kar lahko povzroči napako v zaključni fazi prevajanja ali pa nepričakovano obnašanje končnega programa. Poleg tega za ta način obdelave drevesa ne obstaja nobena uradna dokumentacija, kar dodatno oteži iskanje napak. Zaradi tega se je dobro zanašati na razhroščevalnik in obdelano drevo primerjati s podobnim drevesom, ki ga je ustvaril prevajalnik.

Razred `TreeMaker` vsebuje metode za gradnjo elementov sintaksnega drevesa, kar precej olajša delo v tej fazi. Poleg njega potrebujemo na tem mestu še razred `Elements` za sklicevanje na simbole (npr. imena spremenljivk) in razred `Types`, za definiranje tipov. Ko v kombinaciji z njimi uporabimo še nekaj pomožnih metod, na primer za pretvorbo med standardnimi `java.util.List` seznami in internimi `com.sun.tools.javac.util.List` seznami, ki jih uporablja prevajalnik, koda za neposredno gradnjo dreves postane precej berljiva.

Izsek kode 3.1 prikazuje, kako z metodami iz razreda `TreeMaker` zgradimo stavek `List<Method> candidates = methodMap.get("metoda").get(3)`. V primeru uporabljamo dve pomožni metodi za delo z prevajalnikovimi internimi seznami – `asJavacList()`, ki iz podanih objektov ustvari seznam in `emptyExpr()`, ki vrne prazen seznam izrazov tipa `JCExpression`. Spremenljivki `tm` in `el` sta razreda `TreeMaker` in `Elements`, ki smo ju ustvarili ob postavitvi obdelovalca anotacij, kar je opisano v razdelku 3.3. V primeru je uporabljenih nekaj prej omenjenih metod iz razreda `TreeMaker`:

```
JCTree.JCVariableDecl declaration = tm.VarDef(
    tm.Modifiers(0),
    el.getName("candidates"),
    tm.TypeApply(
        tm.Ident(el.getName("List")),
        asJavacList(tm.Ident(el.getName("Method")))
    ),
    tm.Apply(
        emptyExpr(),
        tm.Select(
            tm.Apply(
                emptyExpr(),
                tm.Select(
                    tm.Ident(el.getName("methodMap")),
                    el.getName("get")
                ),
                asJavacList(tm.Literal(TypeTag.CLASS, "metoda"))
            ),
            el.getName("get")
        ),
        asJavacList(tm.Literal(TypeTag.INT, 3))
    )
);
```

Izsek kode 3.1: Primer gradnje sintaksnega drevesa.

`Ident` – Iz imena spremenljivke ustvari izraz tipa `JCIdent`, ki vrača njeno vrednost.

`Select` – Vrne element razreda (vrednost ali metodo) v obliki izraza `JCFieldAccess`, podamo pa ji izraz, ki določa spremenljivko z razredom in ime želene metode oziroma vrednosti.

`Apply` – Ustvari razred `JCMethodInvocation`, ki predstavlja klic metode. Za parametre ji podamo seznam parametrov tipa (samo v primeru generičnih metod), izraz, ki predstavlja izbrano metodo, in seznam parametrov, ki so prav tako podani v obliki izrazov oziroma razredov tipa `JCExpression`.

`Literal` – Vrača literale v obliki izraza `JCLiteral`. Podamo mu konstanto, ki predstavlja enega od možnih vrst literalov, in objekt, ki predstavlja njegovo vrednost.

`Modifiers` – Zgradi nam objekt `JCModifiers`, ki predstavlja modifikatorje vezane na neko deklaracijo. Podamo mu bitno masko, kjer vsak bit predstavlja enega od modifikatorjev, opcijsko pa tudi seznam anotacij.

`TypeApply` – S to metodo lahko ustvarimo sestavljene tipe, tako da ji podamo osnovni tip in seznam parametrov. V našem primeru sta to osnovni tip `List` in seznam s tipom `Method`, ki skupaj tvorita sestavljeni tip `List<Method>`.

`VarDef` – S to metodo tvorimo deklaracije spremenljivk tipa `JCVariableDecl`. Podati ji moramo modifikatorje, ime spremenljivke, želen tip in izraz za takojšno inicializacijo.

Podobnih pomožnih metod, ki so za razumevanje in uporabo precej enostavne, je v razredu `TreeMaker` na voljo še zelo veliko – za vse možne konstrukte, ki jih Java ponuja. Več o njih in

o ostalih opisanih postopkih za gradnjo sintaksnega drevesa je mogoče razbrati iz izvorne kode prevajalnika [15], ki vsebuje precej uporabnih komentarjev, druga uradna dokumentacija pa zanje ne obstaja.

3.7 Simbolne tabele

S prej opisanimi mehanizmi lahko povsem spremenimo abstraktno sintaksno drevo, ki se nato vključi v nadaljevanje prevajanja. Poleg spreminjanja samega drevesa je treba paziti tudi, da so simboli za vse elemente nekega razreda zabeleženi v njegovi simbolni tabeli. Preko simbolnih tabel prevajalnik lahko ve, kateri elementi so v vsakem od razredov prisotni, kar potrebuje pri pretvarjanju sintaksnega drevesa v izvajalno kodo.

Simboli so oznake v elementih drevesa, ki predstavljajo deklaracije, na primer `JCClassDecl`, `JCMethodDecl` in `JCVarDecl`. Izpeljani so iz abstraktnega razreda `Symbol` in vsebujejo podatke o imenu deklariranega elementa, njegovih modifikatorjih, njegovem staršu znotraj sintaksnega drevesa in druge podatke, preko katerih jih lahko prevajalnik najde v zaključnih fazah prevajanja.

Simbolne tabele prevajalnik sproti polni med prvo fazo prevajanja, če pa med obdelavo anotacij v razred dodajamo nove elemente, je treba njihove simbole v tabele dodati ročno. To lahko storimo z uporabo odsevnosti, tako da iz simbola razreda, ki je tipa `ClassSymbol`, pridobimo element `members_field` tipa `Scope`, ki predstavlja simbolno tabelo za ta razred. Vanjo lahko z metodo `enter` dodamo simbol nove metode ali polja, ki smo ga dodali v sintaksnem drevesu. Ta postopek je treba ponoviti vsakič, ko v razred dodamo novo metodo, globalno spremenljivko ali ugnezdeni razred, ne pa tudi na primer pri pomožnih spremenljivkah znotraj neke metode.

3.8 Knjižnica Project Lombok

Eden od redkih primerov, kjer se mehanizmi za spreminjanje kode med obdelavo anotacij resno uporabljajo, je knjižnica Project Lombok [16]. Namenjena je generiranju delov kode, ki so zelo enostavni, vendar so za programerja precej zamudni, kot so *get* in *set* metode ali preverjanje `null` vrednosti, pa tudi naprednejših funkcionalnosti, kot so spremenljivke z implicitnimi tipi in poenostavitve beleženja dnevniških zapisov.

Lombok kodo generira na podoben način kot zgoraj opisani postopki, le da podpira več različnih prevajalnikov. Zaradi tega ima implementirano svoje abstraktno sintaksno drevo, na katerem izvede vse potrebne spremembe ločeno od katerekoli dejanske implementacije prevajalnika. Za delovanje tako potrebuje dodatno kodo, ki zna Lombokovo sintaksno drevo prevesti v strukturo, ki jo razume določen prevajalnik, na primer `com.sun.tools.javac.tree.JCTree`, ki ga uporablja prevajalnik `javac` in smo si ga pogledali zgoraj, ali `org.eclipse.jdt.internal.compiler.ast.ASTNode`, ki je interna struktura v Eclipsovem prevajalniku. Pri obdelavi programa tako Lombok najprej

prevede kodo v svojo interno strukturo, na njej izvede vse potrebne operacije, nato pa jo prevede nazaj, da prevajalnik lahko nadaljuje z delom. To knjižnici poleg kompatibilnosti z različnimi razvijalskimi orodji omogoča tudi lažje prilagajanje na morebitne spremembe v novih izdajah prevajalnikov in nedokumentiranih metod, ki se v njih uporabljajo. Poleg tega je oblika Lombokovega drevesa prilagojena tako, da je njegovo obdelovanje čimbolj enostavno in učinkovito.

Poglavje 4

Implementacija

To poglavje opisuje, kako implementirana knjižnica z uporabo obdelave anotacij v Javi simulira večkratno razpošiljanje. Knjižnica je na voljo v treh različicah, od katerih vsaka uporablja drugačen pristop za simuliranje razpošiljanja in ima zato drugačne prednosti in slabosti.

4.1 Simuliranje večkratnega razpošiljanja

Večkratno razpošiljanje simuliramo tako, da ob klicu ne izvedemo metode, ki jo je določil prevajalnik, ampak poiščemo eno izmed definiranih metod, ki najbolj ustreza podanim parametrom. Pri iskanju najustreznejše metode izbiramo samo med takimi, ki imajo enako ime in število parametrov kot prvotni klic. V nadaljevanju bomo tako skupino metod, ki jo določata ime in število parametrov, imenovali *model* metode, posamezne metode pa *primerki* ali *instance* tega modela. Ko se v kodi pojavi klic metode, to za nas torej pomeni klic modela, razpošiljanje pa mora nato ugotoviti dejanske tipe parametrov in izbrati primerek klicanega modela, ki se tem tipom najbolj prilega.

Postopek je v treh različicah knjižnice implementiran na tri različne načine:

Odločitveno drevo z neposrednim preverjanjem tipov – Pri tem načinu zgradimo odločitveno drevo, v katerem neposredno preverjamo tipe posameznih parametrov metode. Vsak nivo drevesa predstavlja enega od parametrov, vsaka veja pa je eden izmed možnih tipov za ta parameter. Vsak list v takem drevesu je en izmed možnih primerkov, med katerimi mora mehanizem razpošiljanja izbrati. Drevo je v knjižnici implementiramo kot zaporedje `if` in `instanceof` operatorjev. Podrobnosti o implementaciji tega načina so na voljo v razdelku 4.4.

Odsevnost – Pri tem načinu program uporabi mehanizme za pregled lastne strukture, da ugotovi, katere metode so na voljo, in jih vstavi v tabelo metod. Pri klicanju metod nato

vsakič izbere najbolj ustrezno izmed njih, kar deluje na podoben način kot knjižnica JMMF ter jezika Clojure in Julia, kar smo opisali v razdelku 2.3. Prednost tega načina je njegova enostavnost, saj koda je za vsako metodo lahko (skoraj) enaka, ker se del logike za razpošiljanje prestavi v fazo izvajanja. Podrobneje je mehanizem opisan v razdelku 4.5.

Večnivojski obiskovalec – Ta način je razširjena verzija znanega načrtovalskega vzorca obiskovalec. Za svoje delovanje izkorišča enojno razpošiljanje, ki ga Java podpira – če več razredov implementira isto metodo, bo izvajalno okolje preverilo, za kateri dejanski tip parametra gre, in izvedlo metodo iz ustreznega razreda. Postopek je podoben kot pri odločitvenem drevesu, le da operatorje `if` in `instanceof` nadomestimo z mehanizmom enojnega razpošiljanja, kar bo morda pohitrilo izvajanje. Slaba lastnost tega pristopa je, da je treba za vsako vejo drevesa implementirati svojo metodo in to v vsakem razredu, ki lahko nastopa kot parameter, kar privede do precej večje količine kode kot v ostalih dveh različicah knjižnice. Podrobnejši opis večnivojskega obiskovalca je v razdelku 4.6. Podoben pristop implementiran v jeziku C# je opisan v sorodni literaturi [14], vendar namesto enojnega razpošiljanja za razreševanje tipov uporabi mehanizme za delo z dinamičnimi tipi spremenljivk, ki so na voljo v tem jeziku.

4.2 Uporaba knjižnice

Knjižnica je zasnovana tako, da je za uporabnika čim bolj enostavna in da se koda lahko piše, kot da Java podpira večkratno razpošiljanje. Za uporabo je treba v kodo dodati tri anotacije:

`@MultiDispatch` – To anotacijo se doda na metode, pri katerih želimo uporabljati večkratno razpošiljanje. Med prevajanjem knjižnica tako označene metode združi v skupine glede na ime in število parametrov in iz vsake skupine ustvari model metode. Za vse primerke istega modela ni nujno, da se nahajajo v istem razredu oz. v isti datoteki, vse tako označene metode pa morajo biti javne (`public`).

`@MultiDispatchClass` – To anotacijo je potrebno dodati na vse razrede, v katerih se uporabljajo klici prej označenih metod, za katere želimo, da se izvedejo z večkratnim razpošiljanjem. Obdelovalec anotacij bo za take razrede znal klice metod prilagoditi, če pa eno izmed označenih metod uporabimo v neoznačenem razredu, se bo poklicala na običajen javanski način.

`@MultiDispatchVisitable` – S to anotacijo označimo vse razrede, ki lahko nastopajo kot parametri v metodah, med katerimi razpošiljanje izbira. Anotacija je obvezna le, če uporabljamo različico z obiskovalcem – glede na te oznake bo obdelovalec vedel, v katere razrede je potrebno dodati obiskovalske metode. V nekaterih primerih, na primer če ti razredi niso del

naše kode, tega ne moremo storiti, zato pristop z obiskovalcem ne bo deloval. Preostala dva načina delujeta tudi brez te anotacije.

Dodatna omejitev je, da morajo vsi parametri v metodah izhajati iz razreda `Object`, kar pomeni, da ne moremo uporabljati enostavnih tipov. Zato je treba namesto tipov `int` ali `double` uporabiti pripadajoče razrede `Integer` oziroma `Double`. V primeru razpošiljanja z obiskovalcem lahko definiramo nov razred, ki vsebuje enega od želenih enostavnih tipov, nanj pa nato lahko dodamo anotacijo `@MultiDispatchVisitable`.

Za generiranje kode na tri zgoraj opisane načine so implementirani trije obdelovalci anotacij:

- `ProcessorTree`, ki deluje na podlagi odločitvenega drevesa,
- `ProcessorReflection`, ki deluje s pomočjo odsevnosti,
- `ProcessorVisitor`, ki uporablja razširjen vzorec obiskovalec.

Izbranega obdelovalca pri prevajanju uporabimo tako, da ukazu `javac` ustrezni razred dodamo z uporabo zastavice `-processor`. Na sliki 4.1 vidimo tri primere takega ukaza.

```
javac -cp annotation-1.0.jar:processor-1.0.jar -processor si.kisek.annotationdispatch.ProcessorTree Program.java

javac -cp annotation-1.0.jar:processor-1.0.jar -processor si.kisek.annotationdispatch.ProcessorReflection Program.java

javac -cp annotation-1.0.jar:processor-1.0.jar -processor si.kisek.annotationdispatch.ProcessorVisitor Program.java
```

Slika 4.1: Primer uporabe obdelovalca anotacij z ukazom `javac`.

Če za prevajanje uporabljamo sistem maven, lahko obdelovalca anotacij uporabimo z elementom `<annotationProcessor>` v konfiguraciji za `maven-compiler-plugin` vtičnik. Tako je to narejeno tudi v *Demo* primeru, vključenem v pripadajoči kodi na `https://github.com/thenejcar/AnnotationDispatch/tree/master/Demo`. Slika 4.2 prikazuje izsek iz take konfiguracijske datoteke.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.6.2</version>
    <configuration>
        <compilerVersion>1.8</compilerVersion>
        <source>1.8</source>
        <target>1.8</target>
        <annotationProcessors>
            <annotationProcessor>
                si.kisek.annotationdispatch.ProcessorTree
            </annotationProcessor>
        </annotationProcessors>
    </configuration>
</plugin>
```

Slika 4.2: Primer uporabe obdelovalca anotacij s sistemom maven.

4.3 Osnovni obdelovalec anotacij

Vse prej naštete izvedbe obdelovalca anotacij razširjajo abstraktni razred `MultidispatchProcessor`, ki vsebuje skupno logiko. Tudi sicer vsi trije delujejo na podoben način, ki ga lahko posplošimo na tri korake:

1. **Iskanje označenih metod** – Obdelovalec na začetku poišče vse metode, označene z `@MultiDispatch` anotacijo. Najdene metode z enakim imenom in številom parametrov združi v modele, tako da za vsakega ustvari objekt tipa `MethodModel`, za vsako metodo pa tudi objekt tipa `MethodInstance`, ki vsebuje tipe parametrov za določen primerek. Vse skupaj si nato zapomni s tabelo, ki preslika model v množico njegovih primerkov (spremenljivka tipa `HashMap<MethodModel, Set<MethodInstance>>`), ki je nato dostopna v vseh nadaljnjih korakih.

2. **Nove metode za razpošiljanje** – Obdelovalec v naslednjem koraku za vsak model ustvari novo metodo, ki sprejme primerno število spremenljivk tipa `Object` in jih nato z enim od treh mehanizmov razpošlje najustreznejšemu primerku metode. Sam mehanizem razpošiljanja lahko seveda vsebuje tudi svoje dodatne metode ali razrede, ki jih prav tako generiramo na tem mestu. V tem koraku se trije mehanizmi najbolj razlikujejo med seboj, podrobnosti implementacij pa so opisane v nadaljevanju poglavja.

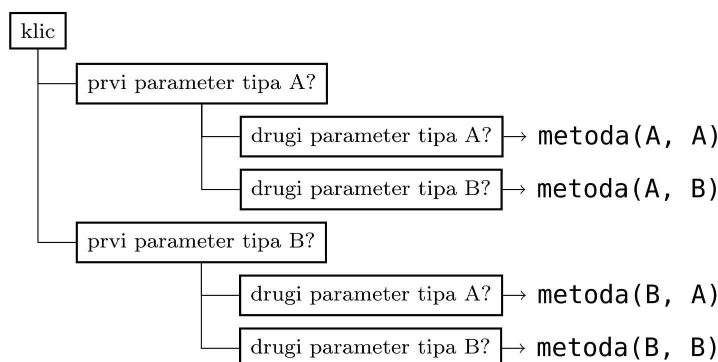
3. **Zamenjava metod** – V vseh razredih, ki so označeni z anotacijo `@MultiDispatchClass`, obdelovalec poišče klice označenih metod in jih zamenja s klici novih metod, generiranih v prejšnjem koraku. S tem se bo ob vsakem klicu sprožila logika za razpošiljanje in izvedla se bo najustreznejša metoda.

4.4 Odločitveno drevo

Ta način je najbolj podoben logiki, ki bi jo programer najverjetneje uporabil, če bi razpošiljanje implementiral ročno. Delovanje tega postopka temelji na odločitvenem drevesu, po katerem metode iščemo z neposrednim preverjanjem tipov.

Primer sheme takega drevesa je prikazan na sliki 4.3. Metode so razporejene v liste drevesa, do katerih pridemo tako, da z vsakim nivojem drevesa ugotovimo tip enega od parametrov. Preverjanje je vedno urejeno po specifičnosti, tako da najprej preverjamo ozko določene tipe, kasneje pa šele njihove starše oziroma nadtipe. Tako razpošiljanje je enostavno in razumljivo, vendar lahko za veliko število parametrov in globoke hierarhije tipov dobimo zelo veliko `if` stavkov, ki morajo biti urejeni na točno določen način. To pomeni, da postane koda pri uporabi tega načina za večje primere prezapletena, da bi jo programirali ročno.

Obdelovalec za vsak najden model zgradi drevesno strukturo v obliki razreda `MethodSwitcher`. Drevesna struktura je zgrajena rekurzivno v več nivojih – na i-tem nivoju rekurzije za vsak



Slika 4.3: Shema razpošiljanja z odločitvenim drevesom za štiri metode.

primerek modela pogledamo tip njegovega i-tega parametra in take z istim tipom združimo v skupino. Tako dobimo preslikavo iz tipa v množico primerkov, ki nam glede na tip i-tega parametra vrne množico ujemajočih metod, znotraj katerih bomo izbirali v naslednjem nivoju drevesa. Tako se lahko s preverjanjem tipov i-tega parametra odločimo, v katerem poddrevesu bomo nadaljevali razpošiljanje.

Ko imamo preslikave med možnimi tipi in pripadajočimi skupinami metod, jih moramo še urediti v vrstni red. Ko bomo preverjali ujemanje tipov, je treba začeti s čim bolj specifičnimi, da bomo na koncu našli metodo, ki se podanim parametrom prilega čim tesneje. Urejanje tipov po specifičnosti poteka tako, da iz seznama tipov najprej zgradimo razred `TypeTree`, ki predstavlja razredno hierarhijo. Razred `TypeTree` je zgrajen kot drevo, kjer so elementi tipi, njihovi otroci pa tipi izpeljani iz njih. Če na takem drevesu naredimo obhod z iskanjem v globino, dobimo vrstni red elementov, kjer so izpeljani podtipi vedno za svojimi starši. Če ta seznam obrnemo, dobimo ravno vrstni red, ki ga potrebujemo pri preverjanju tipov parametrov v odločitvenem drevesu – preverjanje za nek tip se bo izvedlo šele, če bomo prej preverili vse tipe, ki so iz njega izpeljani.

Tako imamo na nivoju i urejen seznam tipov in njihove pripadajoče skupine. Za vsako skupino nato rekurzivno ponovimo celoten postopek, pri tem povečamo i (zaporedno številko opazovanega parametra) za ena, informacijo o tipu, ki skupino določa, pa prenesemo na naslednji nivo. Rekurzija se ustavi, ko pridemo do zadnjega parametra, pri čemer je v preostalem naboru vedno le še ena metoda, ki jo postavimo v list drevesa.

Tako dobljeno drevo lahko nato neposredno pretvorimo v kodo, ki bo izbirala najprimernejšo metodo. Vsak nivo drevesa pretvorimo v zaporedje stavkov `if`, vsak od katerih vodi v novo poddrevo – na primer `if(arg instanceof A){ /*poddrevo */}`, kjer je `arg` i-ti parameter klicane metode in `A` tip, ki določa to vejo. Vsako notranje poddrevo je nato zgrajeno na enak način, dokler ne pridemo do listov, kjer znotraj `if` stavka pokličemo najdeno metodo in zaključimo z izvajanjem. Primer celotnega drevesa je prikazan na izseku kode 4.1

Da je prva najdena rešitev res optimalna in se tipi v njej ujemajo čim tesneje, poskrbi

```
class O {...}
class A extends O {...}
class B extends O {...}

void metoda(A arg0, A arg1) {...}
void metoda(A arg0, B arg1) {...}
void metoda(B arg0, B arg1) {...}
void metoda(O arg0, O arg1) {...}

void odlocitvenoDrevo(Object arg0, Object arg1) {
    if (arg0 instanceof A) {
        if (arg1 instanceof A) {
            metoda((A) arg0, (A) arg1);
            return;
        }
        if (arg1 instanceof B) {
            metoda((A) arg0, (B) arg1);
            return;
        }
    }
    if (arg0 instanceof B) {
        if (arg1 instanceof B) {
            metoda((B) arg0, (B) arg1);
            return;
        }
    }
    if (arg0 instanceof O) {
        if (arg1 instanceof O) {
            metoda((O) arg0, (O) arg1);
            return;
        }
    }
    throw RuntimeException("Ni ustrezne metode za to kombinacijo parametrov")
}
```

Izsek kode 4.1: Drevo za štiri primerke metode s tremi tipi parametrov.

urejenost glede na specifičnost tipov. Če izvajanje zaide v vejo, kjer se nobena metoda ne ujema, se izvajanje nadaljuje v naslednjem `if` stavku, ki je na voljo – to zagotovimo tako, da ne uporabimo `else` stavka, temveč pustimo, da se `if` stavki izvajajo eden za drugim, izvajanje pa se ustavi s klicem `return`, ko najdemo ustrezno metodo. Če gre izvajanje mimo vseh `return` stavkov in pride do konca drevesa, to pomeni, da ustrezne metode ni bilo mogoče najti in razpošiljanje ni uspelo, zato v tem primeru javimo napako.

4.5 Odsevnost

Ta način za razliko od ostalih dveh, ki skušata čim večji del razpošiljanja razrešiti že med prevajanjem, velik del logike izvede šele med samim izvajanjem programa. Princip je podoben kot pri knjižnici JMMF [7], ki razpošiljanje v celoti razreši med samim izvajanjem, saj kot običajna knjižnica med prevajanjem niti ni na voljo. Naša implementacija obdelovalca anotacij sicer izkoristi še nekaj informacij o parametrih metod, ki jih ima na voljo med prevajanjem, da pohitri izvajanje.

Ideja temelji na tem, da ob začetku izvajanja programa poiščemo vse metode, ki so na voljo, ob klicu pa nato vsakič izberemo najprimernejšo izmed njih. Vse to se dogaja med samim izvajanjem s pomočjo odsevnosti (angl. *reflection*), ki nam omogoča, da pregledujemo vsebino razredov, spremenimo nekatere lastnosti metod ali pa, da katero izmed metod poženemo.

Na začetku tako kot pri ostalih dveh načinih zgradimo seznam vseh modelov in njihovih pripadajočih primerkov metod. V vsak razred, ki vsebuje označene metode, dodamo novo dvonivojsko razpršeno tabelo `Map<String, Map<Integer, List<Method>>> methodMap`, v kateri se bodo lahko hranili seznami referenc na metode, ločeni glede na ime metode in število parametrov. Tako bo na primer klic `methodMap.get("metoda").get(2)` vrnil seznam metod, ki se imenujejo *metoda* in imajo dva parametra.

Nato generiramo inicializacijsko metodo, ki bo napolnila prej omenjene sezname metod. To storimo z ukazom `Class.getDeclaredMethod()`, ki mu podamo imena in tipe parametrov za vse najdene metode, med katerimi bomo razpošiljali. Vrstni red metod določimo že med prevajanjem, da pohitrimo razpošiljanje, uredimo pa jih na enak način kot pri mehanizmu z odločitvenim drevesom – metode s čim bolj ozko določenimi parametri spredaj. Primer take inicializacijske metode je prikazan v izseku kode 4.2.

```
void init() throws NoSuchMethodException {
    methodMap = new HashMap<>();
    Class c = this.class;
    Method m = null;
    List<Method> list = new ArrayList<>();

    m = c.getDeclaredMethod("metoda", A.class, A.class);
    m.setAccessible(true);
    list.add(m);

    m = c.getDeclaredMethod("metoda", A.class, B.class);
    m.setAccessible(true);
    list.add(m);

    m = c.getDeclaredMethod("metoda", B.class, A.class);
    m.setAccessible(true);
    list.add(m);

    methodMap.putIfAbsent("metoda", new HashMap<>());
    methodMap.get("metoda").put(2, list);
}
```

Izsek kode 4.2: Inicializacijska metoda pri razpošiljanju z odsevnostjo.

Urejanje metod po specifičnosti je izvedeno z enakim postopkom kot pri odločitvenem drevesu. Na enak način zgradimo objekt `MethodSwitcher`, ki ga nato pretvorimo v seznam tako, da na njegovi drevesni strukturi uporabimo obhod z iskanjem v globino. Tako dobimo seznam metod, ki so urejene na enak način kot pri odločitvenem drevesu, le da niso v razvejani strukturi. S tem ponovno zagotovimo, da je metoda s parametri določenega tipa na vrsti kasneje kot metoda, ki za parametre uporablja njihove podtipe.

Tabelo bi lahko napolnili tudi šele med samim izvajanjem programa, s čimer bi še doda-

tno poenostavili fazo prevajanja in obdelovanja anotacij, saj metod ne bi bilo treba urejati in bi bila logika za razpošiljanje za vse modele vedno enaka. To bi storili tako, da bi z ukazom `Class.getDeclaredMethods()` pridobili vse metode v tem razredu, nato pa s pregledovanjem anotacij na vsaki metodi poiskali take, ki morajo sodelovati v razpošiljanju. V generirano kodo bi morali nato vključiti še algoritem za sortiranje metod glede na tipe parametrov. Podoben način uporablja knjižnica JMMF, opisana v razdelku 2.5, le da metod ne poišče preko anotacij, ampak preko imena in števila parametrov, ki jih je programer podal konstruktorju. V naši implementaciji tako polnjenje tabel med izvajanjem ni uporabljeno, saj bi s tem upočasnili vsako izvajanje programa, pri polnjenju tabel med prevajanjem pa je celoten postopek potrebno izvesti samo enkrat.

```
void dispatch(Object arg0, Object arg1) {
    if (methodMap == null) init(); // ob prvem klicu se napolni tabela metod

    for (Method m : methodMap.get("metoda").get(2)) {
        Class[] params = m.getParameterTypes();

        if (params[0].isAssignableFrom(arg0) && params[1].isAssignableFrom(arg1)) {
            // prva najdena metoda, kjer se ujemata oba parametra, se izvede
            m.invoke(this, arg0, arg1);
            return;
        }
    }
}
```

Izsek kode 4.3: Razpošiljevalna metoda pri razpošiljanju z odsevnostjo.

Ko imamo sezname metod, ki so razdeljeni po modelih in pravilno urejeni, je potrebno generirati še kodo, ki bo izvedla razpošiljanje. V ta namen za vsak model ustvarimo razpošiljevalno metodo, katere primer je prikazan v izseku kode 4.3. Metoda sprejme ustrezno število razredov `Object`, ki so parametri namenjeni klicani metodi. Če razpršena tabela `methodMap` še ni napolnjena, najprej pokličemo prej generirano inicializacijsko metodo. Nato iz tabele pridobimo kandidate za razpošiljanje v obliki seznama referenc na metode, ki se z modelom ujemajo po imenu in številu parametrov. Za vsakega kandidata s pomočjo metode `Class.isAssignableFrom()` preverimo, če se tipi njegovih parametrov ujemajo s parametri, ki so bili podani v razpošiljevalno metodo. Prvi kandidat, ki se ujema na vseh mestih, je najustreznejša metoda, saj so metode v seznamu že pravilno urejene.

4.6 Večnivojski obiskovalec

Ta način razpošiljanja je najbolj kompleksen od treh implementiranih v tej nalogi, saj je za njegovo delovanje potrebno v kodo dodati vrsto novih metod in celo nekaj novih razredov. Njegova prednost v primerjavi z drugima dvema postopkoma je v tem, da za iskanje ustrezne metode izkoristi mehanizme, ki jih za (enojno) razpošiljanje ponuja izvajalno okolje. Tako nam ni treba

implementirati preverjanja tipov in urejanja metod, saj oboje izvede JVM zaradi definicij in klicev metod, postavljenih in izvedenih v točno določenih razredih. Pri tem upamo, da bodo mehanizmi, ki jih JVM za to uporabi, hitrejši od mehanizmov, ki smo jih uporabili v prejšnjih dveh načinih.

4.6.1 Uporaba obiskovalca za razpošiljanje

Mehanizem je izpeljan iz načrtovalskega vzorca obiskovalec, ki ga v Javi običajno implementiramo z uporabo metod *sprejmi* (angl. *accept*) in *obišči* (angl. *visit*), kar je prikazano v izseku kode 4.4. Vse metode *obišči* se nahajajo znotraj obiskovalčevega razreda, vsaka pa kot parameter sprejme enega od razredov, ki jih obiskovalec zna obiskati. Obiskani razredi so običajno implementacije vmesnika (angl. *interface*), ki vsebuje metodo *sprejmi*, ta pa kot parameter prejme obiskovalca in na njem pokliče metodo *obišči*, ki ji poda svoj razred (s spremenljivko `this`). Če na takem obiskanem razredu pokličemo metodo *sprejmi*, ki ji podamo obiskovalca, bo izvajalno okolje s pomočjo enojnega razpošiljanja ugotovilo dinamični tip objekta in poklicalo pravo metodo *sprejmi*. Znotraj metode *sprejmi* se pokliče metoda *obišči*, ki se ji poda parameter `this`, za katerega prevajalnik vnaprej pozna tip. Tako se izvede prava metoda *obišči*, ne da bi bilo treba kje v kodi uporabiti `if` stavke, ukaz `instanceof`, mehanizme odsevnosti ali kakršenkoli drug način za preverjanje tipov.

```
class Visitor {
    void visit(A a) { /* obisk tipa A */ }
    void visit(B b) { /* obisk tipa B */ }
}

interface Visitable {
    void accept(Visitor v);
}

class A implements Visitable {
    void accept(Visitor v){
        // prevajalnik ve, da je this tipa A in vnaprej izbere ustrezno metodo visit
        v.visit(this);
    }
}
class B implements Visitable{
    void accept(Visitor v){
        v.visit(this);
    }
}
...

Visitable x = new A();
// izvajalno okolje bo ugotovilo katera metoda accept se mora izvesti
x.accept(new Visitor());
```

Izsek kode 4.4: Običajni vzorec obiskovalec.

Delovanje teh metod se zanaša na enojno razpošiljanje, pri katerem izvajalno okolje zna med izvajanjem ugotoviti tip prejemnika in tako pokliče pravo metodo *sprejmi*. Ker na ta način

obiskovalec določi tipa dveh podanih argumentov, to sta tip prejemnika in tip prvega argumenta, je s tem izvedel dvojno razpošiljanje. Ideja večkratnega razpošiljanja z večnivojskim obiskovalcem temelji na tem, da lahko ta vzorec razširimo za metode s poljubnim številom parametrov na način, ki je podoben postopku z odločitvenim drevesom iz razdelka 4.4, le da namesto ukazov `if` in `instanceof` uporabimo več nivojev metod *sprejmi* in *obišči*.

Za potrebe takega sistema metod potrebujemo dva nova razreda – obiskovalec oz. `Visitor`, ki bo vseboval metode *obišči*, in vmesnik obiskovani oz. `Visitable`, ki vsebuje vse možne metode *sprejmi*. Vsi razredi, ki so uporabljeni kot parameter v kateremkoli primerku metode, morajo implementirati vse metode iz vmesnika `Visitable`. Nekatere izmed teh metod *sprejmi* sodelujejo pri razreševanju tipov parametrov, preostale pa predstavljajo zgrešitve pri iskanju, kar je podrobneje razloženo v nadaljevanju.

4.6.2 Nivoji obiskovalca

Metode *sprejmi* in *obišči* so pri večnivojskem obiskovalcu razdeljene v več nivojev, kjer je vsak zadolžen za razrešitev tipa enega parametra, informacijo o tipih predhodnih parametrov pa dobi od prejšnjega nivoja. Metode na zadnjem nivoju imajo tako informacijo o tipih vseh parametrov in zato lahko pokličejo ustrezen primerek metode. Primer poenostavljene implementacije obiskovalca za tri metode s tremi parametri je prikazan na izseku kode 4.5, ki prikazuje primer z dvema primerkoma, ki za parametre uporabljata dva različna razreda. Prikazani razred `Visitor` vsebuje metode *obišči*, razreda `A` in `B` pa vsebujeta vse metode *sprejmi* – nekatere izmed njih so relevantne za razpošiljanje, druge pa predstavljajo zgrešitve, saj se kombinacija njihovih parametrov ne ujema z nobeno od primerkov metod. V nadaljevanju je pojasnjeno delovanje posameznih nivojev metod, odzivanje na zgrešitve, ki v izseku kode 4.5 ni prikazano, pa je pojasnjeno v naslednjem razdelku.

Prvi nivo Metode *sprejmi* prvega nivoja so implementirane v vseh razredih, ki lahko nastopajo kot prvi parameter v eni od metod in prejmejo obiskovalca ter preostale parametre (od drugega naprej) v obliki objektov tipa `Visitable`. Metoda pokliče obiskovalčevo metodo *obišči* na prvem nivoju, ki ji kot prvi parameter poda objekt `this`, s čimer določi tip prvega parametra.

Metode *obišči* na prvem nivoju prejmejo en parameter v obliki objekta končnega (ravnokar razrešenega) tipa in preostale parametre v obliki razredov `Visitable`. Obiskovalec mora vsebovati eno tako metodo za vsak tip, ki v nekem primerku nastopa kot prvi parameter. Vse, kar taka metoda naredi, je, da na drugem parametru pokliče metodo *sprejmi* drugega nivoja, ki ji namesto samih nerazrešenih parametrov poda enega razrešenega in nerazrešene parametre od tretjega naprej.

```
void metoda(A arg0, A arg1, A arg2) { ... }
void metoda(A arg0, B arg1, A arg2) { ... }
void metoda(B arg0, B arg1, A arg2) { ... }

class Visitor {
    void visit1(A arg0, Visitable arg1, Visitable arg2){ arg1.accept2(this, arg0, arg2); }
    void visit1(B arg0, Visitable arg1, Visitable arg2){ arg1.accept2(this, arg0, arg2); }

    void visit2(A arg0, A arg1, Visitable arg2){ arg2.accept3(this, arg0, arg1); }
    void visit2(A arg0, B arg1, Visitable arg2){ arg2.accept3(this, arg0, arg1); }

    void visit3(A arg0, A arg1, A arg2){ metoda(A arg0, A arg1, A arg2); }
    void visit3(A arg0, B arg1, A arg2){ metoda(A arg0, B arg1, A arg2); }
    void visit3(B arg0, B arg1, A arg2){ metoda(B arg0, B arg1, A arg2); }
}

class A implements Visitable {
    void accept1(Visitor v, Visitable arg1, Visitable arg2){ v.visit(this, arg1, arg2); }

    void accept2(Visitor v, A arg0, Visitable arg2){ v.visit(arg0, this, arg2); }
    void accept2(Visitor v, B arg0, Visitable arg2){ /* zgresitev */ }

    void accept3(Visitor v, A arg0, A arg1){ v.visit(arg0, arg1, this); }
    void accept3(Visitor v, A arg0, B arg1){ v.visit(arg0, arg1, this); }
    void accept3(Visitor v, B arg0, B arg1){ v.visit(arg0, arg1, this); }
}

class B implements Visitable {
    void accept1(Visitor v, Visitable arg1, Visitable arg2) { v.visit(this, arg1, arg2); }

    void accept2(Visitor v, A arg0, Visitable arg2) { v.visit(arg0, this, arg2); }
    void accept2(Visitor v, B arg0, Visitable arg2) { v.visit(arg0, this, arg2); }

    void accept3(Visitor v, A arg0, A arg1){ /* zgresitev */ }
    void accept3(Visitor v, A arg0, B arg1){ /* zgresitev */ }
    void accept3(Visitor v, B arg0, B arg1){ /* zgresitev */ }
}
```

Izsek kode 4.5: Primer metod *sprejmi* in *obišči* za tri metode in dva razreda.

Vmesni nivoji Metode *sprejmi* i-tega nivoja so implementirane v vseh razredih, ki v enem od primerkov metod nastopajo kot i-ti parameter. Poleg obiskovalca kot parameter prejmejo kombinacijo razrešenih in nerazrešenih tipov. Prvih $i-1$ tipov je bilo razrešenih v prejšnjih nivojih, zato ima prvih $i-1$ parametrov že znane tipe, preostali pa so tipa `Visitable` in bodo razrešeni na kasnejših nivojih. Metode imajo na predhodnih mestih lahko različne kombinacije tipov, zato na vsakem nivoju potrebujemo eno metodo *sprejmi* za vsako kombinacijo prvih $i-1$ parametrov. Taka metoda ponovno pokliče obiskovalčevo metodo *obišči*, ki ji poda že znane tipe, doda objekt `this`, s čimer razreši tip i-tega parametra, in nato poda še preostale objekte tipa `Visitable`.

Metode *obišči* na i-tem nivoju so enake kot na prvem nivoju – na $i+1$ parametru pokličejo metodo *sprejmi* z nivoja $i+1$, podajo pa ji obiskovalca (`this`), razrešene objekte do vključno i-tega in nerazrešene objekte od $i+2$ naprej. Prav tako je metod *obišči* toliko, kolikor je med primerki metod različnih kombinacij tipov na prvih i mestih.

Zadnji nivo Metode *sprejmi* na zadnjem nivoju med parametri nimajo več nerazrešenih tipov. Na obiskovalcu pokličejo metodo `obišči`, ki ji podajo vse razrešene parametre – tiste, ki

so jih prejele od prejšnjih nivojev in zadnjega kot objekt `this`.

Metode *obišči* na zadnjem nivoju imajo enak seznam parametrov kot originalni primerki metod, saj so vsi tipi že razrešeni. Taka metoda *obišči* mora le poklicati pripadajoč primerek metode in ji podati vse svoje parametre. Če metoda vrača rezultat, si ga obiskovalec na tem mestu zapomni, da ga lahko inicializacijska metoda, ki je poklicala prvo metodo *sprejmi*, vrne kot rezultat razpošiljanja.

4.6.3 Iskanje po drevesu

Drevo zgrajeno iz takih nivojev ima podobno obliko kot odločitveno drevo opisano v razdelku 4.4, kjer vsak nivo predstavlja en parameter metode. Posamezne vejitve se zgodijo, ko obiskovalec pokliče abstraktno metodo *sprejmi*, za katero izvajalno okolje ugotovi, v kateri razred spada. S klicem metode *sprejmi* na prvem nivoju se torej sproži iskanje v globino, ki išče najustreznejši primerek metode. Pri iskanju v globino lahko zaidemo v napačno vejo – v našem primeru se to zgodi takrat, ko se nekaj začetnih parametrov tesno ujema z enim od primerkov, kasnejši parametri pa ne.

Za ilustracijo vzemimo primer, prikazan v izseku kode 4.6, ki vsebuje hierarhijo razredov z vrstami živali in nekaj metod, ki sprejmejo različne kombinacije teh razredov.

```
class Zival {...}
class Sesalec extends Zival {...}
class Ptica extends Zival {...}
class Macek extends Sesalec {...}
class Pes extends Sesalec {...}

void metoda(Macek arg0, Pes arg1, Ptica arg2) {...}
void metoda(Sesalec arg0, Sesalec arg1, Sesalec arg2) {...}
void metoda(Sesalec arg0, Zival arg1, Zival arg2) {...}
void metoda(Zival arg0, Zival arg1, Zival arg2) {...}
```

Izsek kode 4.6: Primer razredov in metod, ki lahko povzročijo zgrešitve.

Za kombinacijo parametrov tipa `Macek`, `Pes` in `Pes` je najtesneje ujemajoča metoda druga (s kombinacijo tipov `Sesalec`, `Sesalec`, `Sesalec`). Zaradi ujemanja prvih dveh parametrov prva dva nivoja metod *sprejmi* in *obišči* vodita v vejo, ki pelje proti prvi metodi, tretji parameter pa se z njo ne ujema, kar pomeni, da pride do zgrešitve pri iskanju. Zaradi tega je treba zagotoviti mehanizem za vračanje po drevesu navzgor, ki bo v primeru neujemanja to sporočil prejšnjemu nivoju in tako iskanje usmeril v naslednjo vejo. Pri prejšnjih dveh načinih simuliranja razpošiljanja posebnega mehanizma za to nismo potrebovali, saj smo se v primeru neujemanja preprosto premaknili naprej, dokler nismo našli rešitve.

Vračanje lahko implementiramo tako, da metode *sprejmi* in *obišči* vračajo indikator tipa `boolean`, ki nakazuje, ali je bilo razpošiljanje uspešno ali ne. Ko v listu drevesa metoda *obišči* izvede ustrezno najdeno metodo, si zapomni njen rezultat, nato pa vrne vrednost `true`. Do

zgrešitve pride, če gre za metode *sprejmi* iz vmesnika *Visitable*, ki jih nismo generirali po postopku iz prejšnjega razdelka – to so manjkajoče veje v drevesu. Take metode ne vsebujejo klica ustrezne metode *obišči*, ampak vračajo vrednost `false`. Vse preostale metode preprosto vračajo rezultat svojega poddrevesa.

Ko imamo v vseh metodah indikatorje, ki povejo, ali je razpošiljanje v posameznem poddrevesu uspelo, se je treba na njih še ustrezno odzvati. V vsaki metodi *sprejmi* se preveri rezultat poddrevesa – če je negativen, je treba iskanje preusmeriti v naslednjo najtesneje ujemajočo vejo, kar se stori tako, da se pokliče metoda *sprejmi* na staršu tipa (spremenljivka `super`). Za tipe brez starša, ki tega ne morejo storiti, zgrešitev pomeni, da so bile preizkušene že vse možnosti za ta parameter in da je težava na enem izmed prejšnjih nivojev, zato preprosto vrnejo `false`. Če je rezultat poddrevesa pozitiven, metoda vrne vrednost `true`, da s tem o uspehu obvesti še nivoje pred seboj. Na vrhu drevesa v inicializacijski metodi, ki pokliče prvo metodo *sprejmi*, rezultat *true* pomeni, da je razpošiljanje uspelo, `false` pa, da razpošiljanja ni bilo možno izvesti in program zato javi napako.

```
class Macek extends Zival implements Visitable{

    boolean accept1(Visitor v, Visitable arg1, Visitable arg2) {
        if (v.visit1(this, arg1, arg2)) {
            return true; // poddrevo se ujema, vrni true
        } else {
            return super.accept1(v, arg1, arg2); // poskusi drugo vejo
        }
    }

    boolean accept2(Visitor v, Macek arg0, Visitable arg2) {
        // zgresitev - Macek nikoli ne nastopa kot 2. parameter
        return super.accept2(v, arg0, arg2);
    }
    ...
}

class Zival implements Visitable{

    boolean accept1(Visitor v, Visitable arg1, Visitable arg2) {
        if (v.visit1(this, arg1, arg2)) {
            return true; // poddrevo se ujema, vrni true
        } else {
            return false // poddrevo se ne ujema, vrni false
        }
    boolean accept2(Visitor v, Macek arg0, Visitable arg2) {
        if (visit2(arg0, this, arg2)) {
            return true; // poddrevo se ujema, vrni true
        } else {
            return false; // poddrevo se ne ujema, vrni false
        }
    }
}
```

Izsek kode 4.7: Nadzor zgrešitev v metodah *sprejmi*.

V izseku kode 4.7 je prikazanih nekaj metod za razred `Macek` in njegov nadtip `Zival`, ki vsebujejo mehanizme za nadzor zgrešitev. Razred `Macek` ob zgrešitvi pokliče `super.accept`, s

čimer preusmeri razpošiljanje v drugo vejo, razred `Zival`, ki je na vrhu hierarhije, pa v primeru zgrešitve le vrne vrednost `false`. Celoten potek razpošiljanja z obiskovalcem, kjer se zgodi več zgrešitev, je podrobno opisan na sliki 4.4.

4.6.4 Generiranje kode za obiskovalca

Generiranje kode med obdelavo anotacij je pri tem načinu razpošiljanja nekoliko bolj zapleteno, saj moramo generirati razred `Visitor` in vmesnik `Visitable`, moramo pa tudi dodati kodo v vse razrede, ki lahko nastopajo kot parametri metod.

V prvem koraku tako kot pri ostalih dveh izvedbah zgradimo razpršeno tabelo modelov in primerkov označenih metod. Nato glede na kombinacije tipov parametrov v najdenih primerkih generiramo opise vseh možnih metod *sprejmi* v obliki razredov `AcceptMethod`. Na i-tem nivoju razpošiljanja generiramo eno metodo za vsako kombinacijo prvih $i-1$ parametrov, ki se pojavi med primerki. Ob tem si zapomnimo, v katerih razredih bo vsaka izmed metod sodelovala pri razpošiljanju, saj jo bomo morali v njem implementirati. Iz dobljenih opisov metod nato generiramo razreda `Visitor` in `Visitable`, ki ju dodamo razredu, kjer je definiran model metode.

V drugem koraku obdelamo vse razrede, ki nastopajo kot parametri v najdenih metodah in jih lahko najdemo s pomočjo anotacije `@MultiDispatchVisitable`. Vsak razred obdelamo v treh korakih:

- Razred označimo kot implementacijo vmesnika `Visitable`.
- V razred dodamo metode *sprejmi*, ki so zanj relevantne (naslednji parameter bo morda tega tipa). Telo takih metod je generirano po postopku opisanem v razdelku 4.6.2. Za potrebe odzivanja na zgrešitve preverimo, ali ima obdelovani razred nadtip, ki prav tako razširja razred `Visitable`. Glede na to klic metode *obišči* izvedemo na dva različna načina, kot je opisano v razdelku 4.6.3.
- Generiramo metode, ki jih vmesnik `Visitable` vsebuje, a za razred niso relevantne. V njih takoj pokličemo ekvivalentno metodo sprejmi v razredu `super` oziroma vrnemo `false`, če tak razred ne obstaja.

V naslednjem koraku generiramo metodo, ki ustvari obiskovalca in pokliče prvo metodo *sprejmi*, s čimer sproži razpošiljanje. Ta inicializacijska metoda mora v primeru, da model pričakuje vračanje vrednosti, tudi vrniti rezultat, ki si ga je obiskovalec zapomnil ob izvedbi najdene metode.

V zadnjem koraku vse klice primerkov metode zamenjamo s klici inicializacijske metode iz prejšnjega koraka, podobno kot smo to storili pri ostalih dveh implementacijah razpošiljanja.

Izvorna koda:

```
Zival arg0 = new Macek();
Zival arg1 = new Macek();
Zival arg2 = new Ptica();

metoda(arg0, arg1, arg2);
```

Namesto klica metode se izvede logika za razpošiljanje:

1. Ustvari se obiskovalec `Visitor v = new Visitor()`
2. `arg0.accept1(v, arg1, arg2)`, izvajalno okolje klic pošlje v razred `Macek`
3. `v.visit1(this, arg1, arg2)`, prvi parameter je bil določen kot `Macek`
   - 3.1 Obiskovalec pozna razrede:
     `arg0: Macek, arg1: Visitable, arg2: Visitable`
   - 3.2 Obiskovalec kliče `arg1.accept2(this, arg0, arg2)`, JVM klic pošlje v razred `Macek`
   - 3.3 `Macek` nikoli ne nastopa kot drugi parameter, zato kliče `accept2` na nadtipu `Sesalec`
   - 3.4 `Sesalec` ni drugi parameter, če je `Macek` prvi, zato kliče `accept2` na nadtipu `Zival`
   - 3.5 `Zival` ni nikoli drugi parameter, zato metoda `accept2` vrne `false`
   - 3.6 `false` se po skladu vrne do metode iz koraka 3 v razredu `Macek`
4. Namesto koraka 3 se pokliče `super.accept1(v, arg1, arg2)` na nadtipu `Sesalec`
5. `v.visit1(this, arg1, arg2)`, prvi parameter je bil določen kot `Sesalec`
   - 5.1 Obiskovalec pozna razrede:
     `arg0: Sesalec, arg1: Visitable, arg2: Visitable`
   - 5.2 Obiskovalec kliče `arg1.accept2(this, arg0, arg2)`, okolje klic pošlje v razred `Macek`
   - 5.3 `Macek` ne nastopa kot drugi parameter, zato pošlje klic na nadtip `Sesalec`
   - 5.4 `v.visit2(arg0, this, arg2)`, drugi parameter je bil določen kot `Sesalec`
     - 5.4.1 Obiskovalec pozna razrede:
       `arg0: Sesalec, arg1: Sesalec, arg2: Visitable`
     - 5.4.2 `arg2.accept3(v, arg0, arg1)`, izvajalno okolje klic pošlje v razred `Ptica`
     - 5.4.3 `Ptica` ne more biti tretji parameter, če sta prva dva `Sesalec`, kliče nadtip `Zival`
     - 5.4.4 `Zival` ne more biti tretji parameter, če sta prva dva `Sesalec`, zato vrne `false`
     - 5.4.5 `false` se vrne po skladu do koraka 5.4 v razredu `Sesalec`
   - 5.5 Namesto koraka 5.4 se pokliče `arg1.accept2(this, arg0, arg2)` na nadtipu `Zival`
   - 5.6 `v.visit2(arg0, this, arg2)`, drugi parameter je bil določen kot `Zival`
     - 5.6.1 Obiskovalec pozna razrede:
       `arg0: Sesalec, arg1: Zival, arg2: Visitable`
     - 5.6.2 `arg2.accept3(v, arg0, arg1)`, izvajalno okolje klic pošlje v razred `Ptica`
     - 5.6.3 `Ptica` ne nastopa v taki kombinaciji parametrov, zato pošlje klic nadtipu `Zival`
     - 5.6.4 `v.visit3(arg0, arg1, this)`, tretji parameter je bil določen kot `Zival`
     - 5.6.5 Obiskovalec pozna vse tri razrede:
       `arg0: Sesalec, arg1: Zival, arg2: Zival`
     - 5.6.6 Obiskovalec pokliče metodo `metoda(Sesalec arg0, Zival arg1, Zival arg2)`

Slika 4.4: Opis poteka razpošiljanja z večnivojskim obiskovalcem, kjer pride do več zgrešitev. Uporabljeni so razredi in metode iz izseka kode 4.6.

Poglavje 5

Primerjava različic knjižnice

Tri različice knjižnice oziroma tri pristope k simuliranju večkratnega razpošiljanja smo eksperimentalno primerjali glede na čas izvajanja, čas prevajanja (hitrost obdelovanja anotacij) in velikost prevedenih programov. Vsi eksperimenti so bili izvedeni na procesorju Intel Core i7 s štirimi jedri, operacijskem sistemu macOS 10.13 in z JDK različice 1.8. Eksperimentalno primerjanje, kar zajema tudi generiranje in prevajanje vseh testnih primerov, se je izvajalo približno šest ur. Koda za generiranje testnih primerov, njihovo izvajanje in izris grafov je objavljena v sklopu implementirane knjižnice na `https://github.com/thenejcar/AnnotationDispatch/tree/master/Testing`.

5.1 Generiranje testnih primerov

Za eksperimentalno primerjanje so bili uporabljeni programi, ustvarjeni z generatorjem testnih primerov. Ta primere generira glede na 8 podanih vrednosti:

- Število modelov metod, ki določa, koliko različnih modelov bo v testnem programu prisotnih oziroma koliko različnih imen metod bo uporabljenih.
- Število primerkov metod, ki pove, koliko različnih primerkov metod bo imel vsak izmed generiranih modelov. To število torej pomeni, koliko definiranih metod v programu bo imelo enako ime.
- Število parametrov, ki jih bodo imele generirane metode in ki določa mestnost razpošiljanja – pri metodah z enim parametrom se izvaja dvojno razpošiljanje (prejemnik in en dodaten parameter), pri metodah z dvema trojno in tako dalje.
- Globina, širina in število razrednih hierarhij, iz katere bodo vzeti parametri metod. Vsaka izmed hierarhij se generira tako, da najprej ustvarimo korenski razred, nato pa iz njega izpeljemo njegove podrazrede. Postopek rekurzivno ponavljamo na izpeljanih razredih,

dokler ne dosežemo zahtevane globine. Število izpeljanih razredov je na vsakem koraku naključno izbrano med nič in podano vrednostjo za širino.

- Parameter, ki določa, ali bodo metode tipa `void` ali bodo vračale vrednosti tipa `String`.
- Število klicev vsakega modela metode, ki se bodo v testnem programu izvedli.

Privzete vrednosti zgoraj naštetih parametrov generatorja, ki so bile uporabljene pri eksperimentih, so bile 1 model metode, 5 primerkov, 2 parametra, globina, širina in število razrednih hierarhij pa so bile nastavljeni na 3. Ko smo generirali testno množico za enega izmed eksperimentov, smo pri prvem od generiranih primerov uporabili privzete vrednosti, pri ostalih pa smo eno izmed vrednosti spreminjali, da se je s tem pokazal njen vpliv na razpošiljanje. Kakšne so bile vrednosti generatorja za posamezne testne primere, je opisano v razdelku 5.2.

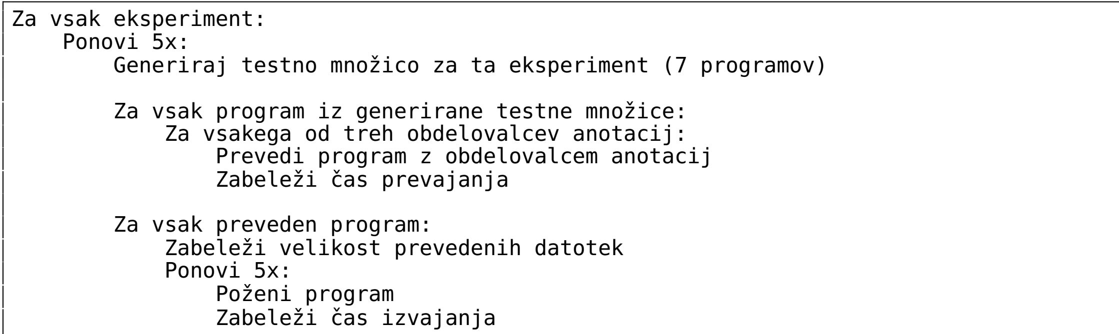
Zadnji dve vrednosti se med posameznimi testnimi primeri nista spreminjali – vse metode so vračale vrednost tipa `String`, število klicev metod pa je bilo v vseh primerih izbrano tako, da je vsak program vseboval sto različnih klicev, ne glede na število modelov in primerkov metod.

Posamezen generiran program je relativno enostaven in v celoti vsebovan v eni datoteki. Najprej so definirani razredi, ki bodo uporabljeni kot parametri v metodah. Njim sledijo definicije vseh generiranih metod, nato pa še metoda `main`, znotraj katere poteka glavni del programa. Metoda `main` se začne z definicijo spremenljivk, ki bodo nastopale kot parametri v metodah, zanje pa so vedno uporabljeni tipi z vrha razrednih hierarhij. Dejanski dinamični tipi objektov, ki se v njih shranijo (razredi uporabljenih konstruktorjev), so izbrani naključno izmed tipov izpeljanih iz statičnega tipa spremenljivke. Definicijam spremenljivk sledita dve `for` zanki, v katerih se izvede sto klicev generiranih metod. Prva zanka se izvede z 10 000 ponovitvami in je namenjena ogrevanju izvajalnega okolja – JVM operacije, ki se večkrat izvajajo, optimizira s sprotnim prevajanjem, zaradi česar prvih nekaj klicev vsake metode traja dlje kot kasnejši klici. Čas izvajanja merimo v drugi zanki, ki prav tako vsebuje vseh sto klicev, izvede pa se z 200 000 ponovitvami. Na koncu program izpiše trajanje izvedbe druge zanke v nanosekundah, ki torej predstavlja čas za 20 milijonov razpošiljanj.

5.2 Vrste in izvedba eksperimentov

Eksperimentalno primerjanje je bilo sestavljeno iz petih različnih eksperimentov, izmed katerih je bil vsak namenjen opazovanju vpliva ene od lastnosti programa, ki lahko vpliva na hitrost razpošiljanja oziroma prevajanja ali na velikost prevedene kode. Potek posameznega eksperimenta je prikazan na sliki 5.1.

Za vsak eksperiment se je najprej generirala testna množica, sestavljena iz sedmih različnih programov, ki so se med seboj razlikovali le v eni izmed nastavitev generatorja primerov, tako da smo to lastnost lahko neposredno primerjali z meritvami testnih indikatorjev. S kakšnimi



```
Za vsak eksperiment:
    Ponovi 5x:
        Generiraj testno množico za ta eksperiment (7 programov)

        Za vsak program iz generirane testne množice:
            Za vsakega od treh obdelovalcev anotacij:
                Prevedi program z obdelovalcem anotacij
                Zabeleži čas prevajanja

        Za vsak preveden program:
            Zabeleži velikost prevedenih datotek
            Ponovi 5x:
                Poženi program
                Zabeleži čas izvajanja
```

Slika 5.1: Potek eksperimentalnega primerjanja.

nastavitvami generatorja so se generirale posamezne testne množice, je opisano v podrazdelkih 5.2.1 do 5.2.4. Programi iz testne množice so se nato prevedli z uporabo vseh treh obdelovalcev anotacij, pri tem pa smo zabeležili čas prevajanja. Vsak od 21 prevedenih programov, ki smo jih tako dobili, se je nato petkrat pognal, ob vsaki izvedbi pa se je zabeležil čas izvajanja, ki ga je program izpisal. Celoten postopek se je ponovil petkrat, pri čemer se je na začetku vsake ponovitve tudi testna množica generirala na novo.

5.2.1 Vpliv števila parametrov

S tem eksperimentom želimo preveriti, kakšen vpliv ima število parametrov metode, za katero se razpošiljanje izvaja. Pričakujemo lahko, da bo število parametrov bistveno vplivalo tako na hitrost izvajanja kot na hitrost prevajanja in količino prevedene kode, saj vsak izmed parametrov pomeni en dodaten tip, ki ga mora razpošiljanje prepoznati.

Testna množica je bila sestavljena iz sedmih programov, kjer je bil prvi generiran s privzetimi vrednostmi (2 parametra), pri drugem smo število parametrov nastavili na ena, pri ostalih petih pa na vrednost, višjo od privzete (3, 5, 10, 15 in 20 parametrov). Tako smo dobili sedem programov, ki so vsebovali po en model metode s petimi primerki, ki pa je imel v vsakem programu drugačno število parametrov.

5.2.2 Vpliv oblike razredne hierarhije

Pri tem eksperimentu opazujemo vpliv globine in višine razredne hierarhije, v kateri nastopajo razredi, uporabljeni kot parametri v naših metodah, zato potrebujemo dva eksperimenta z različnima testnima množicama. Pričakujemo lahko, da bo oblika razredne hierarhije vplivala tako na izvajanje kot na prevajanje, saj bo logika za razpošiljanje zaradi več možnih tipov morda delovala počasneje. Prav tako vrstni red izbiranja metod pri vseh treh načinih razpošiljanja temelji na iskanju v globino, zaradi česar bo morda oblika hierarhije vplivala na obnašanje razpošiljanja.

Za vsakega izmed dveh testov smo generirali po sedem programov. Nastavitve generatorja, ki se nanašajo na metode, smo pustili na privzetih vrednostih, spreminjali pa smo vrednosti za hierarhije razredov, ki nastopajo kot njihovi parametri. Vsak testni program je imel tako po en model s dvema parametra in petimi primerki, razlika med njimi pa je bila v hierarhiji tipov, ki so bili v njih uporabljeni. V prvi testni množici je imel eden izmed programov privzeto obliko razredne hierarhije (globina in širina 3), pri dveh smo globino zmanjšali (1 in 2 nivoja), pri štirih pa povečali (4, 5, 8 in 10 nivojev). Druga množica je bila zelo podobna, le da smo spreminjali širino oziroma največje dovoljeno število otrok v razredni hierarhiji – eden izmed programov je imel ponovno privzeto obliko (globina in širina 3), dva sta imela ožje hierarhije (1 in 2 otroka na razred), štirje pa širše (4, 5, 8 in 10 otrok na razred). Tako smo dobili štirinajst različnih programov, pri čemer se je prvih sedem razlikovalo le po globini, drugih sedem pa po širini razredne hierarhije.

5.2.3 Vpliv števila modelov metod v programu

S tem eksperimentom želimo preveriti, kako na razpošiljanje vpliva število modelov metod v programu. Pri razpošiljanju se kandidate za optimalno metodo vedno izbira le med primerki istega modela, saj imajo vsi enako ime in število parametrov, kar pomeni, da je razpošiljanje neodvisno od ostalih modelov. Zaradi tega več vzporednih modelov metode na hitrost izvajanja ne bi smelo vplivati, lahko pa vpliva na prevajanje, saj ima vsak model svojo razpošiljevalno logiko, ki jo je treba med prevajanjem zgraditi.

V testni množici smo imeli ponovno sedem programov, prvi je bil generiran s privzetimi vrednostmi, pri ostalih pa smo število modelov povečali na 2, 4, 5, 10, 20 in 25. Tako smo dobili programe, ki so vsebovali modele s po dvema parametroma, petimi primerki metod in enako velikimi razrednimi hierarhijami, le da jih je bilo v nekaterih programih več, v drugih pa manj.

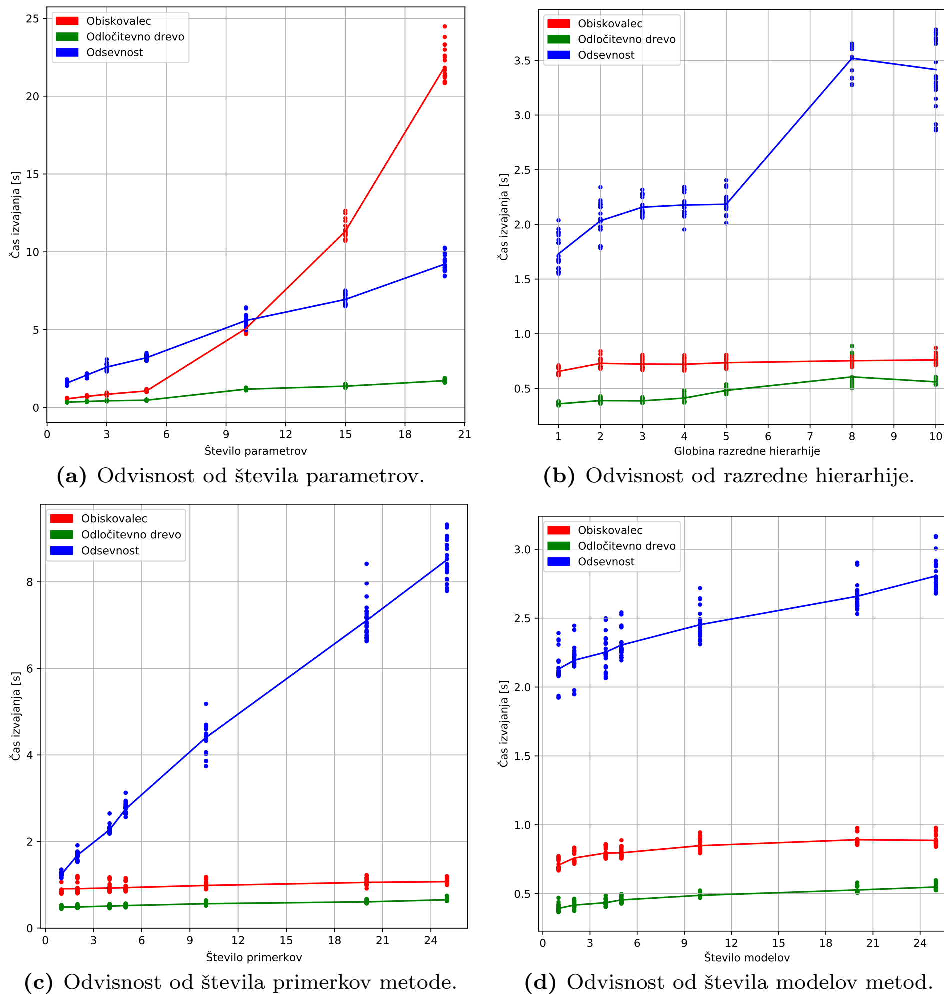
5.2.4 Vpliv števila primerkov modela

S tem eksperimentom lahko opazujemo, kakšen vpliv ima število definiranih metod, ki pripadajo nekemu modelu in med katerimi je potrebno med razpošiljanjem izbrati. Pričakujemo lahko, da bo vpliv na razpošiljanje podoben kot ob spreminjanju števila parametrov pri enem izmed prejšnjih testov, saj število primerkov prav tako vpliva na zapletenost logike za razpošiljanje. Tako lahko sklepamo, da se bo upočasnilo tako izvajanje kot prevajanje.

Testna množica je bila ponovno sestavljena iz sedmih programov, kjer je bil eden generiran s privzetimi vrednostmi (en model s 5 primerki), trije so bili generirani z manjšim (1, 2 in 4 primerki na model), trije pa z večjim številom primerkov (10, 20 in 25 primerkov na model). Vsak program je imel torej po en model z dvema parametroma in hierarhijo razredov širine in globine 3, razlikovali pa so se po številu metod (primerkov modela), ki so bile v njih definirane.

5.3 Hitrost izvajanja

Prva lastnost načina razpošiljanja, ki nas zanima, je hitrost izvajanja prevedenega programa. Po občutku bi lahko pričakovali, da je najpočasnejši način razpošiljanje z odsevnostjo, saj je sam mehanizem proženja metod na ta način precej počasen. Za razpošiljanje z obiskovalcem bi pričakovali hitro izvajanje, saj je večina logike izvedena z mehanizmom enojnega razpošiljanja, ki je implementiran v izvajalnem okolju in zato gotovo močno optimiziran.



(a) Odvisnost od števila parametrov.

(b) Odvisnost od razredne hierarhije.

(c) Odvisnost od števila primerkov metode.

(d) Odvisnost od števila modelov metod.

Slika 5.2: Grafi za hitrost izvajanja.

Slika 5.2 prikazuje hitrost izvajanja pri vseh štirih testih – barva ponazarja enega izmed načinov razpošiljanja, vsaka točka predstavlja enega od pognanih testov, črta pa prikazuje njihove povprečne vrednosti. Ker sta višina in globina razredne hierarhije na čas izvajanja vplivali skoraj popolnoma enako, so v rezultatih vključeni samo grafi odvisnosti od globine. To velja tudi za

čas prevajanja in velikost končnih programov, iz česar lahko sklepamo, da razpošiljanje verjetno ni občutljivo na obliko hierarhije, ampak zgolj na število razredov v njej.

Na grafu 5.2a, ki prikazuje vpliv števila parametrov, lahko vidimo, da čas izvajanja za tri načine narašča različno hitro. Za nizke vrednosti (do pet parametrov) razpošiljanje z obiskovalcem in z odločitvenim drevesom delujeta približno enako hitro, medtem ko je razpošiljanje z odsevnostjo nekoliko počasnejše. Razpošiljanje z obiskovalcem je od treh načinov najbolj občutljivo na povečevanje števila parametrov, saj se pri višjih vrednostih čas izvajanja zelo hitro dvigne in je pri primerih z več kot deset parametri že daljši kot pri razpošiljanju z odsevnostjo. To pomeni, da je za metode z manj kot pet parametri – kar je večina metod, s katerimi se pri programiranju srečujemo – hitrost razpošiljanje z obiskovalcem sprejemljiva, pri metodah z veliko parametri pa je edina dobra izbira razpošiljanje z odločitvenim drevesom.

Če pogledamo graf 5.2c, ki prikazuje vpliv števila primerkov, lahko vidimo, da hitrost izvajanja za razpošiljanje z odsevnostjo in odločitvenim drevesom narašča podobno hitro kot pri prejšnjem grafu, časi za razpošiljanje z obiskovalcem pa tokrat ostanejo nizki tudi za višje vrednosti. Najverjetnejša razlaga je, da pri načinu z odsevnostjo metode po vrsti iščemo v tabeli, zaradi česar je čas linearno odvisen od števila primerkov metod, pri ostalih dveh načinih pa v drevesni strukturi, kar pomeni, da je čas odvisen od logaritma števila primerkov metod.

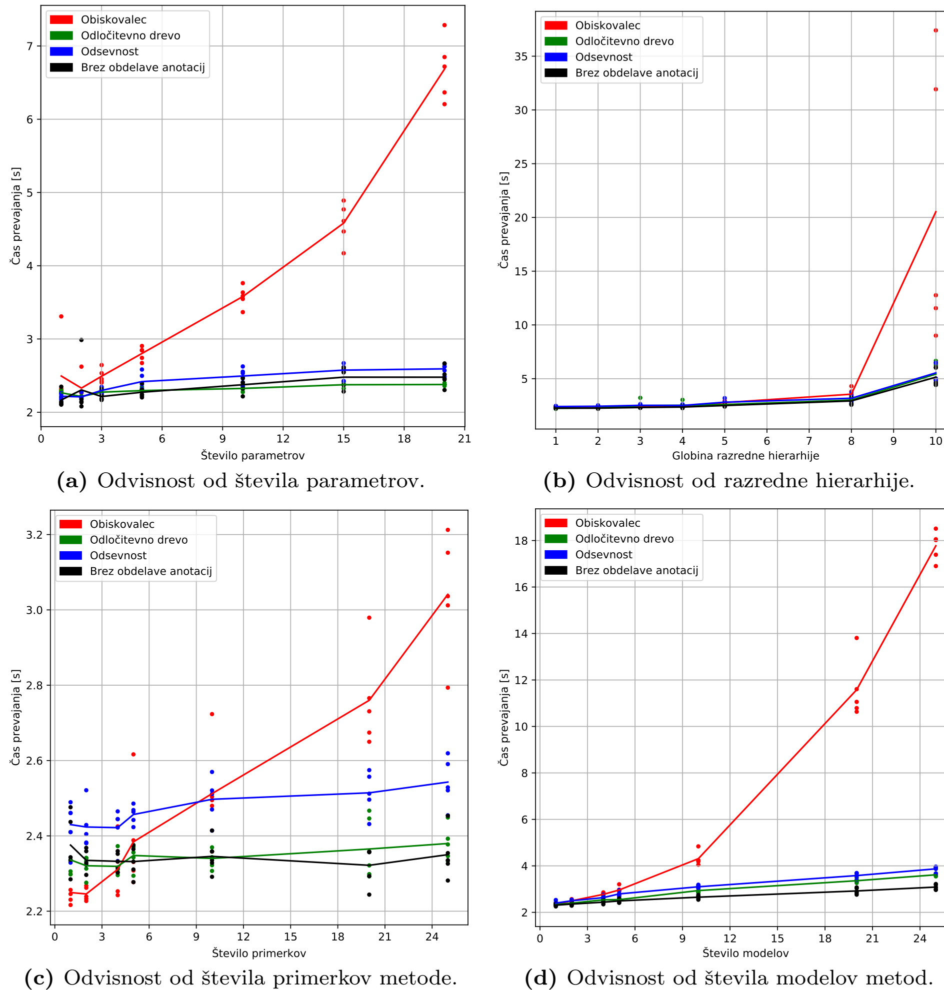
Na grafih na desni strani (5.2b in 5.2d) je situacija podobna kot pri odvisnosti od števila primerkov – časi za razpošiljanje z odločitvenim drevesom in obiskovalcem so nizki in blizu skupaj, časi za razpošiljanje z odsevnostjo pa so opazno višji. Razlika je predvsem v tem, da so tokrat vsi časi precej nižji, najvišji izmed njih je pri približno 3,5 sekunde. Presenetljiv je vpliv števila modelov, za katerega bi pričakovali, da na hitrost izvajanja ne vpliva, saj se razpošiljanje za vsak model izvaja neodvisno. Časi rahlo naraščajo, vendar se naraščanje pri razpošiljanju z odločitvenim drevesom in obiskovalcem kmalu ustavi. Če si pogledamo eksperimente pri višjih vrednostih oziroma z večjim številom parametrov v metodi (ker je prikazano v razdelku 5.6), lahko vidimo, da se tudi časi za razpošiljanje z odsevnostjo kasneje izravnajo. Vzrokov za to je lahko več, na primer različne optimizacje v JVM, ki za manjše število modelov lahko pohitrijo izvajanje, vendar bi bilo treba za točno razlago izvesti še dodatne teste oziroma podrobneje analizirati izvajanje.

Iz dobljenih rezultatov lahko ugotovimo, da je v vseh primerih najhitrejši enostavni način z odločitvenim drevesom, vendar pri metodah z dovolj nizkim številom parametrov obiskovalec ne zaostaja daleč za njim.

5.4 Hitrost prevajanja

Druga lastnost različnih pristopov k razpošiljanju, ki nas zanima, je, kako dolgo traja obdelovanje anotacij med prevajanjem. Tu pričakujemo, da bo način z obiskovalcem najpočasnejši, saj je

potrebno generirati veliko metod v veliko različnih razredih, način z odsevnostjo pa najhitrejši, saj je precej bolj neodvisen od posameznega primera. Grafi na sliki 5.3 so izrisani na podoben način kot pri merjenju časa izvajanja, s črno pa je za primerjavo označen tudi čas prevajanja brez obdelave anotacij.



(a) Odvisnost od števila parametrov.

(b) Odvisnost od razredne hierarhije.

(c) Odvisnost od števila primerkov metode.

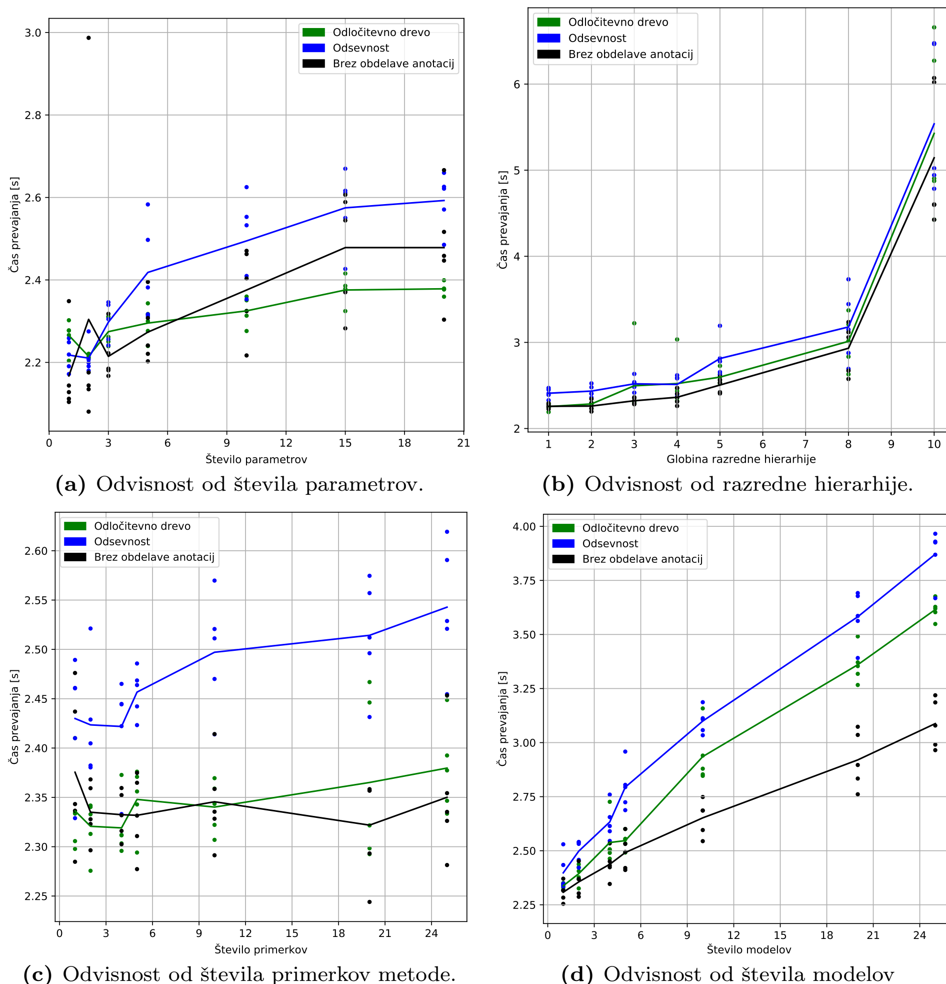
(d) Odvisnost od števila modelov metod.

Slika 5.3: Grafi za hitrost prevajanja.

Vidimo lahko, da čas pri pristopu z obiskovalcem narašča bistveno hitreje kot pri ostalih dveh načinih, ki sta praktično enako hitra kot prevajanje brez obdelave anotacij. Na trajanje prevajanja pri uporabi razpošiljanja z obiskovalcem vplivata predvsem število metod *sprejmi* in število različnih razredov, ki nastopajo kot parametri in v katerih je treba te metode implementirati. Pri ustvarjanju metod *sprejmi* je treba za vsak nivo upoštevati vse možne kombinacije predhodnih parametrov, na kar vplivata tako število parametrov kot število primerkov posamezne metode, zato grafa 5.3a in 5.3d nista preveč presenetljiva.

Velikost razredne hierarhije ima na čas prevajanja pri uporabi obiskovalca velik vpliv, saj pri globini 10 čas poskoči tudi nad 30 sekund. Razlog je verjetno v tem, da vsi razredi sodelujejo v vseh fazah generiranja kode, tudi če na koncu niso uporabljeni, ali pa vsebujejo le po eno metodo *sprejmi*. Tako tudi razredi, ki na čas izvajanja niso imeli nobenega vpliva, na čas prevajanja vplivajo.

Tudi večje število modelov obdelovanje anotacij z obiskovalcem upočasni bolj kot ostala dva načina – čeprav se razpošiljanje za vsak model izvaja ločeno, zaradi česar število modelov ne vpliva na hitrost izvajanja, se med prevajanjem koda za vse metode generira hkrati. Zato za čas prevajanja dodaten model pomeni isto, kot če bi podvojili število primerkov enega modela.



(a) Odvisnost od števila parametrov.

(b) Odvisnost od razredne hierarhije.

(c) Odvisnost od števila primerkov metode.

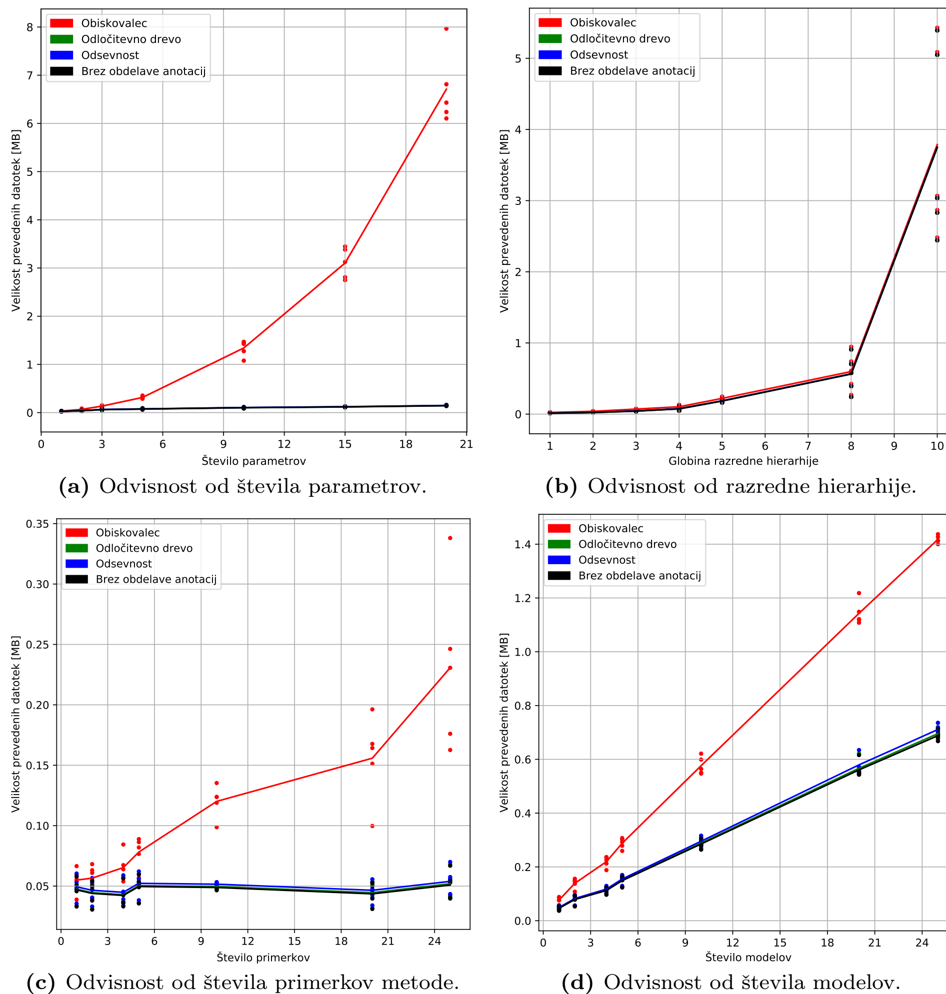
(d) Odvisnost od števila modelov

Slika 5.4: Grafi za hitrost prevajanja brez razpošiljanja z obiskovalcem.

Na sliki 5.4 so prikazani isti rezultati še brez razpošiljanja z obiskovalcem. Vidimo lahko, da so razlike med preostalima dvema načinoma in običajnim prevajanjem zelo majhne, oblike grafa

pa si praktično sledijo. To pomeni, da oba načina minimalno vplivata na čas prevajanja in da naraščanje na grafu izhaja že iz samih testnih primerov.

5.5 Velikost prevedenih programov



(a) Odvisnost od števila parametrov.

(b) Odvisnost od razredne hierarhije.

(c) Odvisnost od števila primerkov metode.

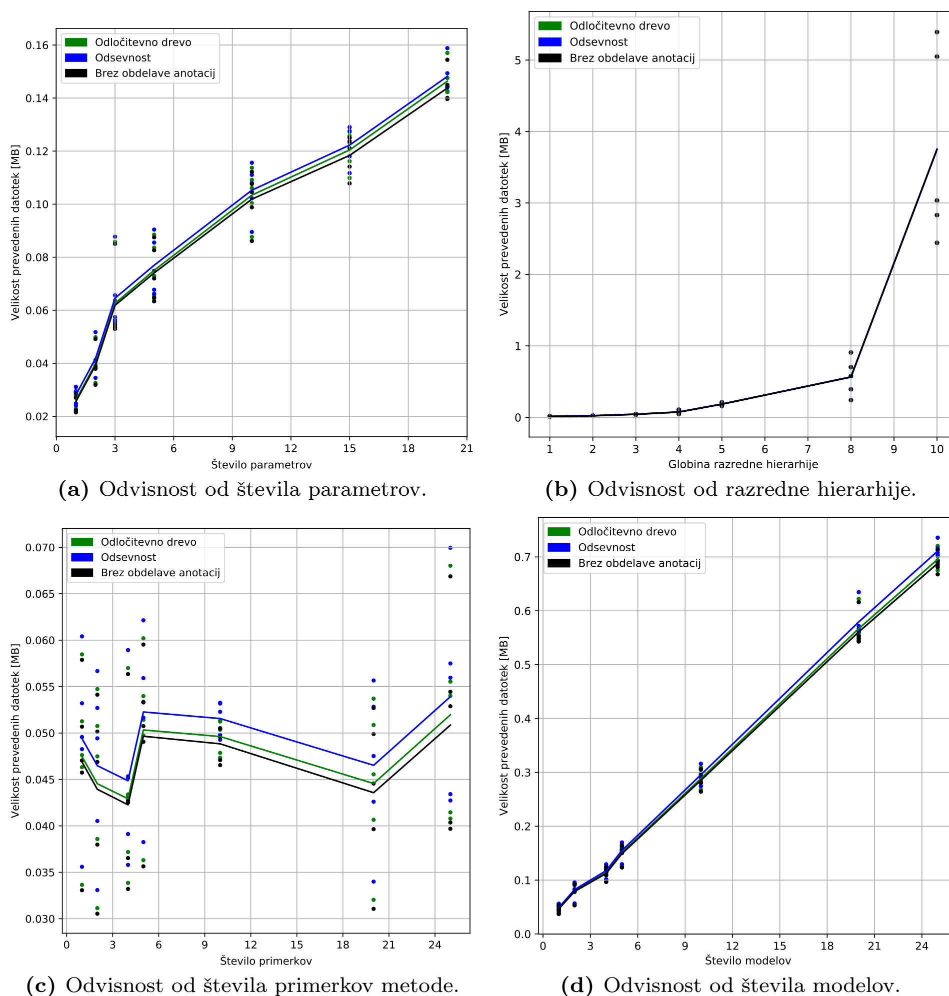
(d) Odvisnost od števila modelov.

Slika 5.5: Grafi za velikost prevedenih programov.

Na velikost prevedene kode pri programiranju običajno ne pazimo tako močno kot na primer na hitrost izvajanja, vendar je v kontekstu te naloge pomembna, saj generiramo dodatno kodo, za katero bi želeli, da je ni preveč. Glede na število dodatnih metod gre pričakovati, da bo obdelovanje anotacij z obiskovalcem ustvarilo največ dodatne kode. Grafi na sliki 5.5 so izrisani na enak način kot pri prejšnjih dveh meritvah.

Mertive za način z obiskovalcem tudi tokrat najhitreje naraščajo. To je najverjetneje posledica števila generiranih metod *sprejmi*, ki je neposredno povezano s povečanjem števila parametrov in primerkov metode. Podobno na velikost kode vpliva število različnih modelov, saj je za vsakega treba ustvariti novega obiskovalca, vmesnik za obiskovane razrede in novo skupino metod *sprejmi*, zato tudi njihovo število močno vpliva na velikost kode.

Zanimivo je, da povečevanje razredne hierarhije ne vpliva na velikost kode nobenega od načinov razpošiljanja, saj vsi ostajajo približno enaki kot pri prevajanju brez obdelave anotacij. To velja tudi pri razpošiljanju z obiskovalcem, saj večina razredov ne nastopa v nobeni metodi in zato ne vsebuje metod *sprejmi*, ki so sicer glavni del novo generirane kode. Tako v nasprotju s časom prevajanja, ki ga je število različnih razredov močno upočasnilo, veliko število praznih razredov na velikost kode ne vpliva.



(a) Odvisnost od števila parametrov.

(b) Odvisnost od razredne hierarhije.

(c) Odvisnost od števila primerkov metode.

(d) Odvisnost od števila modelov.

Slika 5.6: Primerjava velikosti prevedenih programov brez načina z obiskovalcem.

Če pogledamo še grafe za velikost kode na sliki 5.6, kjer niso prikazane meritve za razpošiljanje z obiskovalcem, lahko vidimo, da sta preostala dva načina zelo blizu velikostim brez obdelave anotacij. Koda za način z odsevnostjo je sicer malenkost večja, vendar se vse tri velikosti razlikujejo le za nekaj kB tudi pri zelo velikih programih. Iz tega lahko sklepamo, da noben od teh dveh načinov na velikost kode nima bistvenega vpliva.

5.6 Rezultati primerjanja

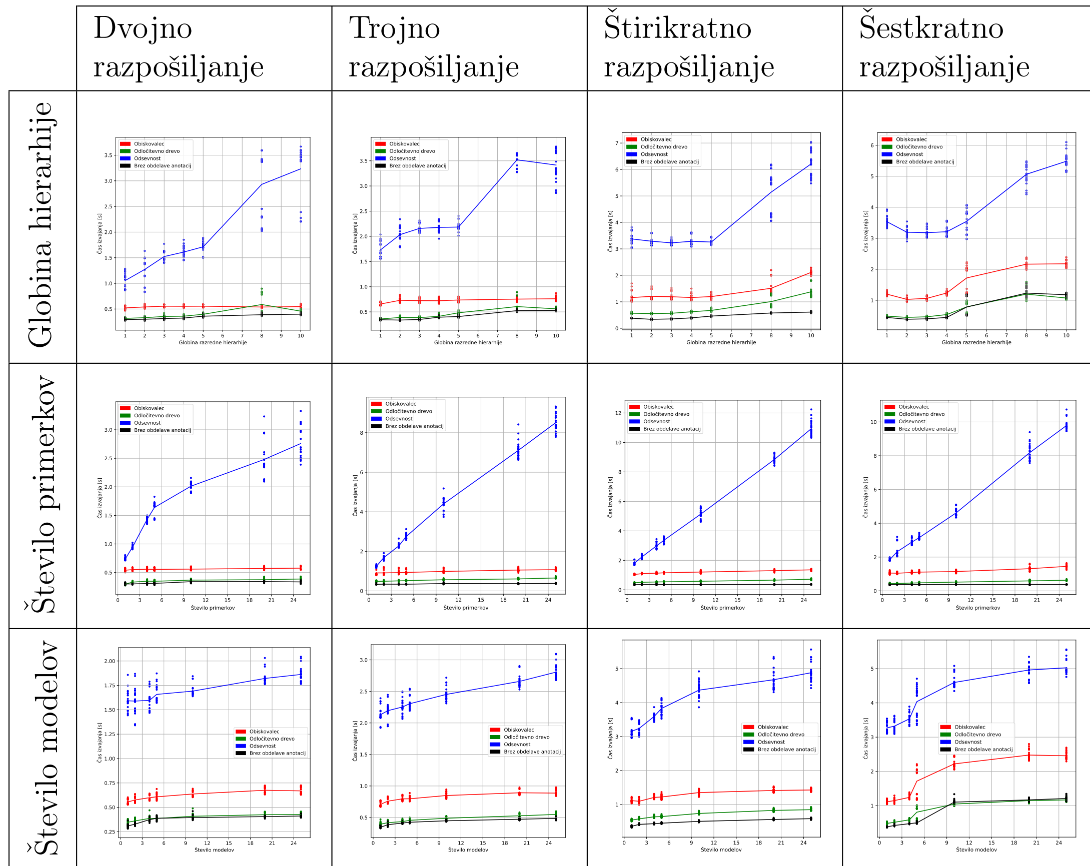
Z eksperimenti smo primerjali tri načine razpošiljanja, da lahko na podlagi rezultatov ugotovimo, katera izmed treh različic naše knjižnice je najprimernejša za uporabo, hkrati pa si lahko z njimi pomagamo tudi, če želimo enega od mehanizmov razpošiljanja napisati ročno.

V rezultatih lahko vidimo, da je odločitveno drevo z neposrednim preverjanjem tipov glede na vse tri opazovane lastnosti – hitrost izvajanja, čas prevajanja in velikost programa – najboljši mehanizem za simuliranje razpošiljanja v Javi. S tem mehanizmom se je razpošiljanje v vseh primerih izvedlo najhitreje, na čas prevajanja in velikost končnega programa pa njegova uporaba ni imela bistvenega vpliva. Od treh obdelovalcev anotacij, ki so v naši knjižnici na voljo, je torej za resno uporabo najprimernejši obdelovalec `ProcessorTree`, ki uporablja ta način razpošiljanja.

Razpošiljanje pri novem mehanizmu z večnivojskim obiskovalcem se je izvajalo hitro, če je bilo število parametrov dovolj nizko. Na sliki 5.2a smo videli, da je hitrost razpošiljanja s tem mehanizmom primerljiva s hitrostjo razpošiljanja preko odločitvenega drevesa za metode z manj kot pet parametri – torej do šestkratnega razpošiljanja. Za večino metod, ki jih uporabljamo v običajnih programih, je tak način razpošiljanja torej primeren, v primeru metod z večjim številom parametrov pa postane prepočasen. Nekoliko slabše se je izkazal pri ostalih dveh kriterijih, saj je močno vplival na čas prevajanja in velikost prevedene kode. Tudi za metode z nizkim številom parametrov je čas obdelovanja anotacij pri mehanizmu z večnivojskim obiskovalcem občutljiv na velikost hierarhije tipov parametrov in na število uporabljenih modelov metod. Velikost prevedene kode je prav tako v vseh primerih opazno večja, kot pri ostalih dveh mehanizmih za razpošiljanje, kar je posledica velikega števila dodatnih metod. Zaradi označevanja vseh obiskovanih razredov z anotacijo `@MultidispatchVisitable` zahteva uporaba tega načina tudi nekoliko več dela, kar v kombinaciji z nekoliko slabšimi rezultati pomeni, da izmed treh načinov razpošiljanja ni najboljša izbira. Ne glede na to smo pokazali, da je tak način razpošiljanja za realne primere uporaben in bi ga brez težav lahko uporabili v programu.

Razpošiljanje z uporabo odsevnosti je pri metodah z običajnim številom parametrov (do desetkratnega razpošiljanja) najpočasnejši od vseh treh načinov. Zelo podoben pristop z nekaj optimizacijami, kot je predpomnenje najdenih metod, se uporablja v nekaterih drugih programskih jezikih, zato lahko predvidevamo, da je tak rezultat posledica neučinkovitih mehanizmov odsevnosti v Javi. Jeziki z vgrajeno podporo za večkratno razpošiljanje se lahko zanašajo na

učinkovite postopke za hranjenje metod v tabeli in njihovo izvajanje, taki postopki v Javi pa so žal veliko počasnejši. Glede ostalih dveh kriterijev je ta način sicer enakovreden razpošiljanju z odločitvenim drevesom, saj njegova uporaba na hitrost prevajanja in velikost končne kode skorajda ne vpliva. V primerjavi z drugima dvema pristopoma, različica knjižnice, ki uporablja odsevnost torej ni dobra izbira. Podoben pristop je vseeno uporaben v situacijah, kjer ga lahko izvedemo na učinkovit način brez uporabe odsevnosti (na primer v jezikih Julia in Clojure), ali kjer programa ne moremo spreminjati med prevajanjem (na primer v knjižnici JMMF).



Slika 5.7: Primerjava hitrosti izvajanja v odvisnosti od razredne hierarhije, števila primerkov in števila modelov pri uporabi metod z različnimi števili parametrov.

Vsi prikazani eksperimenti (razen preverjanja vpliva števila parametrov) so uporabljali trojno razpošiljanje – razpošiljanje metod z dvema parametroma. Da bi preverili, kako se rezultati spremenijo pri metodah z manj ali več parametri, smo vse teste ponovili tudi za metode s po enim, tremi in petimi parametri, s čimer smo izvajali dvojno, štirikratno in šestkratno razpošiljanje. Na sliki 5.7 so prikazani grafi hitrosti izvajanja v odvisnosti od globine razredne hierarhije, števila primerkov in števila modelov za metode z različnimi števili parametrov. Na grafih so s

črno prikazani tudi časi, kjer programa nismo spremenili z obdelovanjem anotacij. Pri njih se večkratno razpošiljanje torej ni izvedlo, zaradi česar je bil tudi sam potek programov drugačen, vseeno pa nam te meritve lahko služijo kot primerjava z običajnimi javanskimi metodami.

Na sliki lahko vidimo, da so oblike grafov tudi pri spreminjanju mestnosti razpošiljanja skorajda enake, le da se pri večjem številu parametrov vsi grafi nekoliko raztegnejo vzdolž y osi in da je vpliv števila modelov in globine razredne hierarhije vse bolj izrazit. Vidimo tudi, da višji časi pri šestkratnem razpošiljanju, predvsem v odvisnosti od razredne hierarhije in števila modelov, ne izvirajo zgolj iz večkratnega razpošiljanja – tudi pri primeru brez obdelave anotacij časi tam rahlo naraščajo, kar pomeni, da je izvajanje metod z velikim številom parametrov že v osnovi počasnejše.

Za rezultate, ki smo jih v tem poglavju opisovali za metode z dvema parametroma, lahko torej predvidevamo, da veljajo za tudi za ostale metode s petimi parametri ali manj, kar zajema večino metod, ki se običajno uporabljajo.

Poglavje 6

Zaključek in nadaljne delo

V nalogi smo si pogledali koncepte razpošiljanja, nekaj obstoječih implementacij razpošiljanja v Javi in drugih jezikih ter predstavili tri mehanizme, s katerimi se večkratno razpošiljanje lahko simulira. Spoznali smo orodja za delo z anotacijami in naredili kratek pregled njihove običajne uporabe ter kako lahko z njimi med prevajanjem spreminjamo kodo. Za tem smo si pogledali še implementacijo knjižnice, ki z omenjenimi mehanizmi med prevajanjem v Javo doda večkratno razpošiljanje, ter primerjali tri njene različne izvedbe.

Z implementacijo knjižnice smo pokazali, da je kodo za razpošiljanje možno generirati med prevajanjem, zaradi česar ni treba bistveno spremeniti načina programiranja. Za vpeljavo večkratnega razpošiljanje v program je potrebno dodati le nekaj anotacij, kar je z vidika poseganja v obstoječo kodo programerju najbolj prijazna izvedba podobne knjižnice do zdaj – kot smo videli, obstoječe rešitve zahtevajo poseben prevajalnik ali uporabo povsem novih koncept deklariranja in klicanja metod.

Pokazali smo tudi, da je nov pristop z večnivojskim obiskovalcem izvedljiv, in predstavili njegovo implementacijo, česar prej v literaturi ni mogoče najti. Pri metodah z realnim številom parametrov je presegel hitrost razpošiljanja z odsevnostjo, ni pa dosegel hitrosti, ki jih dobimo pri razpošiljanju z odločitvenim drevesom. Njegova največja slabost je zapletenost, kar posledično pomeni daljši čas prevajanja in večje končne programe, je pa tudi bolj občutljiv na število parametrov v metodi kot ostala dva načina. Večja kompleksnost kode in posledično daljše prevajanje je bilo sicer pričakovano, saj smo z večjim številom metod poskušali izkoristiti mehanizme za razpošiljanje, ki so vgrajeni v Javo.

Ugotovili smo, da je odločitveno drevo z neposrednim preverjanjem tipov zaenkrat najhitrejši mehanizem za simuliranje razpošiljanja v Javi. Uporaba tega načina razpošiljanja prav tako ni imela bistvenega vpliva na čas prevajanja in velikost končnega programa. Tak pristop je zato izmed treh predstavljenih različic verjetno najboljša izbira, če bi želeli izdelano knjižnico uporabljati v resne namene.

Knjižnica v treh izvedbah deluje in je pripravljena na uporabo, vendar ostaja še prostor za nadaljnje delo. Prva izboljšava bi bila prilagoditev knjižnice na novejšo izdajo Jave, na primer JDK 11, ki je naslednja izdaja Jave s daljšim časom podpore (angl. *long term support*). Za to večje spremembe niso potrebne, spremenilo pa se je nekaj razredov v paketu `com.sun.tools.javac`, ki bi se jim bilo treba prilagoditi. Druga izboljšava bi bila testiranje na realnih programih, saj s testi izvedenimi v sklopu te naloge najbrž niso bile pokrite vse možne kombinacije metod in njihovih parametrov. Različnih kombinacij je zelo veliko, saj morajo biti idealno podprti prav vsi možni programi, ki jih v Javi lahko napišemo.

Slike

4.1 Primer uporabe obdelovalca anotacij z ukazom `javac`. 25
4.2 Primer uporabe obdelovalca anotacij s sistemom maven. 25
4.3 Shema razpošiljanja z odločitvenim drevesom za štiri metode. 27
4.4 Opis poteka razpošiljanja z večnivojskim obiskovalcem, kjer pride do več zgrešitev. Uporabljeni so razredi in metode iz izseka kode 4.6. 37

5.1 Potek eksperimentalnega primerjanja. 41
5.2 Grafi za hitrost izvajanja. 43
5.3 Grafi za hitrost prevajanja. 45
5.4 Grafi za hitrost prevajanja brez razpošiljanja z obiskovalcem. 46
5.5 Grafi za velikost prevedenih programov. 47
5.6 Primerjava velikosti prevedenih programov brez načina z obiskovalcem. 48
5.7 Primerjava hitrosti izvajanja v odvisnosti od razredne hierarhije, števila primerkov in števila modelov pri uporabi metod z različnimi števili parametrov. 50

Izseki kode

2.1 Primer, ki uporablja enojno razpošiljanje. 5
2.2 Primer, kjer bi potrebovali večkratno razpošiljanje. 5
2.3 Primer uporabe večkratnega razpošiljanja. 6
2.4 Primer programa v jeziku MultiJava. 8
2.5 Primer uporabe knjižnice JMMF. 9
2.6 Primer programa z JPred razširitvami. 10
2.7 Primer multimetode v jeziku Clojure. 11
2.8 Primer večkratnega razpošiljanja v jeziku Julia. 13
2.9 Primer parametrov tipa in unij v jeziku Julia. 14
3.1 Primer gradnje sintaksnega drevesa. 20
4.1 Drevo za štiri primerke metode s tremi tipi parametrov. 28
4.2 Inicializacijska metoda pri razpošiljanju z odsevnostjo. 29
4.3 Razpošiljevalna metoda pri razpošiljanju z odsevnostjo. 30
4.4 Običajni vzorec obiskovalec. 31
4.5 Primer metod *sprejmi* in *obišči* za tri metode in dva razreda. 33
4.6 Primer razredov in metod, ki lahko povzročijo zgrešitve. 34
4.7 Nadzor zgrešitev v metodah *sprejmi*. 35

Literatura

[1] Oracle America, Inc. and/or its affiliates, Java Virtual Machine Specification, Chapter 6: The Java Virtual Machine Instruction Set. Dostopno na https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html#jvms-6.5.invokevirtual (avgust 2018).

[2] Oracle America, Inc. and/or its affiliates. OpenJDK Wiki, Virtual Calls. Dostopno na https://wiki.openjdk.java.net/display/HotSpot/VirtualCalls (avgust 2018).

[3] Oracle America, Inc. and/or its affiliates. The Java Tutorials, Lesson: Annotations. Dostopno na https://docs.oracle.com/javase/tutorial/java/annotations/ (avgust 2018).

[4] J. Mihelič, I. Rožanc. Techniques for traversal operation on an object structure: a comparison, 26th Central European Conference on Information and Intelligent Systems, 2015, str. 213–220.

[5] C. Clifton, et al. MultiJava: Design rationale, compiler implementation, and applications. ACM Transactions on Programming Languages and Systems (TOPLAS) 28, 2006, str. 517-575.

[6] The MultiJava Team. The MultiJava Project. Dostopno na http://multijava.sourceforge.net/ (avgust 2018).

[7] R. Forax, E. Duris, G. Roussel. Java Multi-Method Framework. 37th International Conference on Technology of Object-Oriented Languages and Systems, IEEE, 2000, str. 45–56.

[8] R. Forax. Java MultiMethod Framework. Dostopno na http://www-igm.univ-mlv.fr/~forax/works/jmmf/index.html (avgust 2018).

[9] T. Millstein, C. Frost, J. Ryder, A. Warth. Expressive and Modular Predicate Dispatch for Java. ACM Transactions on Programming Languages and Systems (TOPLAS) 31.2, 2009, čl. 7

[10] T. Millstein, C. Frost, J. Ryder, A. Warth. JPred: Practical Predicate Dispatch for Java. Dostopno na http://web.cs.ucla.edu/~todd/research/jpred.html (avgust 2018).

[11] R. Hickey. Clojure reference: Multimethods and Hierarchies. Dostopno na `https://clojure.org/reference/multimethods` (avgust 2018).

[12] J. Bezanson. Abstraction in Technical Computing. Doktorska disertacija, Massachusetts Institute of Technology, 2015.

[13] J. Bezanson, S. Karpinski, V. B. Shah and other contributors. Julia 1.0 documentation. Dostopno na `https://docs.julialang.org/en/v1/` (avgust 2018).

[14] F. Ortin, J. Quiroga, J. M. Redondo, M. Garcia. Attaining multiple dispatch in widespread object-oriented languages. Dyna 81(186), National University of Colombia at Medellín, 2014, str. 242-250.

[15] Oracle America, Inc. and/or its affiliates. OpenJDK JDK8 langtools source code. Dostopno na `http://hg.openjdk.java.net/jdk8/jdk8/langtools/` (avgust 2018).

[16] The Project Lombok Authors. Project Lombok. Dostopno na `https://projectlombok.org/` (avgust 2018).