Univerza v Mariboru

**Fakulteta za elektrotehniko,
računalništvo in informatiko**

Razvoj konceptov dinamičnega metaprogramiranja v statično tipiziranem objektno usmerjenem programskem jeziku

DOKTORSKA DISERTACIJA

Maribor, marec 2009 Sašo Greiner

Avtor: Sašo Greiner

Naslov: Razvoj konceptov dinamičnega metaprogramiranja v statično tipiziranem objektno usmerjenem programskem jeziku

UDK: 004.3:004.43 (043.2)

Ključne besede: programski jeziki, behavioralna refleksija, strukturalna refleksija, metaprogramiranje, statično tipiziranje

ZAHVALA

Zahvaljujem se vsem, ki so prispevali k zaključku tega dela.

Naslov: Razvoj konceptov dinamičnega metaprogramiranja v statično tipiziranem objektno usmerjenem programskem jeziku

UDK: 004.3:004.43 (043.2)

Ključne besede: programski jeziki, behavioralna refleksija, strukturalna refleksija, metaprogramiranje, statično tipiziranje

POVZETEK

V delu predstavljamo načrtovanje in implementacijo metaprogramskih konceptov čistega objektno usmerjenega programskega jezika. Jezik, ki je nadgradnja obstoječega jezika Z_0, imenujemo Zero. Temeljna ideja jezika je združitev konceptov dinamičnega metaprogramiranja s statičnim oz. hibridnim sistemom tipov. Metaprogramski model jezika temelji na behavioralni in strukturalni refleksiji, ki omogočata spremembe obnašanja in strukture programov v času izvajanja. Metafunkcionalnost je realizirana v metarazredih, ki dopolnjujejo obstoječo razredno hierarhijo. Čisti objektni model je razširjen na metode, s čimer so omogočene anonimne metode in metode višjega reda. V okviru statičnega sistema tipov vpeljemo metodne tipe, ki omogočajo tipiziranje metod v času prevajanja in ohranjanje varnosti tipov v času izvajanja.

Title: Developing the concepts of dynamic metaprogramming in a statically-typed object-oriented programming language
UDK: 004.3:004.43 (043.2)

Keywords: programming languages, behavioural reflection, structural reflection, metaprogramming, static typing

ABSTRACT

We present the design and implementation of metaprogramming concepts in a pure object-oriented programming language. We call this language, which is an upgrade of an existing language Z_0, the language Zero. The fundamental idea of Zero is to aggregate the concepts of dynamic metaprogramming with a static/hybrid typing system. The metaprogramming model is based on behavioural and structural reflection, which enable the modification of behaviour and structure of programs at run-time. The metafunctionality is realised in metaclasses which are part of an existing class hierarchy. The pure object model is extended to methods which enables anonymous methods and higher-order methods. As part of a static typing system, we introduce method types, which allow typing methods during compile-time and provide a means to keep method types sound and sane at run-time.

Kazalo

1 Predgovor **1**

2 Objektno usmerjeni programski jeziki **7**

2.1 Razredi in objekti 8

2.1.1 Abstrakcija in koncept objektnega tipa 13

2.1.2 Abstrakcija enkapsulacije 16

2.2 Dedovanje in virtualni mehanizmi 17

2.2.1 Klasifikacija dedovanja 21

2.2.2 Večkratno dedovanje 26

2.2.3 Razpošiljanje 28

2.2.4 Virtualni mehanizmi 28

2.3 Tipiziranje 31

2.3.1 Sistemi tipov 32

2.3.2 Formalne metode za opis sistemov tipiziranja 34

2.3.3 Teoretični vidik podtipov 37

2.3.4 Polimorfno tipiziranje 45

2.3.5 Parametrizacija tipov 46

2.3.6 Polimorfizem z omejitvami 47

3 Jezik Z_0 **49**

3.1 Jezikovni konstrukti 50

3.2 Objektni model 52

3.3 Sistem tipov 54

3.3.1 Primitivni tipi 55

3.4 Struktura vhodnih programov 58

3.5 Prvorazrednost jezikovnih konstruktov 60

3.6 Iterativni stavki 62

3.7 Manipulacija metod 62

3.8 Dedovanje 64

3.9 Dinamično tipiziranje s `Self` 66

4 Metaprogramiranje in refleksija **68**

4.1 Refleksija 70

4.1.1 Behavioralna refleksija 70

4.1.2 Strukturalna refleksija 71

4.2 Metaprogramski model 72

4.3 Metarazredi 74

5 Jezik Zero **76**

5.1 Razširitev jezika Z_0 na sintaktičnem nivoju 78

5.2 Vmesnik za uporabo zunanjega programskega koda 80

5.2.1 Implementacija klica 82

5.2.2 Funkcionalnost zunaj virtualnega okolja 83

5.3 Metainformacije obstoječih razredov 84

5.3.1 Metarazred `Class` 86

5.3.2 Metarazred `Closure` 90

5.3.3 Metarazred `Method` 96

5.4 Metaprogramiranje v jeziku Zero 98

5.4.1 Statična refleksija 98

5.4.2 Dinamična refleksija 100

5.5 Metode višjega reda 105

5.5.1 Teoretične osnove 106

5.5.2 Formalen zapis računa lambda 106

5.5.3 Metode višjega reda v imperativnem jeziku 107

5.5.4 Pretvorba med metodnimi tipi 109

5.5.5 Invokacija 110

5.5.6 Primerjava z drugimi statično tipiziranimi jeziki 111

6 Zaključek **112**

Slike

1 Iskanje metode 24
2 Struktura objekta. 65
3 Hierarhija v Smalltalku 74
4 Shema invokacije zunanje metode. 83
5 Razredna hierarhija krmilnih konstruktov. 85

1 Predgovor

Izmed množice visokonivojskih programskih jezikov, ki so krojili poglede na načrtovanje programske opreme zadnjih nekaj desetletij, bi stežka našli takega, ki bi v popolnosti zadoščal ciljem, za katere je bil v prvi vrsti implementiran. Cilj, smoter in zaenkrat še nekoliko idealizirana želja, je predstavitev problema računalniku v naravnem, človeku razumljivem jeziku [66], iz katerega bo računalnik sposoben razbrati korake rešitve. Značilnosti in prednosti programskega jezika so v nedvoumnosti pomena fraz, zgoščenem slovarju besed in sploh v preproščini izražanja. Za učinkovitost programskega jezika je potrebno zadostiti več ciljem. Prvi je vsekakor ta, da jezik zadošča nekim postavljenim specifikacijam. Drugi predpostavlja, da je programski jezik možno zadovoljivo učinkovito implementirati na ciljnem sistemu, kjer se bo uporabljal (v večini primerov govorimo o računalniku). Splošnonamenski programski jezik mora omogočati seveda tudi zapis poljubne rešljive naloge. Torej mora biti na nek način univerzalen. Ker je splošna univerzalnost prezahteven pogoj, se zadovoljimo s tem, da je jezik domensko-univerzalen, da torej omogoča zapis rešljive naloge zgolj z nekega ožjega področja. Programski jezik je orodje za razvoj in ima zelo velik vpliv na reševanje izbranega problema. Prav s tega stališča mora biti zasnovan z visoko stopnjo odgovornosti in preudarnosti. Programski jezik je netoleranten, saj zahteva neobhodno enoumnost v sintaksi in semantiki programskih stavkov oz. fraz.
Postopek načrtovanja jezika lahko upravičeno imenujemo večkriterijska optimizacija, saj je potrebno dosledno upoštevati veliko množico zahtev, ki se med seboj večkrat izključujejo. Najti zadovoljivo in povrhu optimalno pot med vsemi skrajnimi rešitvami, ki se ponujajo, pomeni kompleksno načrtovanje.

Večkrat pomanjkljivosti jezika postanejo očitne šele z njegovo uporabo v praktičnih aplikacijah, kjer moč in izraznost jezika dobita svojo komponento veljave. Če jezik odlikuje dovolj velika izrazna moč, je njegove omejitve z različnimi tehnikami velikokrat mogoče zaobiti. Kljub temu bi si želeli izraznosti na stopnji, kjer bi lahko vsak segment problema zapisali neposredno s tistim, kar jezik v svojem naboru programskih konstruktov nudi. Zanimivo dejstvo, ki postane jasno šele v poznih ciklih konkretnega

razvoja programske opreme je, da jezikovne pomanjkljivosti, ki zadevajo težave pri kodiranju posameznih algoritmov, prizadenejo celovit spekter programskih jezikov; od najpreprostejših s komaj omembe vredno množico mehanizmov do takih, za katere bi lahko dejali, da programerju nudijo resnično široko paleto jezikovnih mehanizmov. Deviacija med prvimi in slednjimi je zgolj v razsežnosti idej, ki jih je nemogoče ali vsaj zelo težko zapisati s tistim, kar programerju nudijo. Zagotovo velja, da prav ta razsežnost kroji razloček med jeziki, katere imenujemo izrazno močne in tistimi, ki so po svoji izraznosti šibki. Obstajajo pa jeziki, ki imajo sicer veliko izrazno moč ali kakšno drugo pomembno kvaliteto, a kritično zaostajajo na kakšnem drugem specifičnem področju. Zbirni jezik ima izjemno lastnost, da lahko programski kod optimiziramo, kot je tega sposoben malokateri prevajalnik. Toda izrazna moč zbirnika je zelo majhna. Po drugi strani ima objektno usmerjen programski jezik C++ z možnostjo abstrahiranja nad podatki, večkratnega dedovanja ter uporabniškega definiranja operatorjev izjemno veliko izrazno moč. Toda ta isti jezik močno zaostaja v nekaterih konceptih tipov in možnosti vpogleda v lastno izvajalno okolje. Združiti vse dobre lastnosti in zavreči vse slabe in jalove, je verjetno temeljna perspektiva v vsakem procesu načrtovanja novega programskega jezika. Na ta način bi dobili "perfekten" jezik, kar pa v realnem svetu žal ni mogoče. Mi se bomo v dobri veri v "lep jezik" osredotočili na implementacijo dotičnega objektno usmerjenega programskega jezika oziroma razširitev konceptov obstoječega jezika.

Pričujoče delo je logičen nasledek dognanj in sklepov, do katerih smo prišli v magistrski razpravi o implementaciji dinamičnih konceptov objektno usmerjenega jezika Z_0. Omenjeno delo bomo kronološko nadaljevali z nadgradnjo nekaterih že spoznanih izsledkov in njihovo kategorično vključitvijo v razumevanje novih konceptov, ki bodo temeljna nit naših raziskav. Čeprav gre v tej disertaciji za nadaljevanje obstoječih zaključkov, bomo vendarle vpeljali množico novih pojmov in idej, s katerimi se v predhodni razpravi nismo podrobneje seznanili niti srečali.

Zakaj sploh objektno usmerjen programski jezik? Sodoben svet učinkovitih programskih jezikov za razvoj praktičnih aplikacij nedvomno narekuje sledenje objektno usmerjeni paradigmi. Slednja vpeljuje široko množico konceptov, ki so, glede na uporabo

v posameznih jezikih, splošno uveljavljeni ali povsem specifične narave in vezani na dotičen programski jezik. Popolna objektizacija razvoja programske opreme ni več trendovsko pogojena, temveč je spričo obsega današnjih programskih rešitev postala nujna in obvezujoča. Desetletja razvoja v industriji so pokazala, da objektno načrtovanje ni trenutna muha zanesenjaških inženirjev akademskih krogov, ampak ima svojo izjemno veljavo prav v implementaciji povsem praktičnih rešitev. Omenjeno dejstvo izhaja iz tega, da so temeljni koncepti in osnovne ideje objektne usmerjenosti dovolj dovršeni za modeliranje večine entitet realnega sveta. Tehnika modeliranja teh entitet se tako neposredno preslika v objektni model načrtovanja z nekim programskim jezikom, ki omogoča načrtovanje z objektnimi abstrakcijami [44]. Vidno je, da je trend praktičnega objektnega programiranja zares zaživel šele s pojavitvijo jezika C++ in kasneje Jave [51]. Navkljub temu, da Java, še manj pa seveda C++, nista nova, so se trendi objektnega programiranja v polni meri pokazali razmeroma pozno. Razlog za tako impulziven naskok na uporabo objektnega načrtovanja in programiranja je ta, da objektno usmerjeni jeziki, v primerjavi s svojimi imperativnimi proceduralnimi predhodniki, nimajo večjih pomanjkljivosti. Poleg povsem pragmatične uporabe objektnega modeliranja, je pravtako tudi sama teorija objektov dokaj nadrobno raziskana. Še posebej nazorno so ta teoretska znanja vidna na področju abstraktnih podatkovnih tipov in njihovega polimorfnega obnašanja.

Kakorkoli obrnemo, praksa in izkušnje razvijalcev programske opreme so pokazale, da imajo tudi tako opevani in čaščeni objektno usmerjeni programski jeziki svoje pomanjkljivosti. Ker bo naše delo ozko fokusirano na jezik Zero [82], se bomo v svojih raziskavah osredotočili predvsem na težave in slabosti prav tega programskega jezika. Izbirali bi lahko izmed konceptov, ki bi jih jezik Zero lahko vključeval z namenom postati "dovršen" programski jezik. Dinamika tipov jezika že v trenutnem stanju jezika Z_0 omogoča široko aplikacijo na reševanje praktične problematike iz zahtev realnega sveta. Eden izmed zahtevnejših konceptov, ki v jeziku še ni implementiran, je zagotovo generičnost oz. splošnost tipov. Tipske relacije v Z_0 resda omogočajo maksimalno sposobnost dinamičnega tipiziranja, ki je še preverljiva v času prevajanja, toda najzahtevnejše oblike tipiziranja v močno tipiziranih programskih jezikih so primorane v uporabo generičnih mehanizmov. Ti nudijo krepko širše možnosti tipiziranja in kar

je enako pomembno, tipiziranje je mogoče na ta način implementirati zelo učinkovito. Drug, v današnjem času enako pomemben koncept, kadar govorimo o objektno usmerjenih programskih jezikih, je sposobnost objektov vpogleda vase. Govorimo o introspekciji ali refleksiji. Zaznavanje in poznavanje tistega, kar se dogaja v notranjosti živega okolja objektne množice. Možnost vpogleda v stanje lastnega izvršilnega okolja omogoča več dinamike, kot je je mogoče doseči z vsemi drugimi mehanizmi. Prav iz tega razloga smo se odločili, da bomo jezik Z_0 nadgradili z mehanizmi, ki bodo programerju omogočali izrabo tovrstnih idej z namenom povišati stopnjo prilagodljivosti posameznih funkcionalnosti v času izvajanja. Jezik Z_0 v trenutnem stanju pravzaprav že omogoča solidno stopnjo manipuliranja objektnega okolja v času izvajanja, saj vsebuje mehanizem dinamičnega spreminjanja metod. Z njim lahko metode objektov poljubno spreminjamo glede na zahteve, ki se popolnoma konkretizirajo šele v času, ko program že teče in so razredi že instancirani. Toda jezik Z_0 nima možnosti pridobitve refleksijskih informacij o trenutnem stanju okolja. Potrebno bo torej implementirati t.i. meta arhitekturo, ki bo omogočala izrabo meta informacij. Ker je jezik Z_0 čisti objekto usmerjen programski jezik, kjer je torej vsaka entiteta objekt, bo dovolj, da bodo meta informacije zgrajene na nivoju objekta. "Preprostejša" entiteta od objekta, kar zadeva programerja, v jeziku namreč ne obstaja. Sleherna meta arhitektura jezika bi naj omogočala vsaj branje meta informacij. Spreminjanje teh istih meta informacij pa ima precej daljnosežnejše učinke na izvajanje. Če na meta arhitekturo gledamo kot na osnovni oz. temeljni sloj virtualnega stroja, je jasno, da bo vsaka sprememba meta informacije vplivala na celotno izvajalno okolje virtualnega stroja. Z ozirom na zelo željeno lastnost po čimbolj "naravni" komunikaciji s programskim jezikom, lahko z gotovostjo trdimo, da meta arhitekura k tej lastnosti ne prispeva znatno. Stopnja abstraktnosti, s katero programer operira z opcijami specifičnega programskega jezika, ostaja povečini nespremenjena, saj meta arhitektura doprinaša povsod, razen k sami ortografiji jezika. Algoritem za hitro urejanje števil zapišemo na identičen način, ne glede na to, ali ima jezik podporo meta konceptov ali je povsem brez njih. Drugače je z izrazno močjo, ki jo sicer povezujemo s tem, kako lahko rešitev nekega problema v jeziku izrazimo. Velika izrazna moč pomeni, da lahko z obstoječimi jezikovnimi konstrukti realiziramo rešitve, ki bi bile z manjšo izrazno močjo nemogoče in na način,

ki bi sicer bil bolj zapleten. S tem lahko odgovorimo na vprašanje, kako meta koncepti vplivajo na izraznost programskega jezika. Meta arhitektura omogoča vpogled v izvajalno okolje in dinamiko objektnega prostora, ki postane konkreten šele v času izvajanja. Ker veliko parametrov v času prevajanja programa ni znanih, v času izvajanja potrebujemo mehanizem, kako pridobiti znanje o teh parametrih. Če je meta arhitektura jezika dovolj fino razdelana, se lahko dinamično, v trenutku, ko to potrebujemo, dokopljemo do informacij, ki so pogojene s trenutnim stanjem objektnega prostora v računalniku.

Ideja meta programiranja se je pojavila že v samem začetku epohe objektno usmerjenih jezikov, najprej s Smalltalkom in Lispom. Bolj kot sami koncepti metaprogramiranja, so v zadnjih letih napredovale implementacije le-teh, predvsem z razvojem zmogljive strojne opreme. Osnovna zamisel se spreminja zgolj v podrobnostih. Če povzamemo zmogljivosti meta programiranja, ki so bile podane v Smalltalku, vidimo, da so celo v današnjem času redki programski jeziki, ki omogočajo takšen spekter metafunkcionalnosti. Seveda ne gre prezreti, da je Smalltalk v osnovi interpretiran in dinamično tipiziran. Na paleti statično tipiziranih in prevedenih programskih jezikov je trend doseči vsaj del tistega, kar omogočajo dinamični in interpretirani jeziki. Ker je temeljni smoter našega raziskovanja obogatitev jezika Z_0 z meta arhitekturo, se bomo usmerili v koncepte, ki so dosegljivi v mejah statičnosti močno tipiziranega in prevedenega programskega jezika. Dejstvo, da ne bomo mogli implementirati vseh konceptov, ki bi jih želeli, se postavlja že na samem začetku. Obrnimo se torej na ideje, ki so dosegljive.

Z jezikom Zero želimo prikazati, da je statičen sistem tipov primeren za izvedbo konceptov dinamičnega metaprogramiranja. To hipotezo skušamo v pričujočem delu potrditi skozi pazljivo načrtovanje jezikovnih konceptov in njihovo dejansko implementacijo v obliki prevajalnika. Jezik Zero bo imel v polnosti realizirane ideje o branju in spreminjanju informacij trenutnega izvajalnega okolja. Spričo čiste objektizacije bo predstavitev vseh meta entitet poenotena. Skozi raziskovalni postopek bomo polagoma razvili načrt metaprogramskega modela jezika. Ideja metaprogramiranja in refleksije nam bosta vseskozi služili za temeljna koncepta, v okviru katerih bomo skušali umestiti metaprogramski model v obstoječ programski jezik. Metaprogramski model jezika Zero bomo

gradili na konceptih behavioralne in strukuralne refleksije, ki bosta omogočali spreminjanje obnašanja in strukture programov v času izvajanja. Ker gre za razredni jezik, bomo metafunkcionalnost realizirali v obliki metarazredov.

Delo je sestavljeno iz šestih poglavij. V drugem poglavju predstavimo temelje objektno usmerjenih programskih jezikov, kjer podamo koncepte objekta, razreda, dedovanja in tipiziranja. Ker gre v delu za razvoj tipiziranega jezika, posvečamo prav slednjemu konceptu največ pozornosti. Tretje poglavje podaja zgoščen opis jezika Z_0. Osredotočimo se na objektni model jezika, tipiziranje, dedovanje ter mehanizem manipulacije stanja objekta z nadomeščanjem metod. Četrto poglavje predstavlja splošno podano idejo metaprogramiranja in konceptov le-tega. Tukaj se usmerimo v predstavitev temeljnih konceptov in idej metaprogramiranja, kot so metaprogramski model, behavioralna in strukturalna refleksija ter metarazredi. Osredotočimo se na koncepte, ki pridejo v poštev predvsem v statično tipiziranih razrednih in prevedenih programskih jezikih.
Peto poglavje predstavlja jedro raziskovalnega dela. V njem predstavimo jezik Zero, pri katerem pričnemo s sintaktično nadgradnjo in se nato premaknemo v definicijo metarazredov. Pri tem natančno opišemo posamezen koncept metaprogramiranja v kontekstu našega jezika. Šesto poglavje sklene delo z jedrnatim povzetkom konceptov metaprogramiranja, ki so v jeziku implementirani. Podane so tudi opazke in dognanja, ki smo jih tekom naših raziskav uspeli spoznati. Podamo še nekaj idej, ki bi jih v prihodnosti morebiti vključili v jezik.

2 Objektno usmerjeni programski jeziki

Veliko je primernih izhodišč, iz katerih bi se lahko z dobrimi razlogi lotili razprave o tako pomembni veji računalništva, kot so objektno usmerjeni programski jeziki in njihovi koncepti. V pričujočem poglavju se bomo osredotočili predvsem na predstavitev in preučitev temeljnih aspektov statično tipiziranih, razrednih, objektno usmerjenih programskih jezikov, ki so pomembni tudi z gledišča meta arhitekture. Razmišljanje v objektnih smernicah sega desetletja v preteklost. V zgodnjih šestdesetih letih sta norveška inženirja Kristen Nygaard in Ole-Johan Dahl razvila prvi objektno usmerjen programski jezik Simula [78, 60]. Simula je postavila temelje objektne usmerjenosti v praktičnem razmišljanju načrtovanja programskih sistemov in je služila za glavno inspiracijo pri kasnejšem razvoju jezika Smalltalk [49]. Slednjega so v sedemdesetih letih razvili strokovnjaki družbe Xerox PARC. Smalltalk velja za prvi "čisti" objektno usmerjen programski jezik, saj je vpeljal koncepte objektizacije, ki pred tem niso bili implementirani.

Komponente programskega jezika je potrebno razumeti kot interaktivno celoto, ki jezik enoumno identificira in mu pridaja razpoznavne atribute. Posebna pozornost gre sistemom tipov, ki služijo kot temelj pri teoretiziranju in podajanju formalnih opisov objektno usmerjenih jezikov. Natančno razumevanje konceptov sistemov tipov v objektno usmerjenih programskih jezikih ni pomembno zgolj za formalne zapise jezikov, temveč vodi tudi k bolj prilagodljivim, razširljivim ter izrazno močnejšim programskim jezikom. Formalen opis je pri objektno usmerjenih jezikih izjemno pomemben, saj je semantika teh jezikov, kadar želimo opisati najpomembnejše koncepte, precej kompleksna.

Sistemi tipov pri statično tipiziranih objektno usmerjenih programskih jezikih igrajo ključno vlogo, saj se ravno v njihovi domeni nahaja večji del odgovornosti za učinkovito združitev fleksibilnosti tipiziranja in statične varnosti. Dinamični jeziki ugodijo prvi lastnosti, z drugo imajo več težav. Ker gre pri objektno usmerjenih jezikih predvsem za tipiziranje, velja za cilj preudarnega načrtovanja takšnega jezika doseči maksimalno izrazno moč sistema tipov ob maksimalni varnosti. Do velike izrazne moči, ki je sicer relativen pojem, lahko pridemo z bogatim sistemom tipov, ki ima le majhno število

omejitev. Sistem tipov mora biti v primeru statične varnosti formalno preverljiv v času prevajanja, kar pomeni, da se mora s formalnimi metodami dati dokazati, da v času izvajanja ne more priti do napake povezane s tipiziranjem.
Druga potrebna lastnost objektno usmerjenih programskih jezikov je dedovanje, katerega razumemo kot inkrementalni modifikacijski in primarni kompozicijski ali strukturalni mehanizem. Celovita namembnost dedovanja je izjemno široka, saj omogoča teoretično neskončne razsežnosti v modeliranju praktičnih entitet realnega sveta. Ker je dedovanje vendarle v tesno sklopljeni navezi s tipiziranjem, ga je potrebno ustrezno spojiti s sistemom tipov programskega jezika. Navkljub povezanosti s tipiziranjem pa koncept dedovanja verjetno najlažje razumemo v povezavi s specializacijo funkcionalnosti in podatkovnega strukturiranja. Ne glede za kakšno vrsto dedovanja gre, enkratno ali večkratno, starševsko ali nasledniško, ostaja njegova temeljna ideja enaka. Na tehničnem nivoju je funkcioniranje dedovanja izvedeno s pomočjo polimorfnega obnašanja metod. Mehanizem je nepogrešljiv, saj sta prav preko njega omogočeni specializacija in generalizacija ter abstrahiranje nad obnašanjem metod in objektov.
Temeljni koncepti, kot so objekti in razredi, tipiziranje, abstrakcija ter konkretizacija in tudi kanoničnost jezika, naj bodo osnovno vodilo pričujočega poglavja, ki bo služilo za prolog razumevanju programskega jezika Z_0 ter njegove razširitve v jezik Zero.

2.1 Razredi in objekti

Tradicionalni razvojni ciklus programske opreme je s spremembami, ki jih je s svojim univerzalističnim pogledom na modeliranje prinesel objektni model, doživel konkretno prevetritev ne zgolj v stranskih idejah, ampak v samih temeljih svoje filozofije. Narava, ki se izrazito odmika od nekdaj splošno priznanega proceduralnega abstrahiranja, je navdušila svet sodobnih in že manj sodobnih programskih razvojnih sistemov.

Objekti, kot jih razumemo v našem kontekstu, predstavljajo dogodke in entitete realnega sveta, med katerimi poteka interakcija. S programskega vidika objekti kapsulirajo stanje in vzorec obnašanja. Stanje je običajno predstavljeno z instančnimi spremenljivkami, obnašanje pa določeno z metodami, ki manipulirajo stanje. V objektnem modelu računamo z objekti, končni rezultat računanja pa predstavljajo stanja vseh sodelujočih

objektov. Strukturalno lahko na objekt gledamo kot na skupek podatkov in metod. Objekt si lažje predočimo, če poznamo koncept abstraktnega podatkovnega tipa, ki ni nič drugega kot množica podatkov in procedur, ki operirajo nad temi podatki. Razlika je na prvi pogled zelo zamegljena, vendar se je treba zavedati, da so abstraktni podatkovni tipi veliko preprostejši, saj ne poznajo pomena enkapsulacije s skrivanjem podatkov. Abstraktni podatkovni tipi prav tako ne poznajo ideje dedovanja, specializacije in konformance tipov. Podatke znotraj abstraktnega podatkovnega tipa lahko manipuliramo direktno ali posredno preko pripadajočih procedur, medtem ko lahko pri nekaterih objektno usmerjenih jezikih podatke spreminjamo izključno preko procedur. Pomembna je torej možnost skrivanja podatkov, do katerih želimo preprečiti dostop zunaj objekta. Ker objekti med seboj "učinkujejo" s pošiljanjem sporočil, lahko funkcionalno na objekt gledamo kot na abstraktni stroj, ki zna odgovarjati na določena sporočila. Sporočilo preprosto določa invokacijo specifične metode nad objektom. Množico sporočil, na katera zna objekt odgovoriti, imenujemo sporočilni vmesnik ali *protokol* objekta. Programski jezik Eiffel za označitev protokola uporablja *pogodbo*, medtem ko Java isto stvar imenuje kar vmesnik. V objektnem modelu procedure oz. funkcije, ki operirajo nad (enkapsuliranimi) podatki objekta, imenujemo metode. Objekt je abstraktna entiteta, ki jo ustvarimo spontano, brez posebne šablone, ali pa jo ustvarimo po nekem predpisu, tj. z uporabo neke šablone oz. načrta. V prvem primeru govorimo o objektnih jezikih, v drugem pa o objektno usmerjenih jezikih. Objektni jeziki poznajo torej samo objekte, katere ustvarjamo kot primerke in jih po potrebi kloniramo. Kloniranje je temeljna lastnost objektnih jezikov, saj omogoča inkrementalne (redkeje dekrementalne) spremembe že obstoječih objektov in s tem dedovanje.

Posebna veja objektnih jezikov so prototipni jeziki (prototype languages), v katerih ustvarjamo prototipne objekte in iz teh, s kloniranjem, povsem nove objekte. Objektni jeziki so povečini dinamični s stališča, da omogočajo dinamično spreminjanje instančnih spremenljivk. Zanimivejša lastnost je možnost dinamičnega spreminjanja metod, ki omogoča nadomeščanje obstoječe metode s povsem novo različico. Razlika med instančnimi spremenljivkami in metodami se tako praktično izniči. Klonirani objekti sicer lahko dinamično spreminjajo svoje stanje in metode, vendar njihova struktura ostane enaka. Da bi lahko dodali nove lastnosti, je potrebno uporabiti princip dedo-

vanja. Le-ta se v objektnih jezikih razlikuje od klasičnega razrednega dedovanja iz preprostega razloga – ni razredov. Metode za izvedbo dodajanja novih lastnosti in spreminjanja strukture objektov bomo podali v podpoglavju o dedovanju.
Spričo filozofsko in metodološko usmerjenih preučevanj so se pojavile pronicljivo postavljene ideje o tem, kateri objektno usmerjeni programski jeziki se s svojimi lastnostmi uvrščajo med "bolj" objektno naravnane [97]. Razložitev in predočitev teh razlik je bila uperjena predvsem nad globel med razrednimi objektno usmerjenimi in prototipnimi jeziki. Snovalci enih in drugih so s praktičnimi primeri in teoretičnimi razpravami upravičevali svoje poglede na prednosti tistih, ki so jih favorizirali. Kljub pristranskosti vsakega pogleda je potrebno povedati, da imata obe ideji, razredna in prototipna, mesta svoje kritike in poveličevanja. Namen takšnih razglabljanj bomo dobrovoljno prepustili teoretičnim demagogom in razpravljačem, sami pa se poslužili opisov bolj konkretnih oblik. Razločki med obojimi jeziki obstajajo v povsem konkretnih konceptih, a ontološka ideja objektizacije ostaja nespremenjena. Tako objektno usmerjeni kot objektni programski jeziki omogočajo abstrahiranje in dinamično selekcijo metod. Specifične implementacije posameznih jezikov kažejo, da prototipni programski jeziki omogočajo preprostejšo strukturo programov kot razredni jeziki. To izhaja iz dejstva, da prototipni jeziki ne poznajo razredov in imajo iz tega razloga manj sintaktičnih konstruktov za modeliranje objektnih abstrakcij. Toda ta preproščina ima s stališča učinkovitosti tudi svojo manj privlačno plat, saj objekti prevzemajo vlogo razredov in s tem vnašajo inkrementalne spremembe v čas izvajanja. V splošnem velja, da prototipni jeziki izkazujejo večjo prilagodljivost in naravnost kar zadeva tipiziranje. Dobro uveljavljena lastnost je tudi ta, da so prototipni jeziki povečini interpretirani in le redko prevajani, kar nadalje zvišuje njihovo kompetenco prilagajanja in zmožnost izrabe dinamičnih mehanizmov v času izvajanja. Razredni jeziki občutno šepajo pri prilagodljivosti v času izvajanja, vendar ne toliko zaradi razredne filozofije kot zaradi dejstva, da so zvečine prevedeni. Učinkovitost izvajanja ima v industrijskem svetu očitno znatnejšo vlogo, saj je trend v skoraj izključni uporabi prevedenih razrednih programskih jezikov, ki so po svoji naravi mnogo učinkovitejši od prototipnih. Navkljub obstoječim in vse prej kot zanemarljivim razlikam, oboje, razredne in prototipne, uvrščamo med "objektno naravnane" programske jezike.

2.1 Razredi in objekti

Ker jezik Zero uporablja razrede, se bomo osredotočili na razredne programske jezike. Razredni jeziki ne ustvarjajo objektov spontano, tj. neposredno, temveč preko šablonskih opisov, katere zgoščeno imenujemo *razredi*. Razred, v svoji najpreprostejši pragmatični definiciji, služi kot načrt oz. postopek za izdelavo objekta. Razlika med razredom in objektom v prevedenem programskem jeziku je ta, da razred obstaja v času prevajanja, objekt pa v času izvajanja; če omalovažimo možnost dinamičnega prevajanja in ustvarjanja razredov v času izvajanja. Razred, v kolikor ga jemljemo kot definicijo objektovega behavioralnega modela, je le pasivna entiteta, medtem ko lahko objekt smiselno okarakteriziramo s prvinami aktivnega agenta. Razred je preliminarna, deskriptivna in zgolj formalno podana narava objekta, ki še ne obstaja; je definicija na nivoju protokolarnega opisa objekta. Razredni programski jeziki za ustvarjanje objektov običajno uporabljajo posebne mehanizme, Java in C++ to realizirata z operatorjem *new*.

Koren besede klasifikacija je angleška beseda *class*, ki se v razrednih programskih jezikih najpogosteje uporablja za definicijo razreda. Tako je tudi ena izmed prvenstvenih lastnosti razredov prav klasifikacija lastnosti. Razred je klasifikacijski mehanizem, s katerim podvržemo celo družino objektov, izmed katerih imajo vsi identične značilnosti. Razred lahko opisuje povsem fiksne, nespremenljive lastnosti – specifične objekte, za katere pravimo, da so rezultat monomorfne preslikave iz domene razredov. Nasprotno pa lahko razred služi tudi polimorfni preslikavi, kadar določa zgornjo mejo znotraj hierarhične strukture družine objektov. Mnogoličnost razredne opisne šablone je prisotna v vseh jezikih, ki z dedovanjem omogočajo inkrementalne definicije lastnosti. Koncept dedovanja, ki je potreben, da nek jezik imenujemo objektno usmerjen, izkorišča načelo, da ima razredni opis razširljivo implementacijo in protokolno shemo.

Dedovanje na nivoju razrednih definicij je mehanizem, ki je značilen za razredne objektno usmerjene programske jezike. Dedovanje se poistoveti z grajenjem razrednih hierarhičnih struktur. V jezikih, ki vključujejo model čistega objektnega abstrahiranja, med katere spada tudi jezik Zero, obstaja vsaj en osnovni ali ekskluzivni razred, ki je temelj vseh drugih razredov. Jezik Zero imenuje ta razred kar *Object*. Pravila in strukturo razredne hierarhije določa razredna topologija. Iz nekega razreda izpeljan razred imenujemo *podrazred* ali otrok, razred, iz katerega je potekala izpeljava, pa starševski

razred ali *nadrazred*. Razred, ki se v razredni hierarhični strukturi nahaja višje, imenujemo predhodnik, tistega, ki se nahaja na nižjih nivojih pa naslednik. Starševski razred lahko potemtakem imenujemo neposredni predhodnik in otroka neposredni naslednik. V programskih jezikih, ki omogočajo večkratno dedovanje, lahko ima posamezen razred več kot enega starša, tj. več neposrednih predhodnikov, katerih karakteristike deduje. Izkušnje načrtovalcev programskih jezikov, inženirjev in programerjev kažejo, da razrede v razrednih jezikih vseskozi povezujemo s konceptom tipa. Če pogledamo Javo in C++, spoznamo, da lahko ime razreda stoji na vseh mestih, kjer pričakujemo tip. Toda razreda ni mogoče enoumno poistovetiti s tipom, saj gre za dva temeljno raznorod-na koncepta. Tip, sam po sebi, daje opisno sliko protokola, je mehanizem konformance in je sam po sebi "pasivni" pojem. Tudi razred je v primeri z objektom sicer pasiven, toda ima pomembno lastnost, ki je tipu neznana, implementacijo namreč. Medtem ko razred sprejemamo kot definicijsko lupino za implementacijo metod, je tip čista abstrakcija, ki predstavlja domeno vrednosti in nad njo definiranih operacij [19]. Tako razred kot tip lahko s pomočjo mehanizmov dedovanja dopolnjujemo in dograjujemo do višjih oblik, vendar s to razliko, da je pri razredih predmet sprememb implementacija, pri tipih pa abstrakcija oz. protokol.

Smiselno je, da signaturo razreda opredelimo podobno, kot velja za metode. Če razred določa t.i. objektni tip, potem signaturo razreda definira nabor signatur njegovih metod. Signatura razreda je izjemno pomembna informacija, posebno v statično tipiziranih programskih jezikih, saj jo uporabljamo pri določevanju konformance med posameznimi nivoji znotraj razredne hierarhije. Zagotavljanje kompatibilnosti na nivoju signature je temeljna ideja varnega tipiziranja, saj prav ta odtehta pomanjkljivosti statično tipiziranih jezikov v primerjavi z dinamično tipiziranimi. Signatura razreda, ki je določena s signaturami metod, se torej razširi na skupek signatur, od katerih je sleherna določena z imenom metode ter s tipi formalnih parametrov in tipom vračanja:

$$metoda : parameter_1 \times parameter_2 \times \ldots \times parameter_n \rightarrow T$$

V signaturo, vsaj v jeziku Zero, ne vključujemo tipa vračanja metode. V okviru statično tipiziranega programskega jezika nas zanimajo karakteristike in principi sistema tipov, ki ohranja konsistenco in omogoča preverbe v času prevajanja.

Kadar razpravljamo o razredih in objektih, se povečini ne zavedamo, da so tudi strukturalno in funkcijsko najpreprostejši objekti definirani rekurzivno. To izhaja iz dejstva, da objekt za invokacijo metode potrebuje mehanizem samoreferenciranja, saj se implementacija iste metode izvaja nad različnimi instancami [84]. V jeziku Zero je referenca na prejemnika podana z implicitno definirano metaspremenljivko *self*. Java in C++ uporabljata *this*. Z implementacijskega gledišča ni zanemarljivo, da lahko referenca prejemnika opisuje katerikoli razred v hierarhiji. Na objekt lahko torej gledamo kot na hierarhično drevesno strukturo ali kot na celovit objekt, strukturalno in funkcionalno opisan z razredno hierarhijo. Kot bo razloženo v nadaljevanju, jezik Zero, zaradi dinamične narave spreminjanja metod, uporablja dve tovrstni referenci, saj ena v splošnem ni dovolj. Samoreferenciranje je sicer znan koncept objektnih in objektno usmerjenih razrednih programskih jezikov.

2.1.1 Abstrakcija in koncept objektnega tipa

V idejnih temeljih objektno usmerjenega načrtovanja programskih sistemov nedvomno leži abstrakcija. Modeliranje z uporabo abstrahiranja funkcionalnosti je zaradi korenitih sprememb, ki jih je prineslo v razmišljanje, bržkone najpomembnejša zasnova, ki so jo uvedli objektno naravnani programski jeziki, razredni ali brezrazredni. Pojem abstrakcije prav gotovo ni nov, tudi na področju računalništva ne, vendar je njegova vpeljava v načrtovanje in razvoj kompleksnih programskih sistemov prinesla spremembe malodane v vseh pogledih. Prisotnost abstrakcije je v sodobnem načrtovanju programske opreme tako vsestransko pričujoča, da se je povečini sploh ne zavedamo. Ideja abstrahiranja omogoča deduktivno inkrementalno specializacijo tj. razmišljanje od splošnega h konkretnemu. V smislu hierarhije bi takšno metodiko načrtovanja lahko poimenovali tudi top-down. Kako se torej abstrahiranje kaže pri objektno usmerjenih programskih jezikih? Konceptualna zasnova objektno usmerjenih jezikov ne zgolj omogoča, temveč tudi spodbuja načrtovanje s pomočjo abstrahiranja. Abstrakcija je dodelan ali nedodelan šablonski načrt, ki služi kot temeljni napotek specifični implementaciji. Razumljivo in logično je, da sleherni kompleksnejši programski sistem potrebuje takšen načrt, po katerem bo tekla njegova realizacija. Prednost abstrakcije je vselej v njeni širini in univerzalnosti, s katerima lahko zaobjame celo paleto proble-

matike, ki v času načrtovanja še ni dodelana, bodisi zaradi pomanjkanja informacij o končnem delovanju ali nejasnosti dejanske implementacije.
Pojem abstrakcije hitro in enostavno vpeljemo v objektno usmerjene programske jezike, saj jo tukaj razumemo v povsem konkretnem tehničnem smislu. Pravimo, da z mehanizmom abstrahiranja podamo abstrakten opis behavioralnega modela, ki je značilen za sleherni objekt, ki realizira ta model. Za objekt, katerega obnašanje zadošča oz. ustreza podanemu behavioralnemu modelu, pravimo da abstrakcijo konkretizira s funkcionalnostjo. Kadar v okviru objektno usmerjenih jezikov govorimo o konkretizaciji, ponavadi mislimo na podano implementacijo metod, ki sestavljajo protokol objekta.

V različnih programskih jezikih je mehanizem abstrahiranja izveden zelo raznoliko. V razrednih programskih jezikih so abstraktni behavioralni modeli opisani s pomočjo raz-redov, ki imajo v ta namen navadno vpeljane posebne konstrukte za definicijo abstrakt-nih metod. Razred, ki opisuje abstrakcijo, imenujemo preprosto abstrakten razred. V jezikih C++ in Java velja, da je razred nepopolno definiran ali "odprt", v kolikor vsebuje vsaj eno abstraktno metodo, tj. metodo brez implementacije. Abstraktna metoda je popolno definirana zgolj s pripadajočo signaturo, kot je to prikazano v predhodnem podpoglavju. Jezik C++ abstraktne metode definira tako, da jim "priredi" vrednost 0, medtem ko Java vpelje rezervirano besedo `abstract`. Tako v C++ kot v Javi velja, da abstraktnega razreda ni mogoče instancirati oz. iz njega ustvariti objekta, kar je po svoje razumljivo, saj razred ni definiran popolno temveč samo predpisuje model obnašanja, katerega bo dodelal in konkretiziral nek podrazred. Pojavi se vprašanje, kako rokovati s "čistimi" abstraktnimi razredi, razredi, ki imajo vse metode abstrakt-ne. C++ za tovrstno idejo nima posebnega mehanizma in uporablja razrede. Java po drugi strani uporablja konstrukt, ki omogoča čisto objektno abstrakcijo na način, da je omogočen izključno opis protokola in ne tudi implementacija. Java se s tem približa konceptu tipa, ki je mehanizem konformance in nima prav nič opraviti s kakršnokoli implementacijo. Java na ta način, s konstruktom vmesnika (`interface`), izkristalizira razliko med razredom in tipom, ki sta ontološko dva različna pojma. Toda tudi Java omogoča, da so vse metode nekega razreda abstraktne, s čimer žal razvrednoti kanoničnost svojega konstrukta za opis abstrakcije. Javanski vmesnik je striktno tip

in natančno ter enoumno določa metode, ki se lahko nad objektom, kateri ta vmesnik realizira, izvedejo. Vmesnik tako imenujemo `objektni tip`, ki ni nič drugega kakor abstrakten opis objekta in vsebuje samo nabor signatur metod. Pomembna prvina objektnega tipa je ta, da ne predpisuje instančnih spremenljivk, kot konformance stanju objekta. To nadalje pomeni, da lahko objekti z različnimi stanji v času izvajanja ustrezajo istemu objektnemu tipu, v kolikor zadoščajo signaturi metod.

C++ nima dobro razdelanih jezikovnih konceptov modeliranja in abstrahiranja. Koncept razreda je uporabljen univerzalno, za definicijo abstraktnih in konkretnih tipov. Pravtako je uporabljen za implementacijo funckionalnosti skozi metode. Ker sta po naravi različna koncepta, razred in podatkovni tip, zlita v enega, dedovanje prevzame vlogo mehanizma ponovne uporabljivosti ter istočasno služi kot mehanizem za modeliranje hierarhije tipov. Izkušnje so skozi zgodovino objektno usmerjenega programiranja pokazale, da ni brezpredmetno imeti kanoniziran jezikovni konstrukt za modeliranje tipa, ki je ločen od tistega, s katerim opisujemo razrede. Kadar med razredom in tipom velja fizična ekvivalenca, so relacije med podtipi identične tistim med razredi. Praksa kaže, da to ni vselej tudi logično. Razred, na implementacijski ravni, pogosto služi kot temelj, iz katerega dedujemo nov razred izključno zaradi uporabe obstoječe funkcionalnosti, vendar pa ni nujno tudi smiselni nadtip novega razreda. Behavioralni model podrazreda je v posebnih primerih lahko celo nadtip svojega nadrazreda! Posledica eksploatacije koncepta razreda vnaša neželjene komplikacije tako v samem modeliranju kot kasnejši implementaciji abstraktnega modela. Razlog je kritično omejena izraznost programskega jezika, kar zadeva tipe in fleksibilnost dedovanja. Vedeti je treba, da je abstrakcija tipa abstrahiranje nad nekonkretnim, neobstoječim, tj. nad abstrakcijo samo. Iz tega razloga je smiselno za takšno abstrahiranje vpeljati poseben jezikovni konstrukt, ki bo temu namenjen. Striktna kanonizacija jezikovnih konstruktov terja, da je podatkovni tip, ki je semantično gledano množica vrednosti, ločen od konstrukta razreda, ki omogoča izključno konkretno implementacijo. Če se tega ne držimo, pride do anomalij pri semantiki tipiziranja, kot je to razvidno v jeziku C++. Kljub strogi ločitvi pa lahko parcialne abstrakcije, ki so pomešane s konkretno implementacijo, realiziramo s pomočjo razredov, kadar je to smiselno. S stališča učinkovitosti hitro opazimo še eno pomanjkljivost združevanja tipa in implementacije. V jeziku C++ se tako nemalokrat

primeri, da je razred, ki služi izključno za abstrakcijo in pri tem nima nobene funkcionalnosti, instanciran, kot vsi ostali v razredni hierarhiji. Tako se ustvarja objekt, ki bi lahko imel manjšo velikost, in kličejo se konstruktorske metode, ki sicer sploh ne bi bile potrebne. Vpliv koncepta tipa se tako pokaže v času izvajanja, namesto da bi z njim opravili v času prevajanja.
Javina jezikovna zasnova omogoča konstruiranje drevesnih hierahij tipov in medsebojnih relacij, ki niso nujno v logični povezanosti s hierarhijo razredov, v kateri se nahaja realizacija teh tipov. Na takšno razvojno paradigmo jezika Jave so vplivale predvsem očitne slabosti polimorfizma tipov v jeziku C++. Slednji je spričo svoje izjemne razširjenosti na področju tipov doživel številne nadgradnje in razširitve, sicer nestandardne, vendar kljub temu praktično uporabne. Sem spadajo t.i. signature tipov [14, 13], podobne konstruktom v programskem jeziku ML [67, 71] in razredom tipov v jeziku Haskell [21].

2.1.2 Abstrakcija enkapsulacije

Trend objektno usmerjenega razvoja programske opreme je svoje prednosti v največji meri izkazal v evoluciji načrtovanja, na katero večkrat vplivajo nepredvidljivi in iz tega razloga tudi nenačrtovani dejavniki okolja. Prav evolucija terja ideje, kot sta ponovna uporabljivost [16] in prilagodljivost ter razširljivost in odprtost za spremembe. Obnašanje objekta je celovito definirano z razrednim predpisom sporočil, katera je objekt sposoben prepoznati in nanje odgovoriti. Funkcionalnost sporočil je podana z implementacijo metod. Potankosti in specifičnosti implementacije posamezne metode so za zunanje entitete nepomembne. Iz tega razloga lahko implementacijske in strukturalne razsežnosti kapsuliramo v hermetizirano entiteto in jih s tem zakrijemo pred zunanjostjo. To imenujemo enkapsulacija. Koncept enkapsulacije najdemo malodane v vseh objektno usmerjenih programskih jezikih. Strikten princip podatkovnega kapsuliranja zahteva, da je celotno stanje objekta oz. instančnih spremenljivk razreda navzven nepristopno. Tak pristop uporablja tudi jezik Zero, medtem ko Java in C++ to sicer omogočata, vendar ob tem nudita tudi možnost neposrednega dostopa stanja. Striktna politika dostopa prinaša na površje prednosti enkapsulacije, ki so pomembne za vzdrževanje programske opreme skozi njeno evolucijo. Edina metoda pri tem pristopu,

s katero lahko vršimo aktivno manipulacijo objektovega stanja, je pošiljanje sporočil, katerega v sklopu razrednih objektno usmerjenih programskih jezikov razumemo kot invokacijo metod. Metode, s katerimi so realizirane operacije, so, preko razreda, integralni del objekta in imajo zaradi tega polni dostop do strukture objektovega stanja. Enkapsulacija ima na področju programskih jezikov pomembne teoretske in še pomembnejše pragmatične vidike. Razumemo ga kot enega izmed temeljnih konceptov abstrakcije, saj posledično omogoča minimizacijo povezanosti, in kar je še pomembnejše, odvisnosti med implementacijami posameznih entitet. Odvisnosti, ciklične ali enostranske, zelo otežujejo aplikacijo sprememb in novih funckionalnosti, kadar niso realizirane s striktnim pristopom dostopanja do objektovega stanja. Iz takšne miselnosti izhaja tudi programski jezik Zero, ki s striktno politiko manipulacije stanja ne zgolj omogoča, temveč celo obvezuje programerja, da uporablja pristop, s katerim ne more vnašati odvisnosti na nivoju stanja objekta. Programski jezik Java omogoča omenjeno preko vmesnikov, ki služijo kot ločnica med abstrakcijo tipa in strukturalno ter implementacijsko naravo objektov.

2.2 Dedovanje in virtualni mehanizmi

Dedovanju pripisujemo pomembno vlogo in ga zaradi njegovih izjemnih zmožnosti povsem praktične aplikacije štejemo za enega izmed ključnih mehanizmov objektno usmerjenih in objektnih programskih jezikov. Ideja dedovanja, ki nudi odprtost razširitvam [1] in možnost sprememb obstoječih funkcionalnosti, omogoča, vsaj teoretično, neskončno mero faktorizacije. Idealistično razmišljanje nas pripelje v prepričanje, da bi evolucija programske opreme, načrtovane po objektno usmerjenem principu, lahko bila zgolj inkrementalno spreminjanje, dograjevanje in izpopolnjevanje. Vsi omenjeni mehanizmi so temeljni konceptu dedovanja. Organizacija jezikovnih mehanizmov večine programskih jezikov nas napeljuje k spoznanju, da je dedovanje praktični mehanizem za specializacijo do določene mere nedodelane ali povsem abstraktne funkcionalnosti, torej vzvod, s pomočjo katerega abstrakcijo reificiramo in ji damo konkretno sliko. Pravimo, da s tem abstrakcijo usnovimo.

Koncept specializacije je v svojem bistvu nazoren in nezahteven, vendar posamezni ko-

raki niso enostavni, posebno kadar gre za velike programske sisteme, kjer je potrebno preudarno usklajevati številne in velikokrat tudi težko združljive opcije. Specializacija ni samo ohranjanje kompatibilnosti in odprtosti signatur, dotika se tudi behavioralnega modela, ki leži globlje. Obnašanje objekta, ki je določeno prav z behavioralnim modelom, je težko ali celo nemogoče obvladovati, v kolikor se opiramo zgolj na specializacijo na jezikovni ravni. Programski jeziki specializacijo ponavadi tretirajo s pomočjo konformance tipov, kaj več bi zahtevalo inteligentne metode, s katerimi bi lahko preverili, ali se objekti resnično obnašajo po specifikacijah. Specializacija obnašanja temelji na specializaciji tipov, katera z globino omejuje splošnost in vpeljuje vedno striktnejše omejitve funkcionalnosti. Razumljivo je torej, da jezikovni mehanizmi omogočajo samo podlago, katera služi in pomaga programerski disciplini. Le-ta je edini zares zanesljivi aspekt, s katerim je moč zagotoviti specializacijo obnašanja.

Če so tipi princip organizacije nekih omejitev, je dedovanje proces postavljanja relacij med temi omejitvami. Relacije ne pomenijo samo vzpostavljanja povezav med tipi, temveč omogočajo tudi njihovo faktoriziranje po abstraktnostnih stopnjah. Zgodnja miselnost dedovanja je ohranjala prepričanje, da gre za strukturalno sredstvo, s katerim lahko izvajamo modeliranje entitet realnega sveta tako, da jih po logični pripadnosti povezujemo v hierarhije. Toda že hitro so se pokazale prave prvine in nameni dedovanja, ki je z novimi spoznanji preraslo v sredstvo za t.i. konceptualno specializacijo. V takšni luči se uporablja tudi danes, ko si sodobnega načrtovanja ter implementacije programskih sistemov brez uporabe konceptov dedovanja skorajda ne moremo več zamisliti.

Ko uvidimo, da je dedovanje temelj tako abstrakciji kot konkretizaciji, specializaciji z inkrementalnimi spremembami ter ponovni uporabljivosti programskega koda, postane jasno, zakaj je prav ta mehanizem primarne narave v objektno usmerjenih programskih jezikih – razrednih in prototipnih. Glede na vse naštete prednosti je po drugi strani potrebno spoznati tudi slabosti, ki jih dedovanje vnaša v programski jezik. Komplikacije se, kot je to sicer običajno z vsakim jezikovnih konstruktom, najprej pokažejo v semantičnem modelu programskega jezika. Med jezike, kjer je mehanizem dedovanja precej zakompliciral semantiko, spada tudi C++. To gre pripisati več dejstvom. Jezik C++ omogoča večkratno dedovanje, kar znatno zaplete razreševanje konfliktov, hkrati

pa jezik uporablja tudi precej zapletena pravila dedovanja in redefiniranja. Tretji problem, ki se pojavi pri C++, pa ne zadeva samega mehanizma dedovanja, temveč je v tem, da je koncept dedovanja uporabljen za realizacijo različnih, nezdružljivih idej.
Omejimo se na dedovanje v razrednih objektno usmerjenih programskih jezikih. Če se držimo načel specializacije, razredi, ki dedujejo iz starševskih razredov, ustvarijo novo hierarhijo, ki ima bolj specifično funkcionalnost kot hiearhija do starševskega razreda. Podrazred ima moč konkretizirati in s tem zapečatiti abstraktno funkcionalnost starševskega razreda. Metode starševskega razreda določajo vmesnik, kateremu morajo zadoščati vsi podrazredi, v kolikor želimo ohranjati kompatibilnost. Skladnost protokolov posameznih razredov se realizira na nivoju signatur metod. Podrazredi tako nimajo popolne svobode v specializaciji, temveč se morajo držati pravil, ki so bila določena v protokolu starševskega razreda. Omejevanje funkcionalnosti vpliva na celotno razredno hiearhijo.
V čistem objektnem modelu se dostop do stanja vrši izključno preko invokacije metod, vendar ima, zaradi razlogov učinkovitosti, večina razrednih programskih jezikov dodatne mehanizme dostopa. Omejevanje dostopa do objektovega stanja je zunaj razreda navadno veliko striktejša, kot v podrazredih. To je razumljivo, saj podrazredi v času izvajanja predstavljajo integralni del istega objekta. Java in C++ omogočata neposreden dostop do instančnih spremenljivk starševskega razreda z modifikatorjem dostopa `protected`. Neposreden dostop do starševskega stanja je dvorezen meč, saj se po eni strani izogne časovno zahtevni invokaciji metode, po drugi strani pa anulira neodvisnost protokolov in s tem "zaveže" dva razreda skupaj. S tem je onemogočeno kontinuirano vzdrževanje skladnosti med starševskimi in izpeljanimi razredi. Načrtovanje programske opreme po takšnih principih zahteva pristno sodelovanje izvajalcev. Na žalost to ni vedno mogoče, saj imamo velikokrat na voljo izključno preveden binarni kod knjižnic. Jezik Zero uporablja, kar zadeva fleksibilnost in ne učinkovitost, boljši pristop, s katerim ne omogoča neposrednega dostopa do stanja. Čisti "metodni" pristop ohranja strukturalno neodvisnost med posameznimi razredi v hierarhiji, saj ne omogoča direktnega navezovanja na instančne spremenljivke v drugih razredih. S tem se ohranja možnost dodajanja, odstranjevanja in spreminjanja instančnih spremenljivk starševskega razreda brez vpliva na skladnost z izpeljanimi razredi. Jezik Z_0, katerega

v tem delu nadgrajujemo, se zapletljajem z instančnimi spremenljivkami, kar zadeva medrazredno konsistenco, povsem izogne, saj v duhu čistega metodnega pristopa uporablja izključno metode in eksplicitnih instančnih spremenljivk sploh ne predpostavlja. Zdi se, da čisti metodni pristop rešuje vse težave medrazredne kompatibilnosti pri dostopu do objektovega stanja. Čeprav je ta pristop primernejši, saj ohranja vsaj strukturalno neodvisnost, je treba vedeti, da se lahko podrazredi vežejo na metode celotne hierarhije, s čimer prav tako ustvarijo odvisnosti med razredi. To se pojavlja na vseh nivojih razredne hierarhije in lahko v primeru slabe programerske prakse pripelje do resnih popačenj funkcionalnosti. Odvisnosti med razredi predstavljajo problem v vseh objektno usmerjenih programskih jezikih, statično in dinamično tipiziranih. Koncept dedovanja kot sredstva za izvedbo specializacije po splošnem prepričanju razumemo kot omejevanje in konkretizacijo funkcionalnosti z dodajanjem specifičnih, konkretnih lastnosti. Toda povsem sprejemljivo in na mestu je razmišljanje, da bi mogli posamezne lastnosti, predvidene nemara zaradi obsežnosti problema, na višjem abstraktnostnem nivoju, odstraniti, v primeru, da se je izkazalo, da niso več potrebne. Odstranitev lastnosti, ki niso več aktualne in smiselne na konkretnem nivoju, bi zahtevala dinamičen mehanizem dedovanja, s katerim bi lahko, v razrednih objektno usmerjenih jezikih, izločali metode nadrazredov. Statičen sistem tipov otežuje takšno različico dedovanja, saj ne moremo dati zagotovila, da se kateri izmed predhodnih razredov znotraj hierarhije ne navezuje na katero izmed metod, ki bi jo želeli odstraniti. Skladnost po odstranitvi metode lahko zagotovimo samo v razredu, kjer odstranitev izvedemo. Narava dinamično tipiziranih in interpretiranih jezikov, kot je CommonObjects [88] (razširitev CommonLisp [92]), je bolj naklonjena takšnim mehanizmom. V jezikih, ki ne podpirajo eksplicitnega izločanja že definiranih metod, bi to lahko dosegli z zapečatenjem metode tako, da bi ji spremenili pravice dostopa. Kljub temu takšna strategija ne doseže prave ideje odstranitve, pri kateri se metoda odstrani iz celotne hierarhije.

Dedovanje je v večini objektno usmerjenih programskih jezikov tesno povezano s tipiziranjem. V jezikih, ki ne ločujejo med čisto abstrakcijo tipa in konceptom razreda (npr. C++), je hierarhija razredov sočasno tudi hierarhija tipov. Dedovanje je v tem primeru realizacijski mehanizem podtipiziranja, s katerim določamo pravila specializacije tipov in relacije med tipi v hierarhiji. Tipiziranje ima v statično tipiziranih

programskih jezikih velik pomen, saj prav na tem konceptu temelji učinkovitost teh jezikov, ki konformanco med tipi preverjajo že v času prevajanja. Jeziki, ki združujejo tipiziranje in razrede, dodatno povzročajo vnašanje soodvisnosti, saj se s spremembo razredov lahko ruši tudi konsistenca v hierarhiji tipov. Dedovanje je univerzalističen princip, uporaben v specializacijske namene tipov in konkretne funkcionalnosti, vendar je potrebna previdnost, da je dedovanje tipov osnovano na specializaciji obnašanja, sicer lahko pride do kontradiktornih relacij v hierarhiji tipov in pripadajoči hierarhiji razredov.

2.2.1 Klasifikacija dedovanja

Dedovanje kot karakteristično lastnost objektno usmerjenih programskih jezikov lahko klasificiramo po različnih kriterijih, bodisi da na koncept gledamo kot na strukturalni ali kot na modelirni mehanizem. Opredelitev dedovanja sega v same začetke njegove praktične uporabe, saj so se že takrat pojavile ideje o raznolikosti in širokonamenskosti tega koncepta. Omenili smo prototipne in razredne objektno usmerjene jezike. Pri obojih je koncept dedovanja temeljni mehanizem inkrementalnih sprememb in osnova specializacije. Tako lahko že na ravni programskega jezika dedovanje razdelimo na razredno in prototipno. Razredni programski jeziki dedovanje uporabljajo za izgraditev razredne hierarhije, ki na instančni ravni, brez uporabe meta arhitekture, ni vidna. Brez možnosti introspekcije se instancirani razredi (objekti) ne "zavedajo" svoje hie-rarhične strukture, bodisi da je ta drevesna bodisi, da je linearna. Objekt, ki je konkreten primerek razredne abstrakcije, je homogena entiteta, ki jo opisuje enovit behavioralni model in ki operira z enovito funkcionalnostjo. V prototipnih programskih jezikih, kjer ni razredov in s tem tudi ne kategorizacije po razredih, mora biti dedovanje realizirano na drugačen način. Novi objekti, ki bi jih iz obstoječih želeli ustvariti s kloniranjem, morajo imeti neko možnost spreminjanja. Sprememba v prototipnih programskih jezikih je bolj individualne narave kot v razrednih jezikih, saj ima spremembna na ravni objekta vpliv samo na objekt, medtem ko se sprememba razreda uveljavlja v vseh primerkih tega razreda. Prototipni jeziki z dedovanjem omogočajo inkrementalne (ali dekrementalne) modifikacije posameznih eksistenc – prototipov, razredni pa spremembe celih družin.

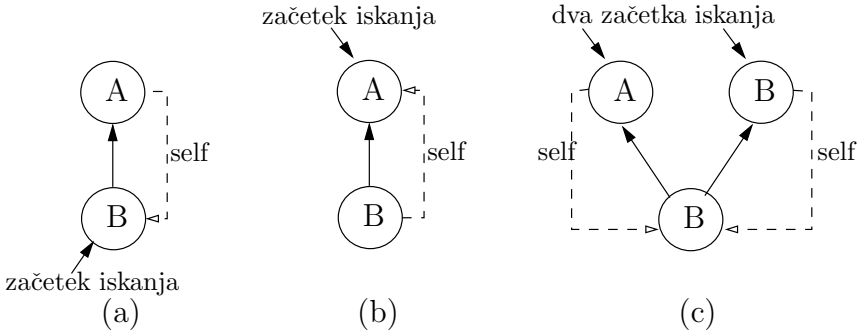
Drug kriterij dedovanja je implementacijska shema, ki je lahko izvedena z delegacijo ali s kopiranjem. Praktične izkušnje v implementaciji so pokazale, da je delegacija primernejša za dinamične in prototipne programske jezike, kopiranje pa se je uveljavilo v razrednih in statično tipiziranih jezikih. Delegacija je tesno povezana s konceptom sporočil. Kadarkoli objekt prejme sporočilo za izvršitev neke metode, na katerega ne zna odgovoriti, to sporočilo delegira svojim staršem. Takšna obravnava se ponavlja vse dokler sporočilo ne prispe do objekta, ki zna ustrezno odgovoriti z izvedbo metode ali dokler se pot ne konča v korenskem objektu. Če noben objekt ne zna odgovoriti, je bilo očitno poslano napačno sporočilo. Pri dedovanju s kopiranjem se vsi atributi nadrazreda prekopirajo v izpeljan razred, kateremu se dodajo nove metode. Dedovanje s kopiranjem je konceptualno preprostejše od delegiranja, vendar tudi manj fleksibilno. Ker nov razred sestavljajo stare metode, katerim dodajamo nov nabor, se takšna implementacija dedovanja imenuje tudi dedovanje s konkatenacijo. Konkatenacija je uporabljena v jezikih C++ in Beta, medtem ko prototipni programski jezik Self [102, 27] implementira dedovanje z delegacijo. Seveda ne gre pozabiti, da obstajajo ideje in rešitve za implementacijo delegacijskih mehanizmov tudi v razrednih in statično tipiziranih programskih jezikih [103, 46, 61]. Delegacija je očitno dovolj učinkovit in primeren mehanizem, da si je utrl pot tudi v idejno zasnovo teh jezikov.

Druga lastnost, po kateri lahko razvrstimo dedovanje, je začetna točka iskanja metode. Da bi metodo lahko izvedli, jo je najprej potrebno poiskati. Iskanje primerne metode je lastno razrednim in prototipnim objektno usmerjenim programskim jezikom. V prevedenih programskih jezikih se metoda najde v času prevajanja, pri interpretiranih, kamor po navadi spadajo prototipni jeziki, pa se iskanje vrši dinamično ob samem izvajanju. Razumljivo je, da je takšnen pristop prilagodljivejši, a tudi časovno zahtevnejši. Metodo lahko torej pričnemo iskati v razredu objekta, nad katerim izvajamo invokacijo. Iskanje je uspešno, če najdemo metodo, katere signatura se ujema z metodo podano v sporočilu. V primeru, da ustrezna metoda ni bila najdena, se iskanje po enakem postopku nadaljuje v nadrazredu. Takšna shema iskanja metode je implementirana v jezikih C++, Java ter Smalltalk. Ker z iskanjem metode pričnemo v najbolj izpeljanem razredu, pravimo dedovanju, ki uporablja takšen princip iskanja, naslednjško dedovanje [98]. Alternativna iskalna shema prične z iskanjem metode v

najvišjem razredu hierarhije. Ker je ta razred starš vsakemu drugemu razredu, tako dedovanje imenujemo starševsko dedovanje. Takšen princip dedovanja se zdi nenavaden, saj je posamezna metoda določena v razredu, kjer je prvič definirana. Kako bi torej uveljavili idejo inkrementalne specializacije? Starševsko dedovanje ima pomembno lastnost, kadar je izvedeno simetrično. Simetrično ali sestavljeno dedovanje pomeni, da se ne izvede samo prva najdena metoda, ampak vse, ki ustrezajo podani signaturi. Takšno dedovanje uporablja programski jezik Beta. Z izvedbo vseh ustreznih metod je ohranjen behavioralni model objekta, saj se funkcionalnost posamezne metode aplicira sistematično inkrementalno. Pri starševski shemi dedovanja se najprej invocira najbolj abstraktna metoda in nato vsaka konkretnejša. To ima za posledico, da podrazredi ne morejo nikoli v celoti redefinirati ali spremeniti funkcionalnosti, ki je bila predvidena na abstraktnem nivoju. Takšen model dedovanja je primeren za inkrementalno specializacijo funkcionalnosti in ima svoje mesto predvsem v akademskih jezikih. Funkcionalnost modela je včasih lažje izraziti hierarhično inkrementalno, tako da vsak nivo hierarhije funkcionalnost konkretizira z dodatnim, lastnim znanjem in informacijami. Starševsko dedovanje je zaradi svoje narave iskanja metode bolj primerno za jezike, ki omogočajo izpeljavo samo iz enega starša. Večkratno dedovanje takšno shemo zakomplicira, saj začetek iskanja v primeru dveh ali več staršev ni več enoumno določeno, ampak bi zahteval razrešitev s strani programerja. Obe shemi, starševsko in nasledniško, prikazuje slika 1.

Da bi v programskih jezikih, ki omogočajo samo asimetrično dedovanje, dosegli podoben učinek, bi morali eksplicitno invocirati ustrezne metode v celotni hierarhiji. S stališča programiranja je to težko zagotoviti, saj nimamo ustreznega mehanizma, ki bi to zahteval. V Javi in C++ lahko to dosežemo z eksplicitnim modifikatorjem dostopa do metode, s katerim metodo tudi statično povežemo.

Klasično dedovanje statično tipiziranih programskih jezikov predvideva, da se razredna hierarhija v času izvajanja ne spreminja. To pomeni, da v hierarhično razredno strukturo v času izvajanja ne moremo vstaviti novega razreda ali izvzeti obstoječega. Navadno obstoječega razreda znotraj hierarhije ne moremo niti spremeniti, kljub temu da bi le-ta ohranjal skladnost signatur. Fiksacija razredne strukture ima drastičen vpliv na učinkovitost statičnega sistema tipov, saj se v čas izvajanja prenesejo le redke pre-



Slika 1: Iskanje metode se pri nasledniškem dedovanju (a) prične v najbolj izpeljanem razredu hierarhije, pri starševskem (b) pa v osnovnem (najvišjem) razredu. Konfliktna točka pričetka iskanja metode pri večkratnem starševskem dedovanju (c).

verbe tipov. Toda takšna shema dedovanja, čeprav uporabljena skoraj v vseh statično tipiziranih jezikih, je skrajno neprilagodljiva. Če bi lahko razredno strukturo v času izvajanja spremenili, če bi lahko dodali ali odstranili kak razred na poljubnem mestu, bi se morali poslužiti dinamičnega dedovanja, katero pride, zaradi nezmožnosti preverjanja napak v času prevajanja, v poštev pri dinamično tipiziranih in običajno prototipnih jezikih. Splošno privzeti mehanizem dinamičnega dedovanja je delegacija, ki je značilna lastnost prototipnih jezikov. Pri delegaciji so reference na starševske razrede običajno shranjene v posebnih režah, katerih vrednosti se lahko spreminjajo. Če lahko izpeljan razred spremeni vrednost v takšni reži, se spremeni celoten nadrazred. Navkljub očitnim prednostim takšnega ravnanja hitro sprevidimo tudi potencialne nevarnosti. Sprememba nadrazreda lahko namreč, če se spremenijo signature metod, poruši funkcionalno in strukturalno kompatibilnost celotne razredne hieararhije. Hitro se poraja vprašanje, ali obstaja kakšna realna možnost, da bi vsaj osnovno idejo dinamičnega dedovanja prenesli pod okrilje statičnega sistema tipov. Izkaže se, da je to mogoče pod premiso, da se ohranja razredna struktura in signature metod. Razred v hierarhiji bi lahko nadomestili z novim razredom v primeru, ko ima ta razred identično strukturo v smislu signatur metod, medtem ko je implementacija teh metod lahko povsem drugačna. S takšno omejitvijo lahko zagotovimo ohranitev varnosti statičnega sistema tipov kot tudi samo invokacijo metod.

2.2 Dedovanje in virtualni mehanizmi

Naslednje vprašanje, ki se poraja, kadar govorimo o dedovanju je, katere lastnosti razreda se sploh dedujejo in ali obstajajo sploh kakšne možnosti selekcije. Java in C++ se poslužujeta pravila, pri katerem se dedujejo vse lastnosti, ki nimajo privatnega dostopa. Ker to ni vselej prikladno in na mestu, nekateri programski jeziki, npr. Kevo [95], omogočajo selektivno dedovanje, pri katerem lastnosti, katere želimo dedovati, eksplicitno navedemo. Selekcija lastnosti, ki se bodo dedovale, pomeni višjo stopnjo fleksibilnosti pri modeliranju razredne hierarhije. Hkrati pa takšna shema povzroča težave v sistemu tipov. Možnost izključitve lastnosti, definiranih v nadrazredu, lahko povzroči, da podrazred več ne zadošča vmesniku, kateri je veljal za nadrazred. Takšno tipiziranje ni statično varno in je zato primerno za dinamične programske jezike.

Vrsta dedovanja, ki ima veliko praktično vrednost in je na žalost zelo redko implementirana v programskih jezikih, je t.i. mešano dedovanje (mixin inheritance) [47]. Velika vloga mešanega dedovanja je v njegovi univerzalnosti kar zadeva funkcionalnost in uporabo. Ne samo, da omogoča dedovanje lastnosti v klasičnem smislu, temveč tudi visoko stopnjo ponovne uporabljivosti obstoječega programskega koda [87, 85]. Zraven praktične veljave ima to dedovanje tudi kvalitetno izpeljana teoretska dognanja [9, 8, 7]. Zaradi vseh naštetih lastnosti je mešano dedovanje postalo splošno priznano in je doživelo svojo implementacijo tudi v jezikih, v katerih sprva ni bilo načrtovano [86]. Funkcionalno zasnovo mešanega dedovanja se da razložiti na preprost način, saj se temeljna ideja opira na dobro uveljavljen pojem inkrementalne modifikacije. Programski kod, ki realizira inkrementalni del, se ne nahaja v izpeljanem razredu, temveč je predstavljen samostojno, tj. z lastnim konstruktom. Mešano dedovanje v razrednem programskem jeziku bi inkrementalni del predstavilo z razrednim konstruktom, s katerim se nato izdela specializiran razred. Razred, ki vsebuje inkrementalni del, imenujemo inkrementalni ali vključitveni razred in ni sintaktično in strukturalno nič drugačen od “normalnega” razreda. Razlika je le v semantiki, saj iz takšnega razreda ne moremo ustvariti objekta, ker razred sam po sebi ne definira zaključene funkcionalnosti. Razred, v privzeti terminologiji objektne usmerjenosti, tudi ni abstrakten, saj definira neko parcialno funkcionalnost z namenom dopolniti drug, samo potencialno izdelan razred. Vključitveni razred je predstavljen individualno in nima starševskega

razreda in je kot takšen neodvisen od hierarhije, v katero je vključen kot inkrementalni del. To pomeni, da ni logično vezan na nobeno specifično mesto v razredni hierarhiji in je iz tega razloga lahko uporabljen na katerikoli poziciji znotraj hierarhije, kjer je njegova funkcionalnost potrebna. Implementacija inkrementalne funkcionalnosti v ločenem razredu je lahko izvedena zelo fino, kar pomeni, da se definirajo samo lastnosti in metode, ki so resnično potrebne. Kontrastna pozicija se pojavi v klasičnem dedovanju, kjer aposteriorno znanje pokaže, da se velikokrat dedujejo lastnosti, katere so nepotrebne, vendar so "v paketu". Mešano dedovanje se tej problematiki izogne. Kot je navada pri klasičnem dedovanju, je tudi mešano mogoče izvesti enkratno ali večkratno, pri čemer gre v slednjem primeru za sestavljanje ločenih funkcionalnosti.
Poleg navedenih prednosti, ki se najvidneje manifestirajo v ponovni uporabljivosti programskega koda, ima mešano dedovanje tudi slabo stran. Ta se izkaže v nekanoničnosti namena uporabe, saj se v razrednem jeziku uporablja razred tako za implementacijo inkrementalnih delov kot klasičnih razredov. Gre torej za dva raznorodna koncepta, ki sta realizirana z istim jezikovnim konstruktom. To pelje v zmanjšano razumljivost hierarhičnega razrednega modela. Koncept mešanega dedovanja je v nekaterih programskih jezikih razširjen na cele module, ki jih imenujemo mešani moduli (mixin modules) [42, 53].

2.2.2 Večkratno dedovanje

Večkratno dedovanje je ena izmed obstoječih shem dedovanj objektno usmerjenih programskih jezikov. Narava takšnega dedovanja ni v svojih partikularnih konceptih na noben način vezana izključno na razredne ali samo na prototipne programske jezike. Večkratno dedovanje najdemo v obeh vejah jezikov. Kapaciteta večkratnega dedovanja pomembno vpliva na razvojni cikel konceptualnega modeliranja realnih problemov v lupini objektno usmerjenih nazorov. Kadar je možno izdelati model z uporabo kombinacije obstoječih entitet, je večkratno dedovanje pravšnji koncept za ta namen. Ne samo, da poenostavlja strukturo samega modela, ampak izboljšuje tudi stopnjo ponovne uporabljivosti programskega koda. Navkljub pomemebnim pozitivnim lastnostim, večkratno dedovanje ni dobilo veliko priložnosti v obstoječih programskih jezikih. Temeljne vzroke gre iskati v zapletenem semantičnem modelu jezika [24, 6, 32] in imple-

mentacijskih zahtevah, ki so znatno neugodnejše, kakor pri enkratnem dedovanju. Ker glavnina programskih jezikov ne podpira večkratnega dedovanja, so bile, skozi raziskovalne študije, predlagane različne alternativne rešitve tej shemi dedovanja [103, 15]. Kljub prednostim in slabostim, so mnenja o upravičenosti večkratnega dedovanja še vedno deljena. Strokovnjaki s področja objektno usmerjenih programskih jezikov so si enolični v tem, da večkratno dedovanje zahteva časovno neugodno analizo v času prevajanja, pri interpretiranih jezikih pa tudi v času izvajanja. Problematika v času prevajanja je semantične narave in je vezana na razreševanje dvoumnosti metod in atributov pri horizontalnem prekrivanju. Enkratno dedovanje se signaturnim dvoumnostim ogne tako, da se vselej izbere metoda ali atribut, ki se najde prvi. Vertikalno prekrivanje pri enkratnem dedovanju tako ne predstavlja posebne težave, saj se resolucija izvede nedvoumno, ne glede na to ali gre za starševsko ali nasledniško shemo dedovanja. Jezik C++ zahteva, da se horizontalno prekrivanje razreši s strani programerja z eksplicitno navedbo razreda, ki metodo definira.

Diagramsko shemo večkratnega dedovanja nazorno predstavimo z usmerjenim acikličnim grafom (directed acyclic graph), v katerem so posamezni razredi predstavljeni kot vozlišča, relacije med njimi pa z usmerjenimi povezavami. Jeziki, ki podpirajo večkratno dedovanje, rešujejo konflikte v dostopu do istoimenskih atributov na dva načina. Prvi način izpeljuje semantiko neposredno iz grafa dedovanja, pri čemer graf ostane nespremenjen. Metode in lastnosti se po grafu dedujejo tako dolgo, dokler jih v podrazredu ne redefiniramo. Kadar razred deduje iste lastnosti ali metode z isto signaturo iz več staršev hkrati, je potrebno konflikt razrešiti. Programski jezik Eiffel tovrstne konflikte rešuje tako, da konfliktne metode in lastnosti preimenuje. Smalltalk v ta namen ustvari operacijo, ki ob invokaciji signalizira napako. V večini drugih programskih jezikov takšna situacija pripelje do napake pri prevajanju. Drug pristop k razreševanju konfliktnih situacij je vezan na spreminjanje grafa dedovanja. Graf dedovanja najprej lineariziramo v seznam, kjer je vsak razred predstavljen kot vozlišče seznama. Linearizirán graf lahko obhodimo po enakem postopku kot enosmerno ali dvosmerno povezan seznam. Linearizirán graf mora ohraniti enako relacijsko strukturo, kot jo je imel prvotni aciklični graf. Težava nastane, kadar nek razred preslikamo v nadrazred razreda, ki je pred linearizacijo imel drug nadrazred. To pripelje do ano-

malije v hierarhični strukturi, saj se relacije starš – otrok ne morejo vselej ohraniti. Razreševanje konfliktov je pri uporabi lineariziranih grafov lažje, saj konfliktov, zaradi anulacije horizontalnega prekrivanja, preprosto več ni. Kljub temu je pristop, ki uporablja graf dedovanja v njegovi nespremenjeni obliki, bolj eksakten, saj ne dopušča, da bi po poti avtomatizacije prišlo do nenačrtovanih in nelogičnih invokacij metod.

2.2.3 Razpošiljanje

Preden se ustrezna metoda lahko izvede, jo je potrebno izbrati. Statično tipizirani prevedeni objektno usmerjeni jeziki selekcijo metode izvedejo na podlagi prejemnika sporočila. Takšno metodo imenujemo enkratno razpošiljanje (single dispatch). Nekateri programski jeziki, predvsem dinamično tipizirani, selekcijo metode izvedejo na podlagi tipa prejemnika, sporočila in podanih parametrov. Takšen pristop imenujemo večkratno razpošiljanje (multiple dispatch). Invokacija metode pri večkratnem razpošiljanju je bolj striktna, saj je metoda definirana za natančno določene argumente. Ker lahko metode po takšnem principu istočasno pripadajo različnim razredom, se imenujejo multimetode. Kljub temu, da večkratno razpošiljanje v določenem pogledu krepi izrazno moč, takšni jeziki niso posebej priljubljeni, saj kršijo nekatere temeljne prvine objektne usmerjenosti. Ker se multimetode razpošiljajo nad različnimi argumenti, niso vezane na posamezen abstraktni podatkovni tip, s čimer kršijo pravila enkapsulacije. Večkratno razpošiljanje je bolj kot objektno usmerjenim jezikom bliže proceduralnim in funkcijskim. Najbolj znan jezik, ki se poslužuje ideje multimetod je CLOS. Čeprav večina jezikov ne podpira multimetodnega pristopa, se predvsem v zadnjem času pojavljajo ogrodja, ki to kljub omejitvam originalnega jezika omogočajo. Implementacija takšnega ogrodja za Javo je opisana v [79].

2.2.4 Virtualni mehanizmi

Abstrahiranje v opisu in funkcionalnosti velja za najmočnejši modelirni mehanizem objektne usmerjenosti. Velikokrat podrobnosti v času podajanja abstraktnega opisa niso znane. Te se dodelajo kasneje, večkrat v povsem drugih razvojnih okoljih in z drugačnimi nameni. Obstoječa funkcionalnost se ne veže na konkretne izvedbe, temveč

na abstrakcijo. V razrednih jezikih je potrebno ustrezno zagotoviti konsistenco med razredi, ki se zanašajo na signaturo abstraktnih razredov in njihovimi dolgoročnimi konkretizacijami. To je še posebej pomembno v prevedenih jezikih, kjer abstraktni razredi niso nujno na vpogled – zanašati se gre zgolj na njihov zunanji vmesnik. Tehnika, ki preko abstraktnih metod omogoča transparentno invokacijo njihovih konkretiziranih različic, se imenuje pozno povezovanje (late binding)[37]. Za jezik, ki omogoča pozno povezovanje, pravimo, da vsebuje virtualne mehanizme. Pozno povezovanje pomeni, da natančni naslovi metod niso določeni v času prevajanja, temveč dobijo svoje prave vrednosti šele ob instanciranju konkretnega razreda. Ker abstraktne metode, vsaj v okviru razrednih jezikov, svoj smisel dobijo v konkretnih definicijah v podrazredu, je virtualni mehanizem tesno povezan z dedovanjem, saj brez njega pravzaprav ne bi obstajal. Zaradi svoje nepogrešljivosti je virtualni mehanizem postal eden izmed temeljnih konceptov objektno usmerjenih jezikov.

Virtualno invokacijo lažje razumemo v povezavi s polimorfnimi funkcijami. Polimorfne ali mnogolične funkcije oz. metode so tiste, katerih obnašanje ni vedno enako. Polimorfne funkcije imajo enako ime, njihova funkcionalnost pa je odvisna od konteksta, v katerem se izvajajo. Nazoren primer polimorfizma je funkcija za seštevanje dveh entitet. Takšna funkcija tipično operira nad argumenti celih, realnih in celo znakovnih tipov. Kadar govorimo o virtualnih mehanizmih, ki operirajo s polimorfnimi funkcijami, mislimo na vključitveni polimorfizem, ki pride do izraza predvsem pri dedovanju. Koncept parametričnega polimorfizma je do neke mere podoben vključitvenemu, vendar se njegova uporaba pokaže drugje. Čeprav bi si želeli statično tipiziran jezik, virtualni mehanizem zaradi svojih temeljev v dedovanju zahteva tudi dinamično preverjanje tipov. Virtualni klici metod se lahko izvajajo z enkratnim ali večkratnim dedovanjem, pri čemer slednje povzroča precej težav, saj struktura objektov v tem primeru ni več trivialna.

Resolucijo virtualnega klica metode v objektno usmerjenih jezikih imenujemo torej razpošiljanje sporočila. Z ozirom na nalogo bi ta postopek bolj dosledno poimenovali "odpošiljanje". Odpošiljanje predstavlja funkcijo, ki prejme ime sporočila in prejemnika. Ime sporočila in prejemnik predstavljata selektor, na podlagi katerega se pokliče metoda, ki ustreza temu paru. Za iskanje ustrezne metode se najpogosteje uporabljajo

tabele odpošiljanja (dispatch table). Te so lahko generirane že v času prevajanja ali pa med samim izvajanjem. Slednje je primerno za dinamično tipizirane jezike. Dinamični prototipni jeziki za iskanje metode uporabljajo enake pristope, kot so tisti za resolucijo metod pri dedovanju. Seveda je dejansko iskanje metode v času izvajanja precej počasnejše kot direktna invokacija preko vnaprej generiranih tabel. Nekateri jeziki zato uporabljajo različne tehnike predpomnjenja, ki iskanje metod precej pohitrijo [105].
Statične tehnike podatkovne strukture za iskanje metode izračunajo vnaprej v času prevajanja ali povezovanja. S tem se čas, potreben za resolucijo metode v času izvajanja, krepko zmanjša. Ker naslovi metod v času prevajanja niso znani, se metode referencirajo z indeksi. Dejanski naslov posamezne metode se določi šele v času povezovanja ali izvajanja in se shrani na eno izmed lokacij v tabeli odpošiljanja. Ko se metoda pokliče, se izvajanje prenese na naslov, ki se nahaja v tej tabeli na indeksu metode. Tabela odpošiljanja je potrebna samo za dinamično povezane metode, statične metode se povežejo že v času prevajanja. Najpreprostejša izvedba klica dinamične metode je z uporabo tabele, katero naslavljamo s prejemnikom in selektorjem (indeksom). Slabost takšne predstavitve je visoka prostorska zahtevnost tabele. Če imamo $r \in [1..n]$ razredov in $s \in [1..m]$ selektorjev je velikost tabele $O(r * s)$. Ker je večina sporočil definirana le v nekaterih izmed razredov r, je tabela zelo razpršena, kar povzroča slabo prostorsko izkoriščenost. Ne glede na hitrost dostopa do naslova metode, ki je enaka za statično in dinamično tipiziranje, se takšne tabele ne uporabljajo prav pogosto.
Najpogostejša tehnika za resolucijo virtualnih metod uporablja t.i. virtualne tabele odpošiljanja (virtual dispatch tables), ki so bile prvič predstavljene v Simuli. Danes ta pristop uporablja večina statično tipiziranih jezikov, tudi Java in C++ [93]. Virtualna tabela določa selektorje metod povsem lokalno, znotraj enega razreda. V primeru enkratnega dedovanja se selektorji označujejo zaporedno od najvišjega do najnižjega razreda v hierarhiji. V razredu, ki razume m sporočil, so torej selektorji označeni z $0..m-1$. Vsakemu razredu se priredi njegova tabela odpošiljanja velikosti m. Ker se selektorji sporočil referencirajo s številkami, morajo vsi podrazredi za sleherno dedovano metodo uporabljati enak selektor. Postopek odpošiljanja najprej naloži prejemnikovo tabelo odpošiljanja, katere indeksi referencirajo dejanske pomnilniške naslove metod.
Večkratno dedovanje otežuje številčenje selektorjev. Ker lahko posamezen razred de-

duje iz večih staršev hkrati, se lahko različnim sporočilom priredi enak selektor, kar je napačno. Ta problem se razreši tako, da se za posamezen razred uvede več virtualnih tabel. C++ uporablja originalno tabelo za razred, ki deduje in prvi razred, iz katerega deduje, za vse ostale razrede pa se generirajo dodatne virtualne tabele. Šibka stran virtualnih tabel je njihova odvisnost od statičnega tipiziranja. Brez poznavanja množice sporočil, ki jih objekt lahko prejme, ne moremo enoumno določiti številk selektorjev za vsa možna sporočila. To je problem dinamično tipiziranih jezikov. Metod, kot so barvanje selektorjev, odmik vrstic in nekaterih drugih ne bomo posebej obravnavali.

2.3 Tipiziranje

Koncept tipa ima v objektno usmerjenih programskih jezikih povsem praktično vrednost. Ne glede na to, da lahko tip gledamo kot abstrakcijo opisa ali kot vmesniški mehanizem, je to koncept, s katerim želimo zagotoviti varnost izvajanja operacij. Mehanizem, ki usklajuje in preverja smiselnost operacij nad določenimi tipi, imenujemo sistem tipov. Ker posamezen tip definira samo izbrane operacije, ga razumemo tudi kot koncept za omejevanje funkcionalnosti. Nespametno bi bilo npr. deliti znakovni niz s številom 2. Prav z omejenim naborom operacij dobijo objekti svoje lastnosti, preko katerih komunicirajo z drugimi objekti. Jeziki, ki funkcionalnosti objektov ne ločujejo z njihovimi tipi, so šibko tipizirani. Šibka tipiziranost povzroča logično neskladje v interakciji med posameznimi objekti, saj nimamo primernega načina, s katerim bi zagotovili pravilnost operacij nad objekti. Ne samo razred, tudi tip, kot organizacijski princip, služi za klasifikacijo objektov. Osnovna razlika je le, da opravlja razred klasifikacijo na nivoju abstraktnih podatkovnih tipov, obnašanja in na strukturalnem nivoju, medtem ko tip klasificira funkcionalnosti objektov v času izvajanja. Z dobro načrtovano strukturo tipov minimiziramo in lokaliziramo odvisnosti med posameznimi abstrakcijami. Ne potrebujemo natančne strukture objekta ali njegovih implementacijskih podrobnosti, dovolj je, da poznamo tip tega objekta, saj nam samo ta zagotavlja natančno specifikacijo objektovih lastnosti.

2.3 Tipiziranje

2.3.1 Sistemi tipov

Sistem tipov je nadvse pomemben notranji mehanizem programskega jezika, s katerim se zagotavlja varnost izvajanja sleherne operacije. Statični sistemi tipov imajo to prednost, da nesmiselne operacije preprečijo že v času prevajanja in s tem pohitrijo izvajanje programov. V času izvajanja je potrebno preveriti samo operacije, katerih končni tip ni mogel biti predviden v času prevajanja. Sistem tipov v našem kontekstu povezujemo z močno tipiziranimi programskimi jeziki, kjer ima vsak izraz svoj tip, s katerim je določen nabor dovoljenih operacij nad tem izrazom. Ob zagotavljanju legalnosti operacij nad posameznim izrazom, je sistem tipov odgovoren tudi za sklepanje o tipih v primeru operacij nad izrazi različnih tipov. Sklepanje pride v poštev, kjer za zagotovitev pravilnosti operacij nad različnimi tipi nimamo dovolj informacij. S pomočjo sklepanja lahko primeren tip izpeljemo. Večina programskih jezikov omogoča implicitno sklepanje, ki ga po določenih pravilih tipiziranja vrši prevajalnik namesto programerja. V jeziku ML [41] je sklepanje tako pomembno, da predstavlja primarni koncept tipiziranja. Sistem sklepanja je v objektnih in objektno usmerjenih jezikih zelo kompleksen, saj mora poleg klasičnih razširitvenih in zožitvenih kriterijev nad integralnimi tipi dosledno upoštevati tudi hierarhično organiziranost abstraktnih tipov, t.j. razredov ali objektov.

Z zagotovitvijo kompatibilnosti med posameznimi tipi lahko preprečimo napake v času izvajanja, vendar s stališča jezikovne izraznosti statični sistem tipov pomeni precejšno omejitev, ki se včasih izkaže za veliko breme. Po drugi strani preverjanje v času prevajanja izboljša učinkovitost sistema tipov, saj ima ta bistveno preprostejšo nalogo ob izvajanju.

Dinamično tipizirani programski jeziki, kot sta Lisp in Smalltalk, preverbo tipov opravijo neposredno pred samo izvedbo operacije. To pomeni, da je tipiziranje zadržano do trenutka izvedbe. Z gledišča učinkovitosti izvajanja je dinamično tipiziranje počasnejše od statičnega. Ker se praktični del tipiziranja opravi šele v času izvajanja, lahko pride do napake tipa. Temeljni razlog, zakaj se dinamično tipiziranje vzlic temu uporablja, je preprostost zapisa algoritma in izrazna moč jezika. Dinamično tipiziranje omogoča programiranje na višji abstraktnostni ravni, saj programerja zanimajo samo vrednosti

in ne njihovi tipi.

Glede na omenjene lastnosti sistemov tipov je razumljivo, da so statično tipizirani jeziki varnejši in performančno učinkovitejši, dinamično tipizirani pa bolj prilagodljivi in izrazno močnejši. Kljub nemajhnim omejitvam statičnih sistemov tipov večina "mainstream" jezikov uporablja prav to metodo tipiziranja. Tudi Java in C++. Statično tipiziranje objektno usmerjenih jezikov je sicer varnejše, vendar še vedno ostajajo problemi pretvorbe tipov po razrednih hierarhijah. Jasno je, da izključno statično preverjanje, tj. v času prevajanja, ni vselej zadostno. Pretvarjanje tipov od splošnih k specifičnim terja preverjanje v času izvajanja. Tako ima statično tipiziranje v objektno usmerjenih programskih jezikih tudi dinamično komponento, brez katere bi bil sistem tipov skoraj neuporaben. Striktno statično tipiziranje iz tega razloga razbremenimo z zahtevo, da mora sistem tipov za vsak izraz zagotoviti skladnost tipa, čeprav sam tip v času prevajanja še morda ni znan. Java in C++ ta problem rešujeta z dinamičnimi pretvorbami tipov (type casts). Tako lahko pride do pretvorbene napake tudi v času izvajanja.

Hibridno tipizirani programski jeziki so torej statično tipizirani z dodatnimi mehanizmi preverjanja v času izvajanja. Ker v to skupino sodijo praktično vsi objektno usmerjeni jeziki, imenujemo jezike s takšnim sistemom tipov kar statično tipizirane. Za poizvedovanje o tipih programskih entitet v času izvajanja uporabljajo jeziki različne konstrukte. Javi v ta namen služi konstrukt *instanceof*, s katerim se preveri dejanski tip entitete. Operator vrne logično vrednost *true*, če je entiteta instanca tipa, podanega kot argument. V kolikor bi se preverba tipa z omenjenim operatorjem izvedla ob vsaki pretvorbi, bi Java bila povsem statično varna. O varnosti javanskega sistema tipov, statičnosti in dinamičnosti je bilo napisanih precej razprav [39, 38, 40], ki z bolj in manj formalnimi pristopi prikazujejo svoja dognanja. C++ ima v ta namen operator *typeid*, za varno pretvorbo tipov po razredni hierarhiji pa operator *dynamic_cast*. Vendar kljub konstruktom, ki omogočajo testiranje dejanskih instanc razredov in varne pretvorbe, jezika C++ še vedno ne moremo imeti za varnega, saj je uporaba omenjenih mehanizmov povsem neobvezujoča. Če bi lahko zagotovili, da bi C++ ob vsaki pretvorbi polimorfnega tipa opravil preverbo z operatorjem *dynamic_cast* in ob napaki ustrezno ukrepal, bi tudi C++ lahko imeli za varen jezik.

Teorija tipov govori, da lahko podtip stoji na vsakem mestu, kjer pričakujemo njegov nadtip. Tako je potrebno skladnost tipov preveriti samo pri razširjanju tipa v njegov podtip. Podtip ima namreč polno funkcionalnost svojega nadtipa, obratno pa nujno ne velja. Preverba tipa se izvrši tako, da prevajalnik na mesto, kjer je to potrebno, vstavi ukaz ali zaporedje ukazov, s katerimi je realiziran test tipa. Namesto prevajalnika, ki to stori implicitno, bi lahko test izvedli sami, npr. v Javi z omenjenim operatorjem *instanceof*. Postavlja se vprašanje, kako obravnavati napako tipa? Ena možnost je programska obravnava izjeme, s katero preprečimo, da bi se program končal z napako. Statično tipizirani programski jeziki, kot sta Java in C++ potrebujejo, zaradi svojega statično omejenega sistema tipov, posebne pretvorbene mehanizme izključno za izhod iz statičnega tipiziranja, v katerem tipov dinamično spreminjajočih objektov ni mogoče enoumno klasificirati. Tako Java kot C++ imata dvonivojski sistem tipov s statično komponento, ki je aktivna v času prevajanja ter dinamično, ki opravlja preverjanja v času izvajanja. Implementacija ključnih konceptov objektno usmerjenih programskih jezikov, kot so mnogoličnost, specializacija ter abstrakcija, temelji prav na dinamični naravi tipiziranja. Dinamični jeziki, kot je npr. Scheme, se soočajo z enakimi problemi tipiziranja, le da je dejansko preverjanje tipov zakasnjeno do časa neposredno pred izvršitvijo. Dinamični jeziki opravijo vsako preverbo, tudi takšno, ki bi jo lahko povsem dokončno opravili med prevajanjem, pred izvršitvijo dejanske operacije.
Sistem tipov je, ne glede ali gre za statičnega, dinamičnega ali hibridnega, osnovno vezivo med vsemi koncepti, ki jih programski jezik omogoča, saj so z njim povezane uporaba, omejitve in dejanska implementacija večine konceptov.

2.3.2 Formalne metode za opis sistemov tipiziranja

Za sistem tipov je zraven praktičnega pomemben tudi formalen pogled. Formalna specifikacija pravil tipiziranja nekega programskega jezika nudi možnost natančnega modeliranja programskih fraz ter dokazovanja in preverjanja pravilnosti tipov že v času prevajanja. Formalne metode tipiziranja iz praktičnih vzrokov temeljijo na tipiziranem računu lambda ($\lambda - calculus$) [25, 23]. Pravila tipiziranja podamo induktivno na podlagi produkcij kontekstno proste gramatike, ki generira jezik.
Tipiziran lambda račun potrebuje natančne informacije o tipih prostih spremenljivk

znotraj posameznih izrazov. Informacija o tipu vključuje pomen tipa in njegovo strukturo. Spremenljivke predstavimo kot identifikatorje povezljivih vrednosti znotraj nekega okolja. Okolje takšnih spremenljivk označimo s Π in pomeni končno množico asociacij med identifikatorji in tipiziranimi izrazi. Asociacijo zapišemo kot $s : T$, kjer je s edinstven identifikator v okolju Π in T njegov pripadajoči tip. Če velja relacija pripadnosti $s : T \in \Pi$, to zapišemo kot $\Pi(s) = T$. Da bi bila pravila tipiziranja razširljiva, je potrebno dopustiti možnost vključevanja definicij novih tipov. Ker tip pomeni asociacijo med identifikatorjem tipa in njegovo definicijo, se lahko identifikator tipa v pravilu nadomesti z njegovo definicijo. Definicijo tipa zapišemo z relacijo $t = T$, kjer je t identifikator tipa in T izraz tipa. Sistem omejevanja tipov označimo s $\mathcal{S}$ in ga definiramo na sledeč način:

- prazna množica $\emptyset$ je sistem omejevanja tipov,
- če je $\mathcal{S}$ sistem omejevanja tipov in t identifikator tipa, ki še ni v $\mathcal{S}$ ali T, potem je tudi $\mathcal{S} \cup \{t = T\}$ tak sistem.

Pri tolmačenju tipiziranih izrazov in drugih programskih konstruktov mora sistem $\mathcal{S}$ vsebovati opisne definicije vseh tipov, ki se pojavijo v kontekstu konstruktov in izrazov. Sistem $\mathcal{S}$ ne sme biti statičen, saj mora imeti mehanizem za odstranjevanje obstoječih definicij tipov v trenutku, ko ti prenehajo obstajati. Sodobni programski jeziki imajo polimorfne sisteme tipov, kar pomeni, da je takšno naravo tipov potrebno upoštevati tudi pri formalnih modelih tipiziranja. Sistem tipov mora imeti poleg monomorfnih tipov tudi možnost definicije polimorfnih tipov.
Sistem $\mathcal{S}$ prenesemo v funkcijo $\mathcal{S}(T)$, katera vrača izraz tipa tako, da se vsi identifikatorji tipa v T nadomestijo s svojimi definicijami v $\mathcal{S}$. Za sistem tipov, ki vključuje konstantne in referenčne tipe, razredne in objektne ter funkcijske tipe, lahko to funkcijo za izraz tipa T definiramo na sledeč način:

- če je t identifikator tipa, je $\mathcal{S}(t) = \begin{cases} \mathcal{S}(T) & , \{t = T\} \in \mathcal{S} \\ t & , \text{drugače} \end{cases}$,
- če je c konstanten tip, je $\mathcal{S}(c) = c$,
- $\mathcal{S}(T_1 \times \ldots \times T_n \to T_{n+1}) = \mathcal{S}(T_1) \times \ldots \times \mathcal{S}(T_n) \to \mathcal{S}(T_{n+1})$,

- $\mathcal{S}(ref(T)) = ref(\mathcal{S}(T))$,
- $\mathcal{S}(\mathcal{O}(M)) = \mathcal{O}\ \mathcal{S}(M)$,
- $\mathcal{S}(\mathcal{C}(I,S)) = \mathcal{C}(\mathcal{S}(I), \mathcal{S}(S))$.

Ni neobičajno, da je tip definiran rekurzivno, torej nanašajoč se na lastno definicijo. V takšnem primeru je potrebno zagotoviti končnost funkcije $\mathcal{S}(T)$, kar določa drugi postulat. Ta služi kot izhodni pogoj rekurzivne definicije tipa. Tako postavljen sistem povzroča, da je izraz tipa brez prostih spremenljivk, kadar so vse proste spremenljivke iz T vsebovane v domeni $\mathcal{S}$.

Dokazovanje tipov deluje tako, da pravila tipiziranja obdelujejo programske fraze po vrsti, kot si sledijo v programu in hranijo ažurne informacije o tipih novih identifikatorjev v $\mathcal{S}$.

Pri določevanju tipa izraza imamo opraviti z neko oceno ali sklepom. Če podamo razlago oblike $\mathcal{S}, \Pi \vdash D \diamond \mathcal{S}', \Pi'$, to pomeni, da določitev tipa deklaracije D, ob predpostavkah iz $\mathcal{S}$ in Π, obogati $\mathcal{S}$ v $\mathcal{S}'$ in okolje Π v Π'. To pomeni, da obdelava deklaracije spreminja sistem tipov in z njim okolje. Pravila tipiziranja standardno zapišemo kot množico hipotez in sklepov:

$$\frac{\mathcal{S}, \Pi \vdash D_1 \diamond \mathcal{S}_1, \Pi_1, \ldots, \mathcal{S}_{n-1}, \Pi_{n-1} \vdash D_n \diamond \mathcal{S}_n, \Pi_n}{\mathcal{S}, \Pi \vdash D \diamond \mathcal{S}_n, \Pi_n}$$

Sklep je zapisan na mestu imenovalca, hipoteze v poziciji števca. Izrazi, ki se pojavljajo v hipotezah, so povečini podizrazi tistih, ki jih zapišemo na mestu sklepov. Pravila tipiziranja beremo od spodaj navzgor in od leve proti desni. Gornje pravilo beremo "tipiziranje deklaracije D omogoča, ob predpostavkah iz $\mathcal{S}$ in Π, bogatejše sklepanje o tipih $\mathcal{S}_n, \Pi_n$, če tipiziranje deklaracije iz $\mathcal{S}, \Pi$ da $\mathcal{S}_1, \Pi_1$, ... in $\mathcal{S}_{n-1}, \Pi_{n-1}$ da $\mathcal{S}_n, \Pi_n$". Na tak način zapišemo pravila za vse možne deklaracije in konstrukte programskega jezika.

Tipiziranje preprostega izraza je podobno. Pravila tipiziranja zapišemo v obliki:

$$\mathcal{S}, \Pi \vdash e : T$$

Pravilo prebermo "ob predpostavkah iz $\mathcal{S}$ in okolja Π, sklepamo, da ima e tip T". Da bi natančno definirali sistem tipov nekega programskega jezika, je potrebno zapisati pravila za vse izraze, ki jih jezik omogoča. Formalen opis sistema tipov mora upoštevati praktično vse lastnosti jezika: polimorfnost tipov, shemo dedovanja, tipiziranje s specialnimi tipi itd.

2.3.3 Teoretični vidik podtipov

Teoretična osnova tipiziranja in koncepta podtipa se v največjem razmahu realizira v objektno usmerjenih programskih jezikih. Praktična uveljavitev podtipa doseže svoj vrh v ideji dedovanja. Praktični vidik podtipov je v tesni navezi s komponentnim načrtovanjem z uporabo dedovanja. Kot sami tipi, so tudi podtipi zgolj sredstvo za modularizacijo abstrakcij, vendar na bolj razdrobljenem nivoju. S podtipi opisujemo relacije med posameznimi, bolj oddaljenimi, vendar sorodnimi tipi. Če je tip opisan z razredom, potem podrazred dobi vlogo podtipa. Čeprav je to povečini smiselno in upravičeno, je vloga razreda in tipa povsem različna. Tipi in podtipi obstajajo ločeno od razredov in niti ni nujno, da hierarhija razredov smiselno opisuje pripadajočo hierarhijo tipov. Koncept podtipa je vpeljal polimorfno obnašanje objektov, saj lahko isti objekt v različnih kontekstih operira na različne načine. Ne glede na način opisa podtipa, ta vselej predstavlja specializiran, bolj omejen zunanji vmesnik, kateremu zadošča v svoji specifikaciji. Edina razlika med tipom in podtipom je v tem, da ima podtip nujno svoj nadtip, medtem ko za tip to ni obvezujoče. Podtip torej sam zase ne obstaja. Lahko pa se zgodi, da je v relaciji z več kot enim nadtipom hkrati. V tem primeru podtip ne samo specializira obnašanja nadtipov, temveč združi njihove vmesnike. Takšno združevanje je možno v Javi in $C^{\#}$ z večkratnim dedovanjem vmesnikov.
Če za trenutek zanemarimo koncepta razreda in objekta ter se osredotočimo zgolj na tip, lahko relacijo podtipa formalno zapišemo z izrazom $P <: T$, kjer P predstavlja podtip tipa T. Praktično gledano, relacija podtipa zagotavlja pravilnost njegove uporabe, saj je podtip P lahko uporabljen na vsakem mestu, kjer pričakujemo tip T. Vrednost tipa P ima lastnosti vrednosti tipa T. Tip T imenujemo tudi nadtip ali supertip tipa P. Ker ima lahko ena vrednost več tipov hkrati, tip P in vse njegove nadtipe, govorimo o polimorfni naravi podtipov. Podtip opisuje operacije svojega nadtipa, katerega z do-

datnimi specifikacijami še izpopolni in specializira v nov tip. Ker gre za vključevanje obstoječih definicij nadtipa, govorimo tudi o vključitvenem polimorfizmu. Programski jezik običajno podpira strukturalno ali eksplicitno tipiziranje. Slednje je prisotno v Javi in v C++, kjer je treba relacijo med tipom in nadtipom navesti eksplicitno.
V kolikor želimo varen in formalno preverljiv sistem tipov, mora podtip, kjer je uporabljen na mestu nadtipa, zadoščati celotni definiciji tega nadtipa. Iz tega sledi, da lahko podtip torej definira nekatere nove funkcionalnosti, ne sme pa obstoječih odstraniti ali jih spremeniti do mere nekompatibilnosti signatur. Funkcionalnost, ki je definirana v okolju pričakovanja tipa, se zanaša samo vmesnik tega istega tipa in ne na vmesnik tipov, ki so izpeljani iz njega.

Relacija podtipa vpliva na sleherno tipizirano frazo programa. Če ima programska fraza tip P, ima po zgoraj zapisani relaciji podtipa $P <: T$, tudi tip T. To pomembno pravilo, ki ga imenujemo subsumiranje, omogoča, da lahko sistem tipov že na statični ravni, tj. v času prevajanja, preveri eksaktnost programske fraze v kontekstu, kjer je pričakovan nadtip njenega lastnega tipa. Objekt lahko "emulira" drug objekt le v primeru, da ima ta manj metod. Objekt, ki emulira oz. oponaša drug objekt ima namreč razširjen vmesnik, v katerega je vključen tudi vmesnik drugega objekta. Tipiziranje z vpeljavo relacij med tipi in njihovimi podtipi oz. nadtipi izboljšuje prilagodljivost in razširljivost pravilno tipiziranih izrazov, saj lahko njihove vrednosti klasificiramo v hierarhično strukturo tipov. Zamenjava vrednosti nadtipa z vrednostjo njegovega izpeljanega tipa v programski frazi nima skoraj nikakršnega vpliva na implementacijski nivo jezika. Objektno usmerjeni programski jeziki s čistim referenčnim modelom objekte opisujejo kot reference, ki so, ne glede na strukturo posameznega objekta, vedno enake dolžine. Tako je pri nadomestitvi izraza podtipa v kontekst, kjer se pričakuje nadtip, potrebno preveriti samo logično eksaktnost podtipa. V sklopu sistemov tipiziranja omenimo še pojem minimalnega tipa. V čistem objektnem sistemu tipov velja, da ima vsaka tipizirana vrednost vsaj svoj minimalni tip. Ideja minimalnega tipa je uporabna v algoritmih tipiziranja, saj zagotavlja, da ima sleherna tipizirana vrednost svoj minimalni ali kanonični tip. V jeziku Zero je minimalni tip `Object`.

Statično ali dinamično tipizirani programski jeziki opravljajo preverjanje tipa na podlagi relacije signaturne kompatibilnosti med posameznimi tipi. Najvišja stopnja kompatibilnosti dveh tipov je signaturna ekvivalenca tipov, kar zapišemo z enačbo $P \equiv T$. Relacija ekvivalence pomeni, da lahko na mesto tipa T postavimo tip P in obratno. Relacija podtipa omogoča nadomestitev samo v eni smeri in je zato striktnejša kot ekvivalenca. Pravilo za sklepanje o tipu izraza zapišemo z izrazom:

$$\frac{\tau' <: \tau \qquad e : \tau'}{e : \tau}$$

Pravilo pravi, da lahko za vsak podtip τ' tipa τ in izraz e tipa τ' sklepamo, da ima izraz e tudi tip τ. Pravilo imenujemo enosmerno substitucijsko pravilo za tipe v izrazih, saj predpisuje relacijo substitucije oz. nadomestitve, ki velja med tipi.

Najpreprostejši neobjektni tip, za katerega lahko izpeljemo relacijo podtipa, je zapis (record). Zapis je struktura, ki je podobna kartezičnemu produktu. Vrednost zapisa je predstavljena z dvojčkom, katerega komponenti sta oznaka in pripadajoča vrednost. Zapis je fleksibilnejši od splošne n-terice, saj je vrstni red elementov zapisa lahko poljuben. Posamezni element zapisa identificiramo s pripadajočo oznako. Zapis lahko zapišemo kot $(o_1 : \tau_1,\ o_2 : \tau_2,\ \ldots, o_n : \tau_n)$, kjer so $o_1, \ldots, o_n$ imensko različne oznake, $\tau_1, \ldots, \tau_n$ pa pripadajoči tipi. Glavna operacija nad zapisom je selekcija posamezne komponente zapisa $Sel(o_i)$. Če označimo zapis z δ in določimo njegov podtip δ', mora le-ta ohraniti operacijo selekcije. Tip τ_i' vsake istoimenske komponente o_i v δ' mora biti podtip tipa τ_i komponente o_i v δ. Pravilo relacije podtipa $\delta' <: \delta$ zapišemo na sledeč način:

$$\frac{\tau_1' <: \tau_1 \ \ldots \ \tau_n' <: \tau_n}{(o_1 : \tau_1',\ \ldots, o_n : \tau_n', o_{n+1} : \tau_{n+1},\ \ldots, o_{n+m} : \tau_{n+m}) <: (o_1 : \tau_1,\ \ldots, o_n : \tau_n)}$$

Podtip δ' tipu zapisa δ je vsak zapis, ki ima dodatne komponente $(o_{n+1},\ \ldots, o_{n+m})$ ali istoimenske komponente tipov, ki so podtipi tipom komponent v δ. Primera podtipa

zapisa sta:

$$(o_1 : Integer,\ o_2 : Integer,\ o_3 : Double) <: (o_2 : Integer)$$

in

$$(o_1 : (o_1 : String,\ o_2 : String),\ o_2 : String) <: (o_1 : (o_1 : String))$$

Prvi primer podtipa zapisa imenujemo podtip v *širino*, drugega pa podtip v širino in *globino*. Relacijo podtipa za zapis bolj lapidarno zapišemo na sledeč način:

$$(o_i : \tau_i)_{1 \le i \le n} <: (o_j : \upsilon_j)_{1 \le j \le k}, \quad k \le n \quad \wedge \quad \tau_i <: \upsilon_i, \quad 1 \le i \le k$$

Takšna formulacija podtipa je primerna samo za zapis, v katerem je edina operacija selekcija komponente $Sel(o_i)$. V kolikor bi želeli komponente tudi spreminjati, bi bilo v relaciji treba uporabiti dodatne omejitve. Kadar komponente zapisa samo izbiramo, imamo opraviti s tipom vrednosti. Če želimo komponento spremeniti, potrebujemo tip reference, saj je vsaka komponenta zapisa shranjena na neki pomnilniški lokaciji. Tip spremenljivke s označimo z $ref(s)$. Pravilo podtipa za zapise, v katerih lahko posamezne komponente spreminjamo, zapišemo:

$$(o_i : ref(\tau_i))_{1 \le i \le n} <: (o_j : ref(\upsilon_j))_{1 \le j \le k}, \quad k \le n \quad \wedge \quad \tau_i \equiv \upsilon_i, \quad 1 \le i \le k,$$

kjer relacija $\tau_i \equiv \upsilon_i$ označuje vzajemnost oz. ekvivalenco podtipov $\tau_i <: \upsilon_i$ in $\upsilon_i <: \tau_i$. Podtip spremenljivega zapisa lahko torej doda nove komponente, obstoječe pa morajo imeti ekvivalentne tipe.

Pri tipiziranju v objektno usmerjenih programskih jezikih je najelementarnejša relacija funkcijskega tipa. Ker so razredi in objekti sestavljeni iz metod, je relacija funkcijskega tipa pomemben člen pri dedovanju in specializaciji. Obravnava funkcijskega tipa ima velik vpliv na izrazno moč sistema tipov, saj pogojuje spremembe, ki so dovoljene pri redefiniciji metod v podrazredih. Funkcijski tip v splošnem zapišemo $\tau_a \to \tau_v$, kjer τ_a predstavlja tip argumenta in τ_v tip vrednosti, ki jo funkcija vrača. Tip argumenta funkcije imenujemo domena, tip vračanja pa zaloga vrednosti. Operator $\to$ imenujemo funkcijski operator ali konstruktor tipa. Že omenjena funkcija selekcije komponente

$Sel(o_i)$ zapisa $(o_1 : \tau_1,\ \ldots, o_n : \tau_n)$, ima torej funkcijski tip **String** $\rightarrow \tau_i$.
Po pravilu subsumpcije, ki velja za tipe, lahko funkcijski podtip tipa $\tau_a \rightarrow \tau_v$ uporabimo, kjerkoli pričakujemo njegov nadtip. Praktično to pomeni, da lahko funkcijo, ki ima funkcijski tip τ, nadomestimo s funkcijo s funkcijskim tipom τ', če je le τ' podtip tipa τ, torej $\tau' <: \tau$. Relacijo funkcijskega podtipa zapišemo $\tau'_a \rightarrow \tau'_v <: \tau_a \rightarrow \tau_v$. Nedoslednost funkcijskega podtipa je v njegovi navidez nelogični interpretaciji. Če privzamemo, da je $\mathbf{N} <: \mathbf{R}$, bi funkcijo $\mathbf{N} \rightarrow \mathbf{N}$, ki slika iz naravnega v naravno število, zlahka imeli za podfunkcijo $\mathbf{R} \rightarrow \mathbf{R}$. Vendar pravilo podtipa zahteva, da lahko vrednost podtipa vstavimo na vsako mesto, kjer pričakujemo nadtip. Če torej na nekem mestu pričakujemo funkcijo $\mathbf{R} \rightarrow \mathbf{R}$, bi lahko na tem mestu poklicali funkcijo $\mathbf{N} \rightarrow \mathbf{N}$. Če pričakujemo, da funkcija vrne realno vrednost, njen podtip pa vrne celo število, je to veljavno, saj vsako celo število uvrstimo tudi v množico realnih števil. Drugače je pri argumentih, saj "nadfunkcija" pričakuje argument tipa **R**. Funkcijo pokličemo z argumentom realnega tipa, vendar če na njeno mesto vstavimo funkcijo, ki pričakuje naravni tip, bo to pomenilo napako, saj je množica **R** močnejša od **N**. Tip argumenta podfunkcije mora po takšnem načelu biti nadtip argumenta nadfunkcije. Pravilo funkcijskega podtipa tako zapišemo:

$$\frac{\tau_a <: \tau'_a \qquad \tau'_v <: \tau_v}{\tau'_a \rightarrow \tau'_v \ <: \ \tau_a \rightarrow \tau_v}$$

Funkcijski operator je monoton v drugem argumentu ($\tau'_v <: \tau_v$) in antimonoton v prvem ($\tau_a <: \tau'_a$). Monotonost pomeni kovariantno spremembo, antimonotonost pa kontravariantno. Tip vračanja podfunkcije je torej lahko podtip vračanja nadfunkcije, medtem ko je pri argumentih ravno obratno. Tip argumenta podfunkcije mora biti enak ali nadtip tipa argumenta nadfunkcije. Antimonotonost tipa argumenta v praksi ni pretirano uporabna, medtem ko ima monotonost tipa vračanja veliko veljavo. Prednost spremembe tipa vračanja ni samo v praktični uporabi, temveč tudi v tem, da je statično preverljiva. Kljub temu dejstvu večina statično tipiziranih programskih jezikov te prednosti ne uporablja in tako ostaja pri invariantnih sistemih tipov. Jezik Z_0 ima sistem tipov, ki omogoča tako kontravariantne kot kovariantne spremembe.

Funkcijski operator lahko apliciramo na koncept polja ali kake podobne podatkovne

strukture. Polje predstavimo s funkcijskim operatorjem $\rightarrow$, ki slika iz tipa indeksa v tip elementa polja. Privzemimo homogena polja, tj. polja, katerih elementi imaj identičen podatkovni tip. Referenciranje elementa polja P[i] v funkcijski notaciji zapišemo $\tau \rightarrow \upsilon$, kjer je τ tip indeksa i in υ tip elementov polja P. Pravilo podtipa zapišemo podobno funkcijskemu podtipu:

$$\frac{\upsilon' <: \upsilon \qquad \tau <: \tau'}{\upsilon'[\tau'] <: \upsilon[\tau]}$$

Tako zapisano pravilo ne vključuje polj, v katerih se lahko vrednosti spreminjajo. Da bi lahko vrednosti spremenili, morata biti tipa obstoječega in novega elementa ekvivalentna:

$$\frac{\upsilon' \equiv \upsilon \qquad \tau <: \tau'}{\upsilon'[\tau'] <: \upsilon[\tau]}$$

Indeks polja se spreminja, podobno kot argument funkcije, kontravariantno, medtem ko je tip elementa fiksen. S stališča variance tipov je relacija podtipa polja s spremenljivimi vrednostmi podobna zapisom s spremenljivimi komponentami. V obeh imamo neposredno referenciranje. Zato morajo tipi komponent oz. elementov biti invariantni. Java omogoča kovariantne spremembe v tipih elementov polja $\upsilon'[\,] <: \upsilon[\,], \quad \upsilon' <: \upsilon$. S tem je onemogočena statična varnost zaradi česar se mora Java zanašati na dinamično preverjanje v primeru, ko gre za prirejanje elementu polja. Takšno tipiziranje je sicer "nepravilno", vendar omogoča večjo fleksibilnost sistema tipov, saj lahko ista funkcionalnost v okviru ene metode operira nad polji različnih tipov. Sistem tipov, ki omogoča generične abstrakcije, takšen način tipiziranja odpravlja.

Večina objektno usmerjenih jezikov relacijo podtipa objektov povezuje s podtipi pripadajočih opisnih mehanizmov – razredov. Relacija podtipa razredov $\tau' <: \tau$ pogojuje istovetno relacijo objektov $\mathcal{O}_{\tau'} <: \mathcal{O}_{\tau}$, kjer $\mathcal{O}_r$ predstavlja objektni tip, tj. instanco $\mathcal{O}$ razreda r. Takšna povezava razreda in objekta ni nujna. Razred lahko kot mehanizem klasifikacije obravnavamo povsem ločeno od objekta. Edina povezava med tako predstavljenima razredom in objektom je vmesnik, kateremu razred služi kot definicijski

mehanizem, objekt pa predstavlja dejansko instanco. Striktna objektna naravnanost zahteva, da je edina operacija, ki se lahko izvaja nad objektom pošiljanje sporočila za izvedbo metode. Operacija pošiljanja sporočila je zelo podobna selekciji komponente zapisa. Objekt, ki ima fiksno signaturno množico, lahko torej predstavimo kot konstanten zapis. Posamezne komponente zapisa nadomestijo metode, za katere velja že definirana relacija funkcijskega podtipa. Objektni tip z naborom definiranih metod zapišemo kot $\mathcal{O}\{m_i : M_i\}$, kjer m_i predstavlja abstrakcijo metode in M_i njen funkcijski tip. Razred objekta na tem mestu nima posebnega pomena, zato ga izpustimo. Relacijo objektnega podtipa sedaj zapišemo na sledeč način:

$$\mathcal{O}\{m_i : M_i'\}_{1\leq i\leq n} <: \mathcal{O}\{m_j : M_j\}_{1\leq j\leq k}, \quad k \leq n \quad \wedge \quad M_i' <: M_i, \quad 1 \leq i \leq k$$

Ker je M funkcijski tip, lahko za konkretni primer i zapišemo:

$$M_i' = \tau_a' \rightarrow \tau_v'$$

ter

$$M_i = \tau_a \rightarrow \tau_v.$$

Ker velja tudi relacija $M_i' <: M_i$, mora po pravilu za podtip funkcijskih tipov veljati tudi $\tau_a <: \tau_a'$ in $\tau_v' <: \tau_v$.

Objektni tip $\mathcal{O}'$ karakteriziramo kot podtip objektnemu tipu $\mathcal{O}$, če $\mathcal{O}'$ vsebuje vse metode iz $\mathcal{O}$, za katere velja monotonost tipov zalog vrednosti in antimonotonost funkcijskih domen. Če na razred ali objekt gledamo kot na zapis, lahko po pravilu podtipa za zapis, tip $\mathcal{O}'$ definira lastne metode, nedefinirane v $\mathcal{O}$. Tipiziranje z objektnimi tipi ima daljnosežen vpliv na varianco sistema tipov. Prav tukaj se določa, ali bo sistem tipov programskega jezika varianten ali invarianten.

Ob objektnem tipu je potrebno določiti še relacijo razrednega podtipa. Razredni tip je smiselno vpeljati samo za jezike, ki za klasifikacijo in opis abstraktnih podatkovnih

tipov uporabljajo konstrukt razreda. Razredni tip, kakor je smatran v večini razrednih objektno usmerjenih programskih jezikov, vključuje informacije o tipih instančnih spremenljivk in signaturah metod. Omejitve relacije razrednega tipa so podobne tistim iz relacije objektnega tipa. Da bi določili, katere metode in instančne spremenljivke lahko v podrazred dodamo, moramo poznati nabor le-teh iz nadrazreda. To velja za sistem tipov, za katerega je zahtevana statična varnost; dinamično tipizirani prototipni jeziki so tukaj veliko prilagodljivejši. Razredni tip opišemo kot $\mathcal{C}\{I, S\}$, kjer je I zapis, katerega komponente predstavljajo nabor instančnih spremenljivk. Z zapisom S predstavimo množico metod. Kadar so vse instančne spremenljivke znotraj razreda definirane s privatnimi specifikatorji dostopa (so nevidne navzven), je tip objekta, ki nastane kot instanca razreda $\mathcal{C}\{I, S\}$ podan s poenostavljenim objektnim tipom $\mathcal{O}\{S\}$. Vidimo, da ima kapsulacija s skrivanjem instančnih spremenljivk tudi formalno podlago pri tipiziranju. Za zapis relacije, s katero lahko ugotovimo, kateri razredni tip je podtip drugega razrednega tipa, je treba določiti operacije, ki se lahko izvajajo nad konstruktom razreda.

Ker je razred opisni mehanizem za specifikacijo obnašanja in funkcionalnosti objektov, je najpomembnejša operacija nad razredom zagotovo ustvarjanje objekta oz. stvaritev instance. Objekt nastane z instanco razreda in to operacijo zapišemo $Inst(\mathcal{C})$. Druga, nič manj pomembna operacija, je konstrukcija podrazreda z dodajanjem novih metod ali redefinicijo že obstoječih v starševskem razredu. Podobno kot to velja za objektni tip, je razredni podtip tipa $\mathcal{C}\{I, S\}$ tip $\mathcal{C}'\{I', S'\}$. Relacija razrednega podtipa je v navezi z relacijama med množicama instančnih spremenljivk I in I' ter množicama metod S in S'. Če mora veljati relacija $\mathcal{C}' <: \mathcal{C}'$ in če $Inst(\mathcal{C}')$ ustvari objektni tip S', potem mora ta objektni tip biti podtip S. Relacija podtipa za razredni tip potemtakem zahteva $S' <: S$. Če dalje velja, da ima razred $\mathcal{C}'$ tip $\{I', S'\}$, razred $\mathcal{C}$ tip $\{I, S\}$ in je $\mathcal{C}'$ podtip $\mathcal{C}$, potem lahko po pravilu subsumpcije $\mathcal{C}$ nadomestimo s $\mathcal{C}'$. Vsaka metoda m iz $\mathcal{C}$ je lahko v $\mathcal{C}'$ redefinirana kot m' ob pogoju, da ima isto signaturo v S in S'. Drug pogoj je, da se v S' nahaja natanko toliko metod kot v S, saj drugače ne moremo zagotoviti kompatibilnosti med $\mathcal{C}$ in razredi izpeljanimi iz $\mathcal{C}'$. Zahtevamo torej ekvivalenco $S \equiv S'$. Relacijo zapišemo z izrazom:

$$\mathcal{C}'\{I', S'\} <: \mathcal{C}\{I, S\}, \quad I \equiv I' \quad \wedge \quad S \equiv S'$$

Zapisana relacija podtipa razrednih tipov je preprosta in nazorna, saj ne vključuje konceptov skrivanja instančnih spremenljivk I in metod S. Specifikatorji dostopa zakomplicirajo relacijo, saj je treba upoštevati še dostopnost in nedostopnost posameznih spremenljivk in metod v podrazredih. Privatni specifikator dostopa (*private*) v C++ in Javi pomeni, da je instančna spremenljivka ali metoda dostopna samo znotraj razreda. Privatni specifikator povzroči, da se metoda ali spremenljivka odstrani iz seznama, kar ima vpliv na pravila pri dedovanju. Razred, ki poleg privatnih in javnih specifikatorjev vsebuje še zaščitene (*protected*), lahko razširimo z obogatenim naborom $\mathcal{C}\{I_p, S_p, S_z, S_j\}$, kjer z I_p predstavimo privatne instančne spremenljivke, s S_p privatne metode, s S_z zaščitene in s S_j metode javnega dostopa. Zapisa za privatne spremenljivke in metode I_p ter S_p relacijo podtipa poenostavita, saj dostop do njih v podrazredu ni mogoč in ju tako lahko odstranimo.

2.3.4 Polimorfno tipiziranje

Danes skoraj ni več programskega jezika s čistim monomorfnim sistemom tipov. Omejitve monomorfnega tipiziranja so prevelike za praktično rabo v implementaciji programske opreme. Polimorfizem razširja moč sistema tipov s tem, da omogoča klasifikacijo tipov na različnih hierarhičnih ravneh. Polimorfizem si najlažje predstavljamo z operacijo nad nekim podatkovnim tipom. Pravimo, da je operacija polimorfna, če lahko enolično operira nad različnimi tipi. To lahko dosežemo z "ad-hoc" polimorfizmom, kjer se funkcionalnost operacije preprosto prilagodi tipu. Takšna rešitev je neelegantna in ne omogoča neskončnih množic tipov. Boljši in danes bolj razširjeni metodi sta parametrični [4, 77] in vključitveni polimorfizem.

Koncept dedovanja vpeljuje vključitveni polimorfizem. Po načelu tranzitivnosti ima vsak podtip nekega tipa tudi tip vseh svojih nadtipov. Po pravilu subsumpcije lahko funkcionalnost, ki je definirana na nekem tipu τ, povsem transparentno operira nad podtipi $\tau'_1, \ldots, \tau'_n$, če so slednji podtipi tipa τ. Vključitveni polimorfizem implicitno izkoriščajo vsi objektno usmerjeni programski jeziki, saj je koncept pridobljen "zastonj"

in ne zahteva dodatne implementacije. Java uporablja vključitveni polimorfizem tudi pri čistih tipskih abstrakcijah – vmesnikih. Tako ideja vključitvenega polimorfizma ni lastna le razredom.
Prehod na vključitveni polimorfizem zahteva dinamične mehanizme tipiziranja, saj se preverjanje vrši v času prevajanja. Obravnava specializiranih tipov, ki vsebujejo nadtipe oz. bolj splošne tipe, je zamudno, saj je potrebno pri pretvorbi v dejanski tip izvesti dinamične preverbe. Podatkovne strukture, npr. polja ali seznami, ki operirajo nad abstraktnim tipom, lahko vsebujejo vsak njegov podtip. To ni vedno zaželjeno, še najmanj pa, kadar imamo opravka z uniformno podatkovno strukturo, v kateri pričakujemo, da bodo vsi elementi istega tipa.

2.3.5 Parametrizacija tipov

Najbolj učinkovito izrabo koncepta polimorfizma omogoča parametrizacija tipov. Koncept tipiziranja s parametriziranimi tipi imenujemo parametrični polimorfizem ali mehanizem generičnih tipov. Koncept je zelo dobro uveljavljen v sodobnih jezikih [48]. Glavna ideja tega koncepta je v tem, da se tip poda kot parameter. Prednosti pred ostalimi koncepti polimorfizma se pokažejo predvsem v varnosti, izrazni moči ter učinkovitosti. Varnost tipiziranja izhaja iz dejstva, da je večino napak nekompatibilnosti tipov mogoče odkriti že v času prevajanja. Ob določitvi tipa parametra neki parametrizirani strukturi, se lahko preveri ali dejanski tip funkcionalno ustreza zahtevam abstraktnega tipa.

Sistem tipov, ki omogoča njihovo parametrizacijo, je tudi čistejši in omogoča bolj zgoščen zapis, saj se izognemo nepotrebnim in včasih nepreglednim pretvorbam med abstraktnimi in konkretnimi tipi. Največja prednost parametriziranih tipov leži v njihovi učinkovitosti. Abstraktni tip se namreč v času ustvaritve parametrizirane strukture nadomesti s konkretnim, ki je podan kot parameter. Pri obdelavi konkretnega tipa v času izvajanja ni potrebnih več nobenih pretvorb.
Formalno lahko parametriziran tip razumemo kot funkcijo, ki slika iz tipa v tip. Funkcija, ki realizira parametrizirani razred, prejme konkreten tip, s katerim se vrši instanciranje in vrne razred prikrojen tipu, podanemu kot parameter. Parametriziran razred

označimo s $\mathcal{C}^p$ in funkcijo preslikave f_t s parametrom τ zapišemo na naslednji način:

$$f_t : \quad \tau \rightarrow \mathcal{C}^\tau$$

Funkcija preslikave tipa lahko operira nad argumentom, ki je sam funkcija preslikave tipa. Tako dobimo večnivojsko parametrizacijo:

$$f_t : (\quad \tau \rightarrow \mathcal{C}^\tau \quad) \rightarrow \mathcal{C}^{\tau'} \qquad \tau' \equiv \tau \rightarrow \mathcal{C}^\tau$$

Prednost parametriziranih tipov leži v njihovi splošnosti, njihova slabost pa v implementaciji. Slednje je še posebej očitno v prevedenih in polprevedenih programskih jezikih. S tipom parametriziranega razreda ni mogoče prevesti na enak način, kot bi to storili z navadnim razredom. Problem nastane v tem, da v času prevajanja parametra ni mogoče določiti. V programskih jezikih z neuniformnim objektnim modelom ne moremo vnaprej določiti velikosti entitete tipa, ki bo podan v času izvajanja s parametrom. Od velikosti entitet so odvisni naslovi izvršnega koda in s tem tudi sama struktura. Iz tega razloga je potrebno na nek način zagotoviti strukturalno pravilnost ne glede na velikost entitet nepoznanih tipov. Preprosta rešitev tega problema je v uniformnem objektnem modelu, kjer so vse entitete predstavljene z referencami. Tako poenotimo velikost. Jezik C++ uporablja metodo vstavljanja, kjer se v času prevajanja abstraktni tip nadomesti z dejanskim. Slabost tega pristopa je v tem, da je potreben izvorni kod parametriziranega razreda, kar ni vedno mogoče. Jezik *C#* parametrizacijo prenaša tudi na neuniformne tipe [55, 109]. Če parametrizacija tipov v jeziku ni neposredno podprta, jo lahko simuliramo s koncepti, ki so v jezik že vgrajeni. Najpreprostejši, vendar najmanj fleksibilen način je predprocesiranje vhoda. Drug način je uporaba refleksijskih mehanizmov meta arhitekture [104].

2.3.6 Polimorfizem z omejitvami

Tudi splošnost s parametriziranimi tipi včasih ni dovolj. Velikokrat se pojavi zahteva, da je tip omejen na neko specifično domeno oz. da zadošča omejitvam nekega drugega tipa. To nam omogoča bolj razdelan nadzor nad definiranjem generične funkcionalnosti. Takšno podzvrst parametričnega polimorfizma imenujemo omejen polimorfizem.

Vendar ima tudi omejen polimorfizem svoje slabosti. Te se izkažejo pri t.i. binarnih metodah [65], ki prejemajo vrednost tipa lastnega razreda. Primer dobro znane binarne metode je `equals` v Javi. Ker ima Java invarianten sistem tipov, metoda `equals` vedno operira nad argumentom tipa `Object`, čeprav je tip binarne metode več kot `Object`. Zaradi teh težav je nastala posplošena oblika parametričnega polimorfizma, imenovana F omejen polimorfizem (F-bounded polymorphism) [22]. Ta oblika polimorfizma ne omogoča samo omejevanje abstraktnega tipa na nek drug tip, temveč tudi parametriziranje tega drugega tipa. Koncept je bil zaradi svoje praktične uporabnosti implementiran tudi kot razširitev Jave [18, 17, 75].

3 Jezik Z_0

Jezik Z_0 je statično tipiziran objektno usmerjen splošnonamenski programski jezik. Jezik Z_0 ima, drugače od večine modernih industrijskih jezikov, čist objektni model, kar pomeni, da so vse podatkovne entitete in krmilne strukture predstavljene kot objekti. Jezik na ta način doseže čisto "objektno naravo". Objektizacija podatkovnih entitet je samoumevna, predstavitev temeljnih jezikovnih konstruktov z objekti pa ima dodatne prednosti, ki pri "klasični" predstavitvi ostanejo zakrite. Jezikovni konstrukti, predstavljeni kot objekti, so zanke, nekateri stavki in bloki. Prednost objektizacije konstruktov je v unifikaciji vseh jezikovnih entitet. Ta omogoča bolj homogen teoretični semantični model in možnost aplikacije parametričnega polimorfizma na vse entitete jezika. Podoben princip uporablja jezik Smalltalk [62], ki je pravtako čist objektno usmerjen jezik, vendar dinamično tipiziran. Statično tipiziranje jezika Z_0 omogoča višjo stopnjo varnosti sistema tipov, saj se večina tipov in njihovih pretvorb preveri že v času prevajanja. Cilj načrtovanja jezika Z_0 je bil poiskati kompromisne rešitve med statičnim sistemom tipov, splošno učinkovitostjo ter izrazno močjo jezika. Resnično učinkovito izvajanje programskega koda je možno samo v primeru, ko je le-ta preveden v obliko, primerno za izvedbo na nekem ciljnem procesorju. Če to ni mogoče, je priporočljivo programski kod prevesti v vmesno obliko, ki je primerna za izvedbo na specifičnem virtualnem stroju. Semantična in strukturalna zapletenost večine dinamičnih jezikov zahteva enako zapleten proces prevajanja, kar je razlog, da je večina dinamičnih jezikov interpretiranih.

Programski kod jezika Z_0 je preveden v vmesno obliko, ki se izvaja v virtualnem stroju "Z". Jezike, prevedene v vmesni kod, imenujemo polovično prevedene. Prednost polovičnega prevajanja je v tem, da je vmesni kod navadno precej preprosteje prevesti v izvršni kod ciljne arhitekture, kot če bi to želeli napraviti neposredno. Vzrok temu je semantični prepad, ki se s stopnjo abstraktnosti programskega jezika povečuje. Z jezikom Z_0 smo pokazali, da je v okviru statičnega sistema tipov, ki ima znane omejitve [34, 107], mogoče realizirati konstrukte, ki imajo bistveno bolj dinamično obnašanje, kot to nakazujejo sodobni statično tipizirani programski jeziki.

Jezik Z_0 omogoča večkratno dedovanje. Pravila dedovanja so preprosta in razumljiva.

Izrazna moč jezika se s poenostavitvijo pravil ni zmanjšala, saj za to skrbi varianten sistem tipov, ki je realiziran v redkih statično tipiziranih jezikih, čeprav je statično preverljiv in omogoča bistvene prednosti pri tipiziranju. Konstrukte jezika Z_0 smo poskušali napraviti karseda kanonične. Vsak konstrukt jezika je definiran na enoumen in razumljiv način. Konstrukt mora imeti natančno specificirano nalogo in namen, od katerega ne sme odstopati. Kanoničnost ali ortogonalnost konstruktov zahteva elementarnost funkcionalnosti. Ortogonalnost konstruktov pomeni, da za vsako stvar obstaja samo en specifičen konstrukt. Objektno usmerjena jezika $C^{\#}$ in C++ nimata ortogonalnega konstrukta kar zadeva abstrahiranje nad podatki, saj lahko le-to izvedemo z uporabo konstruktov razreda ali strukture.
Sintakso jezika Z_0 smo zasnovali z namenom, da bi bila preprosta, berljiva in razumljiva. Spričo popularnosti in razširjenosti jezika Jave smo sintakso jezika Z_0 napravili podobno Javini, z določenimi malenkostnimi spremembami in dodatki.

3.1 Jezikovni konstrukti

Jezik Z_0 ima dve vrsti konstruktov. Prvi so tisti, ki omogočajo abstrahiranje nad podatki in njihovo kapsulacijo, drugi pa krmiljenje. Ker gre za razredni objektno usmerjen jezik, je konstrukt za abstrakcijo razred. Abstrakcija na osnovi razreda vključuje klasifikacijo, kapsuliranje in skrivanje. V prejšnjem razdelku smo omenili koncept ortogonalnosti konstruktov. Ker je razred edini konstrukt za abstrakcijo nad podatki, smo tej zahtevi zadostili. Razred v jeziku Z_0 pa ni samo konstrukt klasifikacije, temveč tudi tipiziranja. Jezik Z_0 namreč ne vključuje posebnega konstrukta za definicijo tipa, kot npr. `interface` v Javi, zato je treba razredu dodeliti tudi vlogo tipa.
Imperativna narava jezika Z_0 implicira operacije nad spremenljivkami. Če čisti funkcijski jeziki ne poznajo stranskih učinkov, je uporaba le-teh v imperativnih zelo pogosta. Prav zaradi tega imajo praktično vsi tovrstni jeziki možnost konstruiranja krmilnih struktur (iteracije, izbira, ...). Jezik Z_0 ima posebne konstrukte iteracijskih stavkov *while*, *do/while* in *for*. Čisti objektni programski model zahteva, da so vse entitete jezika predstavljene kot objekt, tudi krmilni konstrukti. Jezik Z_0 vse krmilne konstrukte tretira kot objekte posebnih metarazredov. Tako ohranimo uniformen način predsta-

vitve. Temeljna operacija vseh krmilnih struktur jezika je podana z metodo `exec`, ki funkcionalnost strukture izvrši. Pobliže jo bomo pogledali v sklopu jezika Zero.
Najprimitivnejši krmilni konstrukt je blok. Hkrati je blok tudi najelementarnejša samostojna enota funkcionalnosti. Blok jezika Z_0 je funkcionalno enak blokom jezikov Jave in C++. Toda blok v jeziku Z_0 je prvorazredna vrednost, predstavljena kot objekt. To pomeni, da lahko z njim manipuliramo na enak način kot s spremenljivko ali literalom. Prvorazredne vrednosti lahko ustvarjamo, shranjujemo, prenašamo in nad njimi invociramo metode. V večini objektno usmerjenih jezikov ima blok povsem sintaktični pomen. Njegova deklaracija pomeni združevanje stavkov, ki so na nek način povezani. Deklaracija spremenljivk znotraj bloka na zunanje deklaracije nima vpliva. Na ta način dosežemo skrivanje in lokalizacijo deklaracij, ki se nanašajo samo na del programskega koda. Skratka, blok je sintaktična struktura, ki se pri prevajanju izgubi, tako da v času izvajanja več ne obstaja. V nasprotju s tem, objektni model jezika Z_0 bloke ohranja tudi v času izvajanja. Bloki so primitivni objekti, nad katerimi lahko izvajamo operacije definirane v metarazredih. Blok se ustvari dinamično na mestu, kjer je definiran. Objektna predstavitev nam omogoča, da z blokom manipuliramo kot s spremenljivko tipa `Block`. Ker je smiselno, da blok referencira spremenljivke definirane v starševskih okoljih, mora blok imeti način dostopa do teh spremenljivk. Iz tega razloga blok vključuje okolje bloku lastnih spremenljivk, ki so definirane znotraj bloka, kot tudi vsa okolja starševskih blokov, katerih spremenljivke se referencirajo. Takšna implementacija predstavlja kompromis med prostorsko in časovno učinkovitostjo. Blok, ki vsebuje okolje vrednosti in programski kod, imenujemo zaprtje (closure) [10, 83].

Tudi metoda je zgolj blok. Razlika med njima je v tem, da blok ni poimenovan, na metodo pa se lahko sklicujemo z imenom. Blok je v svoji zasnovi anonimen, metoda pa ne. Metoda je entiteta, ki jo sestavlja vsaj en blok. Ker se bloki ustvarjajo dinamično, se to odraža tudi v metodah. Metode so, podobno kot bloki in iteracijski stavki, prvorazredne vrednosti, kar pomeni, da jih lahko manipuliramo na enak način. Metoda je samo bolj formalna oblika bloka, saj lahko sprejme parametre in vrača vrednost določenega tipa. Metode se lahko v času izvajanja dinamično spreminjajo, pri čemer je treba zagotoviti statično varnost. Ker je metoda natančno določena s svojo

signaturo, je ta njena najpomembnejša lastnost, vsaj kar zadeva sistem tipov. Ker razredi nimajo lastnosti specificiranih v obliki instančnih spremenljivk, kot je to primer v C++ ali Javi, je dinamika metod še toliko pomembnejša. Dinamično spreminjanje metod je edini mehanizem, s katerim je mogoče spremeniti stanje objekta. Dinamično spreminjanje metod dopuščajo nekateri dinamično tipizirani objektni jeziki. Jezik Z_0 ga uporablja znotraj statičnega sistema tipov.

Spreminjanje metod ima, ob povsem praktični uporabnosti, tudi globlje teoretično ozadje. Objekt si lažje predočimo kot zbirko metod. Na instančno spremenljivko lahko gledamo kot na okleščen koncept metode. Instančna spremenljivka je preprosto metoda, ki ne uporablja implicitnega parametra lastnega objekta (`self`). Pravtako lahko skrivanje objektovega stanja posplošimo na metode, saj imajo tudi te specifikatorje dostopa. Enačenje instančnih spremenljivk z metodami omogoča trivialno pretvorbo med njimi, saj lahko pasivne podatke aktiviramo v kalkulacijo in obratno. Unifikacija instančnih spremenljivk in metod pomeni preprostost, saj je struktura objekta identična znotraj in zunaj njega. Tudi lastne metode objekta spreminjajo objekt samo preko invokacij. Ločevanje metod od spremenljivk pomeni dva različna načina manipulacije objekta. Eksplicitna manipulacija stanja poteka preko instančnih spremenljivk, implicitna z invokacijo metod.

Unifikacija ponuja preprostejši formalni model jezika Z_0. Spreminjanje instančnih spremenljivk se preprosto preslika v spreminjanje metod. Spreminjanje metode lahko služi kot osnova za obliko dinamičnega dedovanja, saj omogoča dinamiko na nivoju metod. Povrhu tega je dinamika metod statično preverljiva, saj za spreminjanje metod zahtevamo enaka pravila kot pri dedovanju. Na ta način lahko simuliramo dinamično dedovanje, ki v splošnem ni statično varno [96] in hkrati zagotovimo konformanco tipov. V jeziku Z_0 se je tak metodno usmerjeni princip izkazal za zelo uporabnega.

3.2 Objektni model

Objektni model jezika Z_0 opisujejo lastnosti, ki določajo njegovo objektno usmerjenost. Ena izmed teh je uniformna predstavitev jezikovnih konstruktov. V jeziku Z_0 je vsak podatek in vsak jezikovni konstrukt predstavljen z objektom pripadajočega tipa. S tem

ni zagotovljena samo unifikacija predstavitve, temveč tudi koherentnost načina dostopa in manipulacije entitet. Objektni model jezika je bil zasnovan tako, da je koncepte jezika mogoče učinkovito implementirati z omejitvijo statičnega sistema tipov.

Sklicevanje na entitete je v jeziku Z_0 realizirano uniformno preko referenciranja, ki je pravzaprav samo kazalec na pomnilniško lokacijo. S "čistim" referenčnim modelom dosežemo poenostavljen formalni model jezika in preprostejšo implementacijo. Ker gre za statično tipiziran programski jezik, je tip entitete znan že v času prevajanja. Kljub temu so v času izvajanja potrebne preverbe pri pretvarjanju tipov iz splošnih k specifičnim. Objekt se v jeziku Z_0 imenuje instanca razreda, iz katerega se je (objekt) ustvaril. Korenski razred celotne hierarhije je `Object`. Vsi podatki so predstavljeni kot objekti, ne glede na to ali gre za primitivne ali uporabniško definirane vrednosti.

Temeljna operacija nad objektom je invokacija metode. Jezik Z_0 to zagotavlja s tem, da ne omogoča instančnih spremenljivk. Razred ima samo metode, tj. funkcijske in proceduralne abstrakcije. Sleherna manipulacija stanja objekta se vrši preko invokacij razrednih metod. Invokacijo metode razumemo kot odgovor na sporočilo objektu, pri čemer ime sporočila sovpada z imenom metode. Metoda v jeziku Z_0 je sama del stanja objekta. Objektovo stanje je popolno definirano z instančnimi metodami. S tem bolje razumemo potrebo po konceptu dinamičnega spreminjanja metod. Metode niso več vezane na družino objektov istega tipa, temveč opisujejo stanje vsakega objekta posebej. Vse, kar ostane nespremenjeno za vse objekte istega tipa, je programski kod in funkcionalnost metod. Posplošitev objektovih lastnosti v metode ima korenine v teoriji objektnega računa (object calculus) [2, 69]. Unifikacija metod in atributov pomeni čistejši objektni model jezika, saj zagotavlja, da je edina operacija nad objektom invokacija metode. Manipulacija je tako bolj poenotena, saj se stanje objekta manipulira izključno preko metod. Formalni opisi objektnega računa so čistejši, kadar imamo opraviti samo z instančnimi spremenljivkami ali samo z metodami, vendar ne z obojim hkrati.

Soroden objektni model jeziku Z_0 ima Smalltalk. Objekti primitivnih tipov so v Smalltalku konstantni. To pomeni, da se njihova vrednost nastavi samo enkrat in ostane nespremenjena celoten življenjski cikel objekta. Nezmožnost spreminjanja objektove vrednosti ima velik vpliv na učinkovitost izvajanja. Namesto spremembe vrednosti se

konstantni objekti ustvarjajo na novo. Ustvarjanje objekta je časovno zahtevna operacija, zato bi se ji bilo bolje izogniti, če je to le mogoče. Primitivni objekti so v jeziku Z_0 implementirali tako, da jih je mogoče ustvarjati na novo, klonirati in spreminjati njihove vrednosti. S tem se izognemo nepotrebnemu poustvarjanju objektov, ki bi se sicer pojavili kot rezultati vmesnih kalkulacij. Seveda je spreminjanje vrednosti primitivnih objektov mogoče samo pri t.i. levih vrednostih, torej takšnih, ki lahko stojijo na levi strani izrazov.

Z objektnim modelom jezika je pogojeno tudi konceptualno modeliranje. Tukaj gre izpostaviti predvsem koncepta abstrakcije in specializacije. Slednja ima svoje korenine v dedovanju, zato je le-to eden izmed najpomembnejših konceptov jezika.

Čistost objektnega modela omogoča veliko izrazno moč jezika, saj sta z njo unificirana tako dostop kot struktura. Koncept "vse je objekt" je nadvse dobrodošel, saj lahko celotno okolje programa preslikamo v množico pasivnih ali aktivnih (med katerimi obstaja interakcija) objektov. Ker tudi krmilne strukture predstavimo z objekti, jih lahko med seboj poljubno povezujemo in sestavljamo skoraj poljubne uporabniško definirane strukture.

3.3 Sistem tipov

Jezik Z_0 je statično tipiziran programski jezik, kar pomeni, da se preverjanje tipov vrši v času prevajanja. Ker gre za objektno usmerjen programski jezik, je potrebno nekatere pretvorbe preverjati tudi v času izvajanja. Jezik Z_0 je močno tipiziran, saj ima vsaka programska fraza jezika predpisan tip. Za statičen sistem tipov se odločamo, ker ta omogoča večjo učinkovitost, kar zadeva izvajanje. Dinamičen sistem bi sicer bil fleksibilnejši in izrazno močnejši, vendar tudi počasnejši. Ker je jezik preveden in ne interpretiran, kot večina dinamičnih jezikov, je takšna odločitev smotrna.

Sistem tipov predstavlja bistvo jezika, saj na nek način povezuje vse pomembne koncepte jezika. Dinamičen sistem tipov je najmočnejša oblika tipiziranja, ki si jo lahko v močno tipiziranem jeziku privoščimo. Tak sistem ima zelo veliko izrazno moč. Drugače je s statičnim tipiziranjem. Ta velikokrat pomeni določeno stopnjo omejitev, kar postane še posebej očitno, ko tipov ne moremo natančno določiti v času prevajanja.

Zaradi tega jezik Z_0 vključuje več različnih mehanizmov tipiziranja.
Implementirano je klasično dinamično tipiziranje, ki se pojavi kot stranski učinek specializacije ali razširjanja funkcionalnosti z uporabo dedovanja. Povsod, kjer v času prevajanja ni mogoče natančno določiti dejanskega tipa, je treba vstaviti dinamično preverbo tipa. Preverbo generira prevajalnik in se izvede implicitno.
Sistem tipov je bilo treba umestiti v statično tipiziranje. Iz tega razloga smo v jeziku Z_0 implementirali posebne konstrukte. Eden izmed njih je tip `Self`, ki predstavlja tip vrednosti `self`. Tip `Self` je povsem dinamičen tip, s katerim je določena dejanska instanca razreda. Tip `Self` omogoča varno tipiziranje izrazov, ki se nanašajo na dejansko instanco in ne na razred, katerega predpostavljamo. Tako se izognemo nepotrebnim in večkrat neelegantnim pretvorbam tipov, ki se pogosto pojavljajo v skoraj vseh sodobnih statično tipiziranih jezikih. `Self` osnovnim razredom hierarhije daje možnost referenciranja objektov izpeljanih razredov. Koncept je bil implementiran v jezikih Eiffel [70], Polytoil [20] in Sather [94, 72].
Velika večina modernih objektno usmerjenih jezikov ima invariantne sisteme tipov. To pomeni, da je signaturo metod pri njihovi redefiniciji potrebno ohraniti. Včasih je to precejšnja omejitev, saj je narava koncepta specializacije navadno takšna, da specializirana metoda operira nad tipi, ki so specializacija tipov metode v starševskem razredu. V poglavju o funkcijskem podtipu smo izpeljali relacijo podtipa z namenom ugotoviti kompatibilno signaturo podmetode. Najmočnejši statični sistem tipov je takšen, ki omogoča kovariantne spremembe v tipu vračanja metode in kontravariantne spremembne v argumentih. Ker smo implementirali tip `Self`, bomo dosegli omejene kovariantne spremembe tudi v tipih argumentov. Tip `Self` lahko namreč stoji na mestu vračanja ali argumenta. Jezik Z_0 ima sistem tipov s tremi preprostimi pravili: kovarianca zaloge vrednosti, kontravarianca domene in invarianca tipa `Self`.

3.3.1 Primitivni tipi

Primitivni tipi jezika Z_0 so tisti tipi, ki so v jezik vgrajeni. Jezik Z_0 ima implementirane vse dobro znane primitivne tipe, ki jih najdemo v skoraj vseh imperativnih programskih jezikih. Celoštevilčni tipi so:

- `Integer` za 32-bitna predznačena cela števila,
- `Short` za 16-bitna predznačena cela števila in
- `Byte` za 8-bitna predznačena cela števila.

Jezik Z_0 ima dva tipa za opis realne vrednosti:

- `Double`, ki predstavlja 64-bitna realna števila in
- `Float`, s katerim predstavimo 32-bitna realna števila.

Predstavitev realnih števil je realizirana po standardu enojne in dvojne natančnosti IEEE 754. Ob številskih tipih Z_0 ponuja še dva pogosto uporabljena znakovna tipa:

- `String`, ki predstavlja znakovni niz poljubne dolžine in
- `Character` za posamezne znake.

Tip, ki zavzame logični vrednosti 0 in 1, se imenuje `Boolean`. Uporablja se v logičnih operacijah, katerih vrednosti temeljijo na vrednostih `true` in `false`.
Vsak primitivni tip ima definirane operacije, ki so primerne in dovolj pogoste, da se jih splača implementirati kot instrukcije virtualnega stroja. Operacije številskih tipov so torej osnovne računske operacije, kot so seštevanje, odštevanje, množenje in deljenje, ter pretvorbene operacije za pretvorbe med posameznimi primitivnimi tipi. Tip `String` definira pretvorbene operacije in operacije za združevanje ter iskanje po nizu.

Osnovni tip vseh ostalih tipov v jeziku Z_0 je `Object`, ki predstavlja najvišji razred v hierarhiji. Razred `Object` ni osnovni razred samo uporabniško definiranim razredom, temveč tudi primitivnim. Uniformna klasifikacija primitivnih in kompleksnih tipov pomeni preprosto aplikacijo mehanizmov vključitvenega in parametričnega polimorfizma. Slednji predstavlja precejšen problem v jeziku C++, saj primitivni in kompleksni tipi niso unificirani kot objekt. Princip čiste objektizacije omogoča široko uporabo parametričnega polimorfizma, skupen nadrazred pa pomeni "poceni" pridobljen vključitveni polimorfizem.
Ker bi bilo ustvarjanje objekta primitivnega tipa z operatorjem `new` neugodno, se le-ta

ustvarja implicitno ob deklaraciji spremenljivke in prirejanju nove vrednosti. Jezik Java ima za vsak vgrajen primitivni tip definiran tudi razred za ta tip, čeprav primitivni tipi niso realizirani kot objekti. Vsak objekt primitivnega tipa v Javi je konstanten, ker se po instanciranju njegova vrednost ne more spreminjati. Nasprotno pa vrednost primitivnega tipa v jeziku Z_0 ni konstanta, saj se lahko med izvajanjem poljubno spreminja, ne da bi se v ta namen moral ustvariti nov objekt. V jeziku, v katerem so tudi primitivne vrednosti predstavljene kot objekti, je to velika prednost, saj je operacija instanciranja razreda časovno zahtevna. Še posebej se to izkaže pri primitivnih tipih, saj je frekvenca uporabe teh zelo visoka.
Primitivni tipi so referenčni tipi. To pomeni, da se njihova vrednost prenaša enako kot vrednost kompleksnega, uporabniško definiranega tipa. Dostop do dejanske vrednosti primitivnega objekta se zaradi učinkovitosti ne vrši preko metode, temveč direktno. Vrednost je shranjena za tabelo metod in meta podatki znotraj objekta. Poudarimo, da z "direktnim dostopom" do vrednosti primitivnega tipa ne mislimo klasičen dostop oblike "`stevilo.vrednost`". Direkten dostop se vrši znotraj virtualnih instrukcij.

Ker je vsaka primitivna vrednost predstavljena kot objekt, so tudi operacije nad primitivnimi vrednostmi samo metode. To pomeni, da izvedba operacije zahteva invokacijo objektove metode. Iz izkušenj vemo, da je invokacija precej zahtevna operacija, saj terja vpogled v virtualno tabelo, kalkulacijo naslova in nazadnje prenos izvajanja na izbrano metodo. Izvedba na videz preprostih operacij nad primitivnimi vrednostmi se na ta način precej zakomplicira. Tudi najpreprostejše operacije, kot so seštevanje, odštevanje in množenje, se prevedejo v eksplicitne invokacije pripadajočih metod. Da bi to očitno neučinkovitost premostili, jezik Z_0 operacij nad primitivnimi tipi ne implementira kot invokacijo, temveč z virtualnimi instrukcijami. Vsaka operacija, definirana nad primitivnim tipom ima svoj ekvivalent v instrukciji virtualnega stroja. Prednost takšne implementacije je očitna, saj je izvedba ene, četudi virtualne, instrukcije neprimerno hitrejša kot invokacija metode, ki mora realizirati identično funkcionalnost. Ker jezik omogoča dedovanje in pozno povezovanje, v času prevajanja ni vedno mogoče enoumno določiti, katera metoda se bo v času izvajanja dejansko izvedla. S tem nastane težava, saj operacij nad primitivnimi tipi ne moremo preprosto prevesti v virtualne

instrukcije, saj bi v času izvanja lahko prišlo do nekonsistence. Do te pride, v kolikor bi želeli neko operacijo, ki je definirana nad primitivnim tipom, prevesti v virtualno instrukcijo ob pogoju, da razred primitivnega tipa ni zaključen. To pomeni, da lahko ta razred dedujemo in razširimo njegovo funkcionalnost v podrazredu. Na mestu, kjer pričakujemo operacijo primitivnega tipa, v času prevajanja ne moremo določiti virtualne instrukcije, saj bo v času izvajanja na tem mestu lahko vrednost osnovnega ali izpeljanega tipa. Pravilo subsumpcije je namreč eno izmed temeljnih in ga moramo nujno ohraniti. Takšno operacijo je potrebno prevesti v invokacijo metode. Najpreprostejša rešitev, v okviru katere bi ohranili čisti objektni model in učinkovitost izvajanja, je ta, da razrede primitivni tipov preprosto zapečatimo. Preprečimo dedovanje na način, pri katerem bi lahko razred primitivnega tipa bil osnovni razred nekega drugega razreda. Nezmožnost dedovanja iz razreda primitivnega tipa se morda zdi omejujoča, vendar omogoča precej preprostejšo in predvsem učinkovitejšo realizacijo funkcionalnosti primitivnega tipa. Poseben primer je tip `Object`, ki pravtako spada med primitivne tipe, vendar lahko služi kot osnova za dedovanje. Izvedba metode nad objektom tega tipa bo zato vedno prevedena v virtualno invokacijo, saj bo dejanski `Object` skoraj vedno objekt nekega podtipa.

3.4 Struktura vhodnih programov

Zapišimo sedaj razred `Person` v jeziku Z_0. Razred opisuje objekt, ki vsebuje podatke o imenu in priimku osebe:

```
class Person {
  // vrne prazen niz
  restricted getFirstName : String {  return ‘‘’’; }
  restricted getLastName : String {  return ‘‘’’; }
  // celo ime = ime + priimek
  restricted getFullName : String {
    return getFirstName() + ‘‘ ’’ + getLastName() ;
  }
  restricted setName(String first, String last) {
```

```
    // metodi vraata nastavljeno ime in priimek
    getFirstName() <-  method : String {  return first; };
    getLastName() <-  method : String {  return last; };
   }
   // ustvarimo kopijo objekta
   restricted clone : Self {
     Self me = new Person;
     me.setName( getFirstName(), getLastName() );
     return me;
   }
 }
  restricted main {
     Person oseba = new Person;
     oseba.setName( ``Viljem'', ``Ockham'' );
     (``Oseba je '' + oseba.getFullName()).print();
   }
 }
```

Po dogovoru je metoda `main` glavna metoda razreda. Modifikator dostopa metode lahko zavzame 4 vrednosti:

- `public`, ki omogoča polni dostop do metode, kar pomeni invokacijo in spreminjanje metode,
- `restricted`, ki pravtako omogoča invokacijo metode, vendar ne njenega spreminjanja,
- `protected`, ki omejuje dostop do metode zunaj razreda vendar ohranja njeno vidljivost v podrazredih in
- `private`, ki omogoča invokacijo in spreminjanje metod samo znotraj razreda.

Metode, ki ne prejmejo argumentov, nimajo deklariranih formalnih parametrov. Oklepaje v tem primeru izpustimo in tako pridobimo na čistosti in zgoščenosti zapisa glave metode. Funkcije, ki vračajo vrednost, imajo tip vračanja naveden za znakom ':'.

Glavni del metode predstavlja njen blok, v katerem so navedene deklaracije lokalnih spremenljivk in stavki.
Iz definicije razreda `Person` vidimo, da "manjkajo" instančne spremenljivke, ki bi v času izvajanja opisovale stanje objekta. Instančne spremenljivke seveda ne "manjkajo", saj je stanje objekta v jeziku Z_0 predstavljeno izključno z instančnimi metodami. V klasični notaciji objektno usmerjenega programiranja bi stanje našega objekta `Person` najpreprosteje opisali z dvema instančnima spremenljivkama – z imenom ter priimkom. Ker v jeziku Z_0 takšnih spremenljivk ne poznamo, v ta namen uporabimo metodi `getFirstName()` in `getLastName()`. Ekskluzivna predstavitev z metodami nas napelje k temu, da na objekt `Person` gledamo kot na entiteto, o kateri lahko poizvemo o imenu in priimku. Kako je dejansko predstavljeno stanje objekta, nas pri "abstraktnem razmišljanju" ne zanima. Spreminjanje stanja objekta, v katerem ni instančnih spremenljivk, je realizirano s spreminjanjem instančnih metod.

Z_0 za spreminjanje metode uporablja poseben operator `<-`. Ta binarni operator na levi strani prejme metodo, ki bo spremenjena in na desni njeno novo definicijo. V gornjem primeru je nova definicija obeh metod realizirana v obliki anonimnih metod. Kar zadeva tipiziranje, je operator `<-` "varen", saj zahteva, da sta signaturi spreminjajoče metode in njene nove definicije identični. S tem se izognemo preverbi tipa v času izvajanja.

Omenimo še metodo `clone`, ki ustvari kopijo objekta. Metoda se nahaja v razredu `Object` in je tukaj redefinirana. Vrača objekt specialnega tipa `Self`. V našem primeru je to tip `Person`, vendar, če bi metodo klicali iz konteksta objekta `Object`, bi vračala tip `Object`. Tip `Self` tako povečuje izrazno moč tipiziranja in je preverljiv v času prevajanja.

3.5 Prvorazrednost jezikovnih konstruktov

Osnova vseh krmilnih konstruktov je zaprtje. Najprimitivnejši konstrukt, ki izhaja iz zaprtja, je blok. Blok je jezikovni konstrukt, v katerem zapišemo programski kod. Obstajata dva tipa blokov. Blok metode definira celotno metodo, lokalni blok pa se

vedno nahaja v bloku metode. Lokalni blok je anonimen, medtem ko je blok metode poimenovan.

Vsak blok ima svoje deklaracije, ki učinkujejo samo v njegovem dosegu. Predstavitev blokov kot prvorazrednih vrednosti pomeni, da jih lahko manipuliramo enako kot spremenljivke ali literale. Blok se lahko izvede na mestu, kjer je deklariran ali kasneje. Zaradi slednjega mora blok vsebovati okolje referenc, s katerimi operira. Blok lahko namreč ubeži svojemu lokalnemu okolju. Lokalne spremenljivke starševskega okolja v času izvedbe bloka nujno še ne obstajajo. Metoda, iz katere je blok prišel kot vrednost, se je namreč lahko že končala. Starševske spremenljivke, ki so v bloku referencirane, nimajo več svoje lokacije v okvirju. Zaradi optimizacije ima blok samo spremenljivke, ki jih dejansko referencira namesto celotnih starševskih okolij. Navedimo primer bloka kot prvorazredne vrednosti:

```
....
Student student(‘‘Marko Prijazen’’);
....
Block b = {
  ....
  student.getPriimek().print();
  ....
};
....
student.setPriimek(‘‘Neprijazen’’);
b.exec();
```

Če definiramo metodo `izvedi`, ki prejme argument tipa `Closure`, ji lahko kot argument podamo spremenljivko tipa `Block`, saj velja `Block` <: `Closure`:

```
izvedi( b );
```

Resnična uporabnost prvorazrednosti blokov pride do izraza pri anonimnih blokih, ki jih lahko podamo neposredno kot jezikovni konstrukt, brez shranjevanja v spremenljivko:

```
izvedi( { student.getPriimek().print(); } );
```

Zaprtje je dinamično v smislu, da vključuje dinamično spremenljivo okolje. Okolje vsebuje originalne reference na vse v bloku uporabljene spremenljivke. Tako kod bloka

vedno operira nad trenutnimi vrednostmi spremenljivk. V gornjem primeru bo priimek, ki se bo znotraj bloka izpisal "Marko Neprijazen" in ne "Marko Prijazen", kot je bila začetna vrednost, ki jo je metoda `getPriimek` vrnila. Na ta način je dosežena konsistenca med vrednostmi starševskih spremenljivk v blokih, ki se lahko nahajajo kjerkoli v pomnilniškem prostoru.

3.6 Iterativni stavki

Iterativni stavki jezika Z_0 so tisti stavki, ki omogočajo iteracijo. Med te prištevamo zanke `While`, `DoWhile` in `For`. Vsaka zanka je sestavljena vsaj iz enega pogoja in bloka, ki se ob tem pogoju izvrši. Ker je zanka prvorazredna vrednost, ima identične značilnosti kot blok.
Modularna predstavitev iteracijskih in vejitvenih konstruktov ima širši pomen. Čista objekta predstavitev omogoča enolično manipulacijo, modularnost pa boljšo razdrobljenost objektov v času izvajanja. Podrobne razlage in primere iterativnih stavkov bomo na tem mestu izpustili, saj to najdemo v referenčnem delu jezika Z_0 [89].

3.7 Manipulacija metod

Ena izmed markantnih lastnosti jezika Z_0 je gotovo predstavitev objektovega stanja. Kot smo dejali, razred v Z_0 nima instančnih spremenljivk in prav to dejstvo postavlja jezik blizu prototipnemu razmišljanju. Spremenljivke so podane samo znotraj metod in so tako vezane na posamezno metodo. Ker se metode od instance do instance razlikujejo, govorimo o instančnih metodah. Stanje objekta se odraža izključno v metodah. Tak pristop je abstraktnejši od tistega, kjer je stanje določeno eksplicitno z definicijo instančnih spremenljivk. Zaradi abstraktnejšega pristopa definiranja objektovega stanja, jezik Z_0 potrebuje ustrezen mehanizem za manipulacijo stanja. Mehanizem, ki to omogoča je dinamično spreminjanje metod [5, 3]. Vsaki metodi razreda lahko, pod določenimi pogoji, spremenimo funkcionalnost. Ker morajo spremembe biti statično varne, nastanejo omejitve pri signaturi metode. Ta mora ostati nespremenjena. Pravzaprav to ni resna omejitev, če pomislimo, da se lahko obnašanje metode bistveno spremeni tudi ob zahtevi ohranitve signature. Metodo spremenimo tako, da navedemo njeno ime in novo definicijo.

Sprememba metode je vidna pri naslednji invokaciji. Možnost dinamičnega spreminjanja funkcionalnosti metode ima veliko prednosti pred ekvivalentno logiko znotraj metode. Logika znotraj telesa metode se izvede ob vsakem klicu, medtem ko se, če metodo spremenimo v celoti, izvajanju dodatne funkcionalnosti izognemo. Prikažimo to na primeru ovrednotenja vozlišča drevesa, ki predstavlja binarno operacijo. Konstruktor razreda `ParseTree` prejme dve celoštevilični vrednosti in operacijo vozlišča. Metoda `evaluate` ovrednoti vozlišče glede na operacijo. Razred je definiran na sledeč način:

```
class ParseTree {
    public ParseTree(Integer a, Integer b, Character op)
    {
       if{ op == '-'; } then {
          evaluate() <- method : Integer {
                              return a - b;
                           }
       } else if{ op == '+'; } then {
          evaluate() <- method : Integer {
                              return a + b;
                           }
       } else if{ op == '*'; } then {
          evaluate() <- method : Integer {
                              return a * b;
                           }
       } else {
          evaluate() <- method : Integer {
                              return a / b;
                           }
       };
    }
    public evaluate : Integer
    {
        return 0;
    }
```

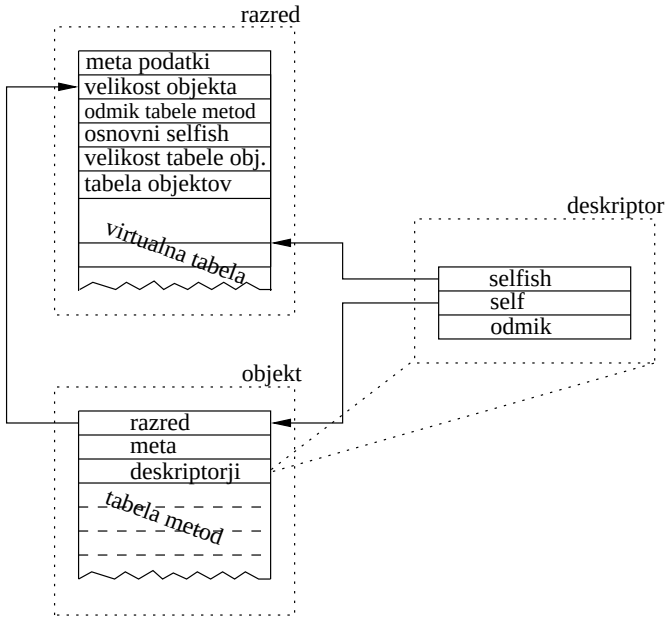
```
}
```

Metoda `evaluate` se definira zgolj enkrat – v konstruktorju, kjer opravimo potrebne primerjave. Ne glede na to, kolikokrat jo pokličemo, se bo vedno izvedla samo specializirana verzija metode. Na ta način se bistveno pohitri izvedba celotne metode. Metoda `evaluate` je preprosta, toda isti princip lahko posplošimo na mnogo kompleksnejše metode.

Na spreminjanje metod lahko gledamo kot na omejeno različico dinamičnega dedovanja. Dinamično dedovanje omogoča, da v času izvajanja spremenimo nadrazred. Sprememba nadrazreda efektivno pomeni spremembo metod definiranih v nadrazredu. V jeziku Z_0 lahko metodo nadrazreda spremenimo, če le zagotovimo ohranitev originalne signature. Tak pristop rezultira v sicer omejenem mehanizmu dinamičnega dedovanja, vendar je ta statično varen.

3.8 Dedovanje

Jezik Z_0 uporablja predstavitev objekta, ki je bližja prototipnim jezikom. Vzrok temu so objekti, katerih stanje se opiše izključno z metodami. Predstavitev objekta je pomembna s stališča učinkovitosti in možnosti manipulacije objektovega stanja. Navezo objekta in razreda v jeziku Z_0 prikazuje slika 2.
Objekt v jeziku Z_0 vsebuje kazalec na razredne podatke in kazalec na pripadajoči metarazred. Razredni podatki se nahajajo v "razrednem prostoru" in se uporabljajo pri instanciranju razreda in manipulaciji deskriptorja. Kazalec na metarazred se uporablja za učinkovitejšo implementacijo metaprogramiranja, saj omogoča, da se metarazred za isti objekt ustvari samo enkrat. Kazalec na metarazred je bil v jeziku Z_0 neuporabljen, izrablja ga šele jezik Zero. Ker so metode vezane na instanco razreda, je tabela metod shranjena v objektu. Posamezen vstop v tabelo metod vsebuje pomnilniški naslov metode, kazalec *self*, kazalec na okolje in kazalec na zaprtje. Tabela metod vsebuje vstope za vsako metodo v razredu.
Deskriptor ali opisnik objekta je trojček, ki vsebuje kazalec na objekt (*self*), kazalec na virtualno tabelo trenutnega objekta (v jeziku Z_0 ga imenujemo *selfish*) in odmik na



Slika 2: Struktura objekta.

razred znotraj hierarhije. Slednji je potreben zaradi večkratnega dedovanja.

V definiciji razreda nadrazred navedemo z rezervirano besedo `inherits`. Zaradi uniformne razredne hierarhije jezika Z_0 vsak razred implicitno deduje razred `Object`. V hierarhiji z večimi nivoji se razred `Object` ne vključi v seznam neposrednih predhodnikov. V tem primeru se `Object` vključi kot nadrazred najvišjega uporabniškega razreda. Dedovanje je bilo v jeziku Z_0 zasnovano tako, da bi bilo resnično preprosto za uporabo in razumevanje. Iz tega razloga ima Z_0 izjemno preprosta pravila in modifikatorje. Dedovanje je vedno "public" in "nevirtualno", kar pomeni, da gre za model dedovanja z replikacijo. Pravila dedovanja se najbolj odražajo pri redefiniranju metod. Na redefiniranje lahko gledamo kot na mehanizem dedovanja, ki omogoča koncept specializacije. Specializacija podaja podrobnejšo in bolj dodelano funkcionalnost, kot je bila na voljo v starševskem razredu. Specializacija, kot temeljni koncept dedovanja, se najbolj izraža v metodah. Ker v našem jeziku nimamo instančnih spremenljivk, je to še toliko bolj res. Specializacijo izvajamo nad metodami razreda. Cilj je zagotoviti kompatibilnost signature metod, saj se lahko le tako zanesemo na statično varnost. Jezik Z_0 metodo tretira kot redefinirano, če ima enako ime in enako število parametrov. Ti

pravili služita kot osnova za redefiniranje metod. Če katerikoli izmed pogojev ne velja, metoda ne velja za redefinirano. V primeru, da je metoda redefinirana, morata veljati še dve pravili: tip vračanja mora biti identičen ali pa se spreminja kovariantno in tipi formalnih parametrov morajo biti identični ali pa se spreminjajo kontravariantno. S temi pravili je omogočena najvišja stopnja sprememb, ki je še statično preverljiva. Poseben primer je tip `Self`, ki je invarianten. To pomeni, da mora ostati enak. Invarianca tipa `Self` je nenavadna, saj se ta tip spreminja implicitno in tako dejansko nikoli ni enak. V primeru, ko katerikoli izmed pogojev redefinirane metode ne velja, prevajalnik javi napako.

Do dvoumnosti lahko pride že pri enkratnem dedovanju. Razred lahko iz povsem praktičnih razlogov definira več metod z istim imenom in enakim številom parametrov. Takšna metoda je polimorfna. Na mestu invokacije prevajalnik na podlagi tipov argumentov izbere najprimernejšo različico metode. V kolikor ustrezna metoda ni najdena, se pokliče naslednja najbolj ustrezna metoda. Če je teh več, se javi napaka. Pri redefiniranju metode v podrazredu je potrebna pazljivost, saj metode vedno ni mogoče redefinirati.

3.9 Dinamično tipiziranje s `Self`

Izrazna moč statično tipiziranih programskih jezikov je v veliki meri omejena s sistemom tipov. Večina statično tipiziranih jezikov ima invariantne sisteme tipov. Tipi instančnih spremenljivk in signature metod se po hierarhiji ne smejo spreminjati. Kovariantne spremembe v tipih vračanja so sicer dobrodošle, vendar vedno ne zadoščajo potrebam. V praksi so kovariantne spremembe velikokrat potrebne tudi v tipih argumentov in instančnih spremenljivk. Eiffel takšne spremembe omogoča, vendar niso statično varne. Instančne spremenljivke za nas niso posebno pomembne, saj jih jezik Z_0 ne omogoča.

Implicitna metaspremenljivka `self` ima identično vlogo kot `this` v Javi ali C++. Referenca `self` je na voljo v telesu vsake nestatične metode. Kaže na objekt, nad katerim se metoda izvaja. S stališča tipiziranja je pomemben tip te reference. Spremenljivka `self` ima enak tip kot objekti, ki se ustvarijo iz razreda, v katerem se na `self` sklicu-

jemo. To pomeni, da ta tip ni enak, temveč se spreminja po hierarhiji.

Primer praktične uporabe tipa `Self` kaže definicija metode `clone` v prejle zapisanem razredu `Person`. Metoda `clone` ustvari kopijo objekta `Person` in jo vrne. Tip vračanja te metode je očitno enak razredu, v katerem je metoda definirana. Metoda `clone` je sprva deklarirana v razredu `Object`, kjer vrača vrednost tipa `Self`, tj. `Object`. Tip `Self` razširja izrazno moč tipiziranja tako, da ni potrebna eksplicitna pretvorba tipa. Tip `Self` tako omogoča še bolj dinamično tipiziranje kot kovariantne spremembe v tipih vračanja. Zelo dobra lastnost tega tipa je, da je statično varen in preverljiv v času prevajanja.

4 Metaprogramiranje in refleksija

Kaj je pravzaprav metaprogramiranje? Dobesedna analiza besede nam razloži, da je to nekaj "nad" ali "za" programiranjem. Širše uveljavljena definicija pa pravi, da je to programiranje, ki je tudi samo sposobno programiranja. Sem torej spadajo programi, ki "pišejo" druge programe ter "programirajo" sami sebe. Izkušnja s programskimi jeziki nas takoj pouči, da programa, ki bi dejansko "programiral" samega sebe, brez iniciative s strani človeškega programerja, zaenkrat ni. Avtonomnost takšne stopnje je domena umetne inteligence, ki pa v času tega pisanja ne dosega ravni, ki bi takšno avtonomnost omogočala. "Veliko" idejo metaprogramiranja tako nekoliko zreduciramo, da postane manj univerzalna, vendar bolj uporabna. Pod pojmom metaprogramiranja v kontekstu programskih jezikov danes razumemo programiranje, ki posega v globlje nivoje programa in interpreterja. Metaprogramiranje interpreterja, ki izvaja ukaze nekega programa, bi pomenilo vplivati na izvajanje teh ukazov. To bi bilo mogoče, če bi interpreter dovoljeval redefiniranje instrukcij, s katerimi izvršuje ukaze programa. Metaprogramiranje lahko ostane tudi na višjem nivoju. Metaprogramiranje bi npr. pomenilo spremeniti shemo dedovanja razredov v objektno usmerjenem programskem jeziku. Na ta način bi lahko določili, kje naj se metoda, ki jo kličemo, prične iskati - ali v zadnjem razredu ali v korenskem razredu hierarhije. Hitro lahko uvidimo, da imajo takšne sposobnosti daljnosežne posledice.

Jezik, v katerem je metaprogram zapisan, imenujemo metajezik. Takšni jeziki so torej vsi tisti, ki omogočajo pisanje metaprogramov. Navadno je jezik, v katerem je program zapisan, hkrati tudi metajezik, saj omogoča metaprogramiranje programov, zapisanih v tem istem jeziku. V tem primeru govorimo o refleksiji, saj ima program možnost vpogleda "vase" in manipulacije lastnega izvajanja. To je v grobem mogoče doseči na dva načina – z manipulacijo obnašanja ter manipulacijo strukture. Iz tega izvirata dva tipa refleksije, katerima pravimo, skladno z njuno nalogo – behavioralna ter strukturalna refleksija. Prav ta dva tipa refleksije sta za nas najpomembnejša, saj se z njima posebej ukvarjamo v našem delu.

Zaradi tesne povezave z interpretiranimi jeziki, kot so Smalltalk [62], 3-Lisp [36], Obj-VLisp [31] in CLOS [64], je metaprogramiranje tesno povezano z introspektivnimi

interpreterji, torej takšnimi, ki omogočajo spreminjanje samih sebe. Metacirkularni interpreter je zapisan v jeziku, katerega interpretira. Program, katerega tak interpreter interpretira, lahko dostopa in spreminja stanje in strukture tega interpreterja [54]. Postopek, ki posamezno komponento interpreterja izpostavi kot vrednost, katero je mogoče manipulirati s strani programa, imenujemo reifikacija [90]. Refleksija gre torej ravno v obratno smer kot reifikacija, saj je njena naloga, da z manipulacijo te izpostavljene vrednosti spremeni interpreter.

Metaprogramiranje pa ni nujno povezano izključno z interpretiranimi jeziki, katerih izvorni kod je na voljo v času interpretiranja. Novejše raziskave so pokazale, da je metaprogramski model primeren tudi za prevedene jezike. Eden izmed pristopov, ki realizirajo to idejo, je metaobjektni protokol [56, 68], ki je izšel iz objektno usmerjenih programskih jezikov. Metaobjektni protokol pomeni razširitev paradigme objektne usmerjenosti s tem, da ima vsak objekt svoj pripadajoči metaobjekt. Metaobjekt je instanca metarazreda. Funkcionalnost metod nekega metarazreda omogoča manipulacijo objekta, "določenega" z metaobjektom tega metarazreda. Gre torej za par objekt–metaobjekt, kjer metaobjekt omogoča manipulacijo svojega pripadajočega objekta. Sem spada, kot smo omenili že zgoraj, shema dedovanja, ki npr. iz enkratnega dedovanja napravi večkratno, invokacija metod, ustvarjanje objekta. Od "moči" metaobjekta je odvisno, kaj lahko z objektom počnemo. Metaobjektni protokol določa stopnjo metaprogramiranja jezika. Metaobjektni protokol se je izkazal kot primeren pristop k refleksiji statično tipiziranih prevedenih jezikov kot sta C++ [28] in Java [101]. Seveda gre pri omenjenih dveh jezikih za metaobjektni protokol v času prevajanja. Očitna slabost takega protokola je nezmožnost sprememb v času izvajanja. Poskus metaobjektnega protokola, ki temelji na "šablonskem" metaprogramiranju in obravnava refleksijo v jeziku C++ v času prevajanja in izvajanja, je bil podan v [11].

Pri večini programskih jezikov, ki omogočajo metaprogramiranje, je le-to usmerjeno v ta isti programski jezik. Gre torej za metaprogramiranje programov zapisanih v istem jeziku kot metafunkcionalnost. Metaprogramski model, ki omogoča prehod na druge, "zunanje" jezike, je bil predlagan v [52]. Programsko ogrodje s podobnim ciljem – omogočiti adaptacijo poljubne aplikacije v času izvajanja, realizira nitrO [76].

Metaprogramiranje ni postranski koncept, nasprotno, je eden izmed konceptov, ki naj-

bolj vplivajo na jezik kot celoto. Jezik lahko napravi izrazno močnejši, a ta lahko iz istega razloga postane tudi kompleksnejši za razumevanje. Pri načrtovanju metaprogramskega modela jezika sta zato na mestu največja previdnost in preudarnost. Nekaj napotkov in zahtev, ki so se izkristalizirali skozi zgodovino metaprogramiranja in ki bi jih sleherni metaprogramski model naj upošteval, je zgoščeno podanih v [91].

4.1 Refleksija

Ker je refleksija temeljni koncept metaprogramiranja v jeziku Zero, si jo poglejmo podrobneje. Kadar govorimo o refleksiji, je treba povedati ali gre za t.i. "statično" refleksijo, ki omogoča samo vpogled programa vase ali za refleksijo, ki ob vpogledu omogoča tudi spreminjanje, t.j. dinamično refleksijo. Statična refleksija pomeni samo prenos določenih informacij iz časa prevajanja v čas izvajanja. Ta vrsta refleksije ni problematična za implementacijo, saj gre zgolj za "branje" podatkov. Popularnost te vrste refleksije je zrasla z Javo, ki je svoj refleksijski model zasnovala prav na možnosti dostopa do informacij o razredih, metodah in tipih. Za nas je zanimivejša dinamična refleksija, saj se prav te lotevamo z nadgradnjo jezika Z_0.
Refleksijo navadno kategoriziramo v behavioralno in strukturalno ali lingvistično [59]. Behavioralna refleksija omogoča manipulacijo obnašanja, strukturalna manipulacijo strukture – torej vpogleda vanjo in spreminjanja. Metaprogramski model jezika Zero temelji na obeh vrstah refleksije.

4.1.1 Behavioralna refleksija

Behavioralna refleksija je navadno realizirana kot "dodana funkcionalnost" posameznim jezikovnim konstruktom. Za kaj pravzaprav gre? Najlaže si primer manipulacije obnašanja predočimo z metodo. Metoda je jezikovni konstrukt, ki realizira neko funkcionalnost. Obnašanje te metode lahko spremenimo tako, da ji dodamo funkcionalnost, ki v zasnovi ni bila načrtovana. S spremenjeno funkcionalnostjo metode se spremeni njeno obnašanje. Mehanizem, ki omogoča dodajanje funkcionalnosti, je tesno povezan z aspektno usmerjenim programiranjem [58, 57]. Pri tem gre za funkcionalnost v obliki metod ali fragmentov programskega koda, ki se "pripne" najrazličnejšim akcijam, kot

so invokacija metode, ustvarjanje objekta, izvršitev bloka itd. Rečemo lahko, da je aspektno usmerjeno programiranje po svojem bistvu le jezikovna konceptualizacija behavioralne refleksije.

Izrazna moč behavioralne refleksije temelji na njenem modelu, ki določa, kaj vse je mogoče spremeniti in na katerih mestih je mogoče te spremembe uveljaviti. Aspektno usmerjeno programiranje namenja veliko pozornosti točkam v programu, kjer je funkcionalnost mogoče razširiti. Čim več je takih točk, tem finejša je možnost manipulacije. Nič manj pomembno ni, kako vplivati na obstoječo funkcionalnost. Ali je originalni funkcionalnosti mogoče samo dodati novo funkcionalnost ali je mogoče originalno funkcionalnost tudi delno ali povsem preprečiti. V slednjem primeru ima behavioralna refleksija bistveno večjo moč, kot v primeru, ko je mogoče funkcionalnost samo nadgrajevati. Veliko takih mehanizmov behavioralne refleksije je bilo realiziranih kot nadgradnja Jave, ki v osnovi te vrste refleksije ne omogoča. Primeri za to so MetaXa [50], Iguana/J [80], Kava [106], ki se poslužuje manipulacije zložnega koda v času nalaganja, in Reflex [99, 45]. Primera refleksije v Javi, ki pravtako temeljita na manipulaciji zložnega koda, sta tudi JOIE [30] in BCEL [33]. Java ni edini jezik, ki je bil deležen številnih nadgradenj, povezanih z refleksijo in metaprogramiranjem. Orodje GEPPETTO [81] razširja jezik Smalltalk s koncepti dinamične behavioralne refleksije.

4.1.2 Strukturalna refleksija

Strukturalna refleksija pomeni možnost vpogleda in spreminjanja strukture programa. Temeljna ideja te vrste refleksije je v možnosti adaptacije in optimizacije programa v času izvajanja. Manipulacija strukture programa je zapletenejša od manipulacije obnašanja z dodajanjem funkcionalnosti. Program navadno sestoji iz krmilnih struktur, kot so bloki, iteracijski stavki, stavki izbire ter preprosti izrazi. Strukturalna refleksija se obrača prav v manipulacijo teh struktur. Hitro postane jasno, da lahko možnost sprememb teh temeljnih struktur pomeni popolno preobrazbo programa. Hkrati z možnostmi adaptacije in optimizacije v času izvajanja, nastopijo tudi težave, povezane z zagotavljanjem varnosti izvajanja. To še posebej velja za statično tipizirane in prevedene jezike, kot je Zero. Spremembe strukture morajo v takem jeziku ohranjati konformanco tipov.

Strukturalna refleksija statično tipiziranih jezikov navadno temelji na eksplicitnem prevajanju v času izvajanja in na tej tehniki temelječemu generiranju koda. Tak pristop, ki je podoben prevajanju "just-in-time" [12], opisuje prispevek [59]. Za kakšen postopek torej gre pri takšnem modelu strukturalne refleksije? V splošnem ga lahko povzamemo takole: izvorni programski kod se v obliki znakovnega niza poda metodi, ki ta kod prevede v izvršnega, navadno kar z invokacijo zunanjega prevajalnika. Tako prevedeni kod se lahko nato uporabi v programu tako, da nadomesti kak fragment ali funkcionalnost celotne metode. Gre torej za generiranje koda v času izvajanja. To nikakor ni revolucionarna tehnika, saj so nekateri programski jeziki isto omogočali že pred leti. Težavni del take manipulacije strukture je v tem, kako jo napraviti preprosto in razumljivo ter kako jo umestiti v programski jezik. Drug problem je "zrnatost" manipulacije, od katere zavisi, do katerega nivoja se lahko manipulacija strukture izvaja. V primeru, da je mogoče na strukturo vplivati zgolj na nivoju metode, je takšna manipulacija zelo omejena. V kasnejših poglavjih bomo pokazali, da se teh zaprek in težavnosti zavedamo tudi pri načrtovanju metaprogramskega modela jezika Zero.

Navkljub očitni praktični uporabnosti, večina statično tipiziranih objektno usmerjenih programskih jezikov ne omogoča uporabne stopnje strukturalne refleksije. Prav iz tega razloga so se, podobno kot to velja za behavioralno refleksijo, pojavile nekatere razširitve obstoječih programskih jezikov s tem konceptom. Tako je strukturalna refleksija v jeziku Java dobila možnost praktične uporabe z razredno knjižnico Javassist [29], ki omogoča dva nivoja manipulacije koda – nivo izvornega koda in nivo prevedenega zložnega koda. Podobno vpeljavo strukturalne refleksije z manipulacijo koda predlagata orodji Jinline [100] za Javo ter ByteSurgeon [35] za jezik Smalltalk. Razširitev Reflective Java [108] predlaga translacijo izvornega koda pred procesom prevajanja, kar pa ima iste omejitve, kot smo jih omenili že v okviru behavioralne refleksije.

4.2 Metaprogramski model

Pod pojmom metaprogramskega modela razumemo skupek jezikovnih metaprogramskih konceptov in relacij med njimi. Med metaprogramske koncepte prištevamo re-

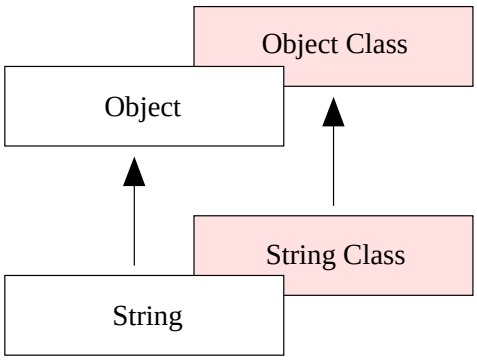
fleksijo, metarazrede, metaobjekte in reifikacijo. Lastnosti teh konceptov vplivajo na metaprogramski model, ga opisujejo, posredno pa vplivajo tudi na celoten jezikovni model. Metaprogramskega modela ne moremo obravnavati ločeno od jezika, saj so njegovi koncepti v tesni navezi z jezikovnimi koncepti, kot so objektni model, sistem tipov, dedovanje idr. Nesmotrno bi npr. bilo statično tipiziran jezik nadgraditi z metaprogramskim modelom, ki bi ne upošteval statičnih omejitev.

Idealna zasnova jezika, ki bi naj omogočal metaprogramiranje, je, da na to možnost mislimo že v samih zametkih jezika. Od temeljnih konceptov jezika bodo odvisni metaprogramski koncepti. "Slaba" zasnova jezika je še posebej vidna pri kasnejših nadgradnjah z metaprogramskim modelom. Pazljivost je torej še posebej na mestu pri jezikih, ki v osnovi ne omogočajo metaprogramiranja, a so kasneje deležni tovrstne razširitve. Tak primer je Java, ki je zaradi svojih omejitev glede tipov, dedovanja in same arhitekturne zasnove virtualnega stroja, povzročila številne razširitve, med katerimi obstaja mnogo podobnih. Kljub temu nobena izmed njih ne poda zadovoljivega metaprogramskega modela. Na morebitne koncepte metaprogramiranja pomislimo torej v samih začetkih načrtovanja programskega jezika.

Metaprogramski model temelji na različnih zasnovah. V razrednem jeziku ga je smiselno zasnovati na osnovi metarazredov; tega pristopa se v jeziku Zero poslužujemo tudi sami. Čisti metaobjektni protokol je primeren tako za razredne kot za prototipne jezike, le da pri slednjih metaobjekt nima pripadajočega metarazreda. Koncept reifikacije, tj. izpostavitve struktur jezika in interpreterja programu, je v takšni ali drugačni obliki prisoten v vseh metaprogramskih modelih, če le omogočajo kaj več kot zgolj vpogled. Metaprogramski modeli variirajo od nadvse preprostih do izjemno kompleksnih. Eden izmed takšnih izjemno preprostih modelov je prisoten v jeziku C++ v obliki predprocesorja, katerega funkcionalnost je na voljo v času prevajanja. Eden izmed poskusov vpeljave metaprogramskega modela na osnovi omejene funckionalnosti predprocesorja jezika C++ je bil podan v [26]. Na drugo stran pa lahko postavimo Smalltalk, ki je že pred desetletji svoj metaprogramski model definiral do mej, katerih večina programskih jezikov še danes ne dosega.

4.3 Metarazredi

Metarazred je temeljni koncept metaprogramiranja v razrednem objektno usmerjenem jeziku. Različni metaprogramski modeli definirajo metarazred na različne načine. V naših raziskavah privzemamo definicijo metarazreda, kot so jo podali snovalci jezika Smalltalk. Ta definicija, ki sicer izvira iz osemdesetih let prejšnjega stoletja, se je ohranila v večini sodobnih metaprogramskih modelov. Najpreprosteje ponazorimo koncept metarazreda z njegovo funkcionalnostjo. Metarazred je repozitorij za informacije o razredu. V tem je metarazred nekaj bolj elementarnega kot razred. V razrednem jeziku postane razred razreda. V Smalltalku razred vsebuje metode instance, metarazred pa metode razreda. Tako imamo dejansko dve razredni hierarhiji – hierarhijo razredov in hierarhijo metarazredov. Če izhajamo iz instance, uvidimo celo tri ravni – instanco, razred ter metarazred. Čeprav v Smalltalku obstaja celotna hierarhija metarazredov, si jo lahko zamišljamo kot hierarhijo, ki se nahaja v ozadju kot neka paralelna hierarhija (slika 3) in do katere dostopamo samo v izjemnih primerih, kadar je npr. potrebno spremeniti obnašanje metod.



Slika 3: Paralelna hierarhija metarazredov in razredov.

Metaprogramski model, ki temelji na metarazredih, ne vpeljuje pripadajočega metarazreda nujno vsakemu razredu. S stališča prostorske in časovne učinkovitosti je celo bolje vpeljati "univerzalen" metarazred, s katerim je mogoče posegati v funkcionalnost vseh razredov. Vendar taka univerzalnost prinaša s seboj tudi izgubo lokalnosti in neko "globalizacijo" funkcionalnosti, ki se razkriva tudi tam, kjer ni potrebna ali sploh smiselna. V okviru načrtovanja metaprogramskega modela jezika Zero privze-

mamo idejo metarazredov, toda brez zapadanja v skrajnosti. Kot bomo pokazali v prihodnjih poglavjih, metaprogramski model našega jezika ne temelji niti na enem niti na neskončnem številu metarazredov, temveč na treh, ki opisujejo metafunkcionalnost razredov, metod in krmilnih konstruktov jezika.

5 Jezik Zero

Načrtovanje programskega jezika je zahteven in kompleksen proces, ki zahteva ekspertizo s področja poznavanja jezikovnih modelov, konceptov in implementacijskih postopkov. Če to velja za proceduralne jezike, se pri objektno usmerjenih, objektnih in funkcijskih jezikih stvar še bolj zakomplicira. Načrtovanje novega jezika je vselej izziv, čeprav je zares "novega" v današnji množici raznovrstnih jezikov težko doseči. Kljub temu obstajajo postopki, pristopi in koncepti, ki lahko imajo radikalen vpliv na jezik. Še posebej se da to doseči z združevanjem konceptov, ki so do nekega trenutka veljali za izključujoče. Pogosto se združujejo koncepti, ki so značilni za določen tip jezikov.
Načrtovanja jezika Zero se lotevamo s ponovno in temeljito analizo obstoječih konceptov jezika Z_0, katerega jemljemo za temelj našega novega jezika. Nadgradnja jezika Z_0 ne zadeva zgolj faze implementacije in prilagajanja, temveč je smotrno začeti pri samem načrtovanju. Tekom te faze smo ugotovili, da nadgradnja jezika Z_0 ne bo samo obogatila z novimi koncepti, ki bodo vplivali na obstoječe, ampak bo nekatere teh konceptov tudi spremenila na način, ki bo ustrezal novi nalogi jezika. V pričujočih podpoglavjih bomo nadgradnjo obstoječih in vpeljavo novih jezikovnih konceptov prikazali s stališč načrtovanja in implementacije. Pri posameznih konceptih bomo preučili omejitve kar zadeva tipiziranje in predlagali morebitne alternativne rešitve.
Jezik Zero predstavlja signifikantno nadgradnjo jezika Z_0. Od slednjega deduje politiko tipiziranja, shemo dedovanja, jezikovne konstrukte, objektni model z možnostjo dinamične manipulacije metod v času izvajanja ter sintaktično strukturo. Večidel konstruktov jezika Z_0 ostaja nespremenjen. Pri jeziku Zero gre predvsem za nove konstrukte in koncepte. Eden teh novih konceptov je gotovo prvorazrednost metod, kateri služi za osnovo metodam višjega reda in anonimnim metodam. Na tem mestu povejmo, da je že jezik Z_0 vključeval omejeno verzijo tega koncepta v obliki prvorazrednih metod, s katerimi se je vršilo nadomeščanje instančnih metod. Koncept prvorazrednosti sedaj razširimo in ga napravimo povsem splošnega. S tem dobijo metode v jeziku Zero resnično prvorazredno karakteristiko, kar pomeni, da z njimi manipuliramo kot s spremenljivkami.

Metode oz. funkcije višjega reda so star in dobro znan koncept funkcijskih jezikov. Zero prikazuje praktično izvedbo tega koncepa v statično tipiziranem imperativnem jeziku. Spričo statičnega sistema tipov uvaja t.i. metodne tipe, ki opisujejo signature metod in jih je mogoče preverjati v času prevajanja. Seveda je tipiziranje tisti koncept, ki jeziku Zero preprečuje, da bi dosegel stopnjo izrazne moči metod višjega reda, ki je dosegljiva funkcijskim dinamično tipiziranim in interpretiranim jezikom. Toda v pričujočih podpoglavjih o metodah višjega reda pokažemo, da se ta koncept, kljub dorečenim omejitvam, lepo vklaplja v jezik imperativne narave. Nasploh je metoda v jeziku Zero zelo temeljni pojem, saj je že v jeziku Z_0 bila edini mehanizem za manipulacijo in odražanje stanja objekta. Jezik Zero razširja idejo koncepta "klasične" metode na anonimne in metode višjega reda. Prvorazrednost metod kot tudi primitivnejših konstruktov je bistvena za metaprogramski model jezika. Večidel refleksijske funkcionalnosti, ki spreminja obnašanje ter strukturo v času izvajanja, temelji prav na predpostavki, da je krmilne konstrukte in metode mogoče manipulirati kot prvorazredne vrednosti. Brez te lastnosti bi metaprogramiranje v jeziku Zero ostalo na precej primitivnejši in okornejši stopnji, kot se nahaja zdaj.

Čisti objektni model in dejstvo, da so tudi primitivni jezikovni konstrukti sami sestavljeni iz objektov, omogočata dinamično manipulacijo strukture in obnašanja. Grobo povedano, gre v osnovi zgolj za manipulacijo referenc ob zahtevi kompatibilnosti tipov. Refleksija jezika Zero je načrtovana na način, ki je združljiv s statičnim sistemom tipov. Dinamika nadomeščanja metod in nadrazredov temelji na kompatibilnosti signatur. Velik del zahtevnosti izvedbe refleksije v jeziku pade na ramena načrtovanja jezika, ki bi smiselno združeval raznorodna koncepta ohranjanja varnosti tipiziranja ter dinamičnega metaprogramiranja.

Metaprogramski model jezika Zero je zasnovan na eksplicitnih in implicitnih metarazredih. Slednji omogočajo neposreden dostop do metafunkcionalnosti krmilnih konstruktov, kar pomeni, da je funkcionalnost metarazredov na razpolago na vsakem koraku. Jezik Zero ne definira "posebnih" namenskih konstruktov za dostop do metafunkcionalnosti, temveč jo programerju izpostavlja na klasičen programski način – z metodami, dedovanimi iz nadrazredov. Metaprogramiranje je v jeziku Zero realizirano s pomočjo refleksije, ki omogoča behavioralno ter strukturalno manipulacijo programa v

času izvajanja. Dinamične spremembe obnašanja in strukture so omejene z zahtevo po ohranjanju konformance tipov, saj gre za tipiziran programski jezik.
Ker jezik Zero temelji na konceptih jezika Z_0, je smiselno, da tudi izvajalno okolje virtualnega stroja ohrani funkcionalnost, ki je bila predvidena za jezik Z_0. Potrebno jo je dograditi z ustrezno podporo novim jezikovnim konceptom. Eden takšnih je nedvomo vmesnik za izvrševanje funkcionalnosti, zapisane v drugem, "zunanjem" programskem jeziku. Metafunkcionalnost metarazredov jezika Zero je zaradi učinkovitosti namreč zapisana v jeziku C++. Drug koncept, ki zahteva učinkovitost, nedosegljivo znotraj interpreterja, je metoda višjega reda. Vsi koncepti, realizirani preko omenjenega vmesnika so za programerja v jeziku Zero povsem transparentni. Da gre za funckionalnost, implementirano v zunanjem jeziku, je vidno zgolj na implementacijskem nivoju.
Sintaksno zasnovo jezika Z_0 spreminjamo le neznatno. Pomembne spremembe se pojavijo pri nadgradnji tipov z metodnimi tipi, pri konstruktu nadomestitve metode ter pri invokaciji metode, ki mora po novem upoštevati še metode višjega reda.

5.1 Razširitev jezika Z_0 na sintaktičnem nivoju

Jezik Zero je sicer nadgradnja obstoječega jezika Z_0 z meta koncepti. Vendar implementacija jezika Zero ne terja samo "sistemskih" sprememb temveč globlji poseg v jezik. Koncepti metaprogramiranja posegajo v temelj jezika in odražajo spremembe na sintaktičnem, semantičnem in nivoju virtualnega stroja. Velik del sintakse jezika Z_0 bomo ohranili, posamezne dele spremenili in določene dodali.

V razredu definirane metode lahko imajo dodatne modifikatorje. Dodali smo modifikatorja statičnih in metod v zunanjih knjižnicah. Sintaksa glave procedur in funkcij se tako spremeni:

```
FunctionDeclaration: MethodAccessModifier AdditionalMethodModifiersOpt
                     IDENTIFIER
                     MethodParametersOpt ':' Type MethodBody

ProcedureDeclaration: MethodAccessModifier AdditionalMethodModifiersOpt
                      IDENTIFIER
```

```
                    MethodParametersOpt MethodBody

AdditionalMethodModifiersOpt:
                              | AdditionalMethodModifiers

AdditionalMethodModifiers:   AdditionalMethodModifier
                           | AdditionalMethodModifiers
                                 AdditionalMethodModifier

AdditionalMethodModifier:    NATIVE
                           | STATIC
```

Razširitev opcijskih modifikatorjev metod je napravljena tako, da lahko v prihodnosti nov modifikator preprosto dodamo v gornjo produkcijo.

Jezik Zero omogoča prvorazredne in anonimne metode. V ta namen potrebujemo konstrukt, s katerim opišemo metodno abstrakcijo. Sledeča produkcija omogoča metodne abstrakcije konstruirane neposredno iz zaprtja ali iz instančnih metod:

```
MethodExpression:          METHOD MethodParametersOpt ReturnTypeOpt Block
                         | METHOD UpdateReference
```

Metode se v jeziku Zero spreminjajo s posebnim operatorjem `<-`:

```
MethodExpression:     UpdateReference '<-' MethodExpression
```

Prvorazredne metode in metode višjega reda zahtevajo metodne tipe. Definicijo tipa dopolnimo na sledeč način:

```
Type:     PredefinedType
        | SELF
        | QualifiedIdentifier
        | METHODTYPE   '(' MethodTypeParameters ')' ReturnTypeOpt
        | METHODTYPE ':' Type
```

Produkcija omogoča definicijo metodnega tipa z argumenti in brez njih.

Potrebujemo še dodatno opcijo za invokacijo metode višjega reda, saj se invokacija lahko veriži do poljubnega nivoja:

```
MethodAccess:   IDENTIFIER '(' ArgumentsOpt ')'
              | MethodAccess '.' IDENTIFIER '(' ArgumentsOpt ')'
              | MethodAccess '(' ArgumentsOpt ')'
```

5.2 Vmesnik za uporabo zunanjega programskega koda

Visokonivojski programski jeziki, pri katerih se izvršni ali objektni kod generira za nek virtualni stroj in pri katerih se izvaja visoka stopnja varnosti izvrševanja, potrebujejo učinkovit mehanizem za izvedbo funkcionalnosti, katere so za programerja tega jezika zelo težavne ali celo nemogoče za implementacijo. Navadno gre za sistemsko pogojene funkcionalnosti, ki so povsem specifične in vezane na arhitekturo operacijskega sistema ali samo strojno opremo. Zaradi lastnosti virtualnega stroja, predvsem zaprtosti in stroge ločenosti od "zunanjega" okolja, v katerem teče operacijski sistem, je uporaba sistemske funkcionalnosti otežena. Da bi znotraj izvajalnega okolja virtualnega stroja bili sposobni poklicati storitev operacijskega sistema, potrebujemo mehanizem za klic splošnih funkcij, katerih implementacija se običajno nahaja v dinamičnih knjižnicah. Implementacija tega ni pretirano zahtevna, saj obstajajo standardni mehanizmi, ki omogočajo dinamičen dostop do funkcionalnosti znotraj knjižnic. Toda potrebno je še več. Zaželjeno je, da je dostop iz okolja virtualnega stroja do zunanjega sveta, kolikor je to mogoče, transparenten. Če se specializiramo na naš jezik, to pomeni, da programer ne ve ali pa mu ni potrebno vedeti, če kliče metodo, ki je implementirana v jeziku Zero ali takšno, ki je dinamično dostopna iz neke knjižnice in je bila napisana npr. v jeziku C++. Podobne filozofije transparentnosti se drži jezik Java, ki omogoča invokacijo metod implementiranih v različnih programskih jezikih. Da bi dosegli zadostno stopnjo transparence, je potreben mehanizem, ki bo s pomočjo posredovanja virtualnega stroja omogočil za programerja "nevidno" preslikavo med metodo definirano v jeziku Zero in

njeno konkretizacijo v zunanji knjižnici.
Postavlja se vprašanje, kdaj opraviti preslikavo med obema metodama? Brez pretiranega razpredanja se izkaže, da je najbolje, če to preslikavo izvedemo kar v času prevajanja. S tem se ne samo ognemo zapletom v času izvajanja, pač pa tudi pridobimo na učinkovitosti, saj je za sleherno kalkulacijo med izvajanjem potreben dodaten čas. Operacijo preslikave smo zato umestili kar na nivo instrukcij virtualnega stroja. Sistemsko enoto virtualnega stroja `Core` smo nadgradili z dvema instrukcijama, eno za invokacijo zunanje procedure in drugo za invokacijo funkcije. Klic funkcije zahteva drugačno strukturo okvirja na skladu, saj vsebuje informacijo o lokalni spremenljivki, kamor se shrani rezultat. Prenos izvajanja v zunanje okolje je tako optimiziran za specifičen klic.
Vmesnik za uporabo zunanjega programskega koda je pomemben, v kolikor želimo učinkovito funkcionalnost metarazredov. Prav iz tega razloga ga prikazujemo na tem mestu, saj je funkcionalnost metaprogramskega modela jezika Zero implementirana skoraj izključno v jeziku C++ in dostopna programom, napisanih v jeziku Zero, preko tega vmesnika.

Signatura metode, katere implementacija se nahaja v zunanji knjižnici, mora biti posebej označena z rezervirano besedo `native`. Označba ima identičen pomen kot v Javi – da se implementacija metode nahaja v zunanji knjižnici. Vzemimo za primer razred `Input`, ki definira metodo `getLine`, katera s standardnega vhoda prebere niz znakov. Metodo definiramo na sledeč način:

```
class Input {
    public Input { }
    public  native getLine: String;
}
```

Metoda `getLine`, vrača instanco razreda `String` in znotraj razreda, v katerem je definirana, nima telesa. Njena implementacija je nevidna, vse kar je znano, je to, da vrača niz znakov. Takšna metoda je podobna abstraktni, saj podrobnosti njene izvedbe ne poznamo. Da bi lahko klic takšne metode na preprost način umestili v obstoječo arhitekturo jezika Zero, smo se odločili, da bomo klic metode ovili (wrapper) v navaden

virtualni klic. Prolog metode je enak za vse metode, le da tukaj dodamo posebno virtualno instrukcijo za invokacijo zunanje metode. Telo funkcije `getLine` se prevede v naslednji kod:

```
Core.new_frame_copy 4, 4
Core.invoke_n_f 25, 4
```

V prologu metode se ustvari okvir, nakar se izvede instrukcija `Core.invoke_n_f`, ki pomeni invokacijo zunanje funkcije. Instrukcija kot prvi operand prejme naslov literala imena metode in velikost okvirja v drugem operandu. Ekspliciten epilog metode je nepotreben, saj omenjena instrukcija opravi tudi vračanje vrednosti in čiščenje sklada po izstopu iz funkcije. Ime metode se generira v času prevajanja in se shrani v seznam literalov v razredni datoteki. Da med posameznimi metodami ne bi prihajalo do imenskih konfliktov, je potrebno določiti dosledno politiko poimenovanja zunanjih metod. Ime metode ja sestavljeno iz predpone, polnega imena razreda, imena metode, ter imen razredov formalnih parametrov. To zapišemo takole:

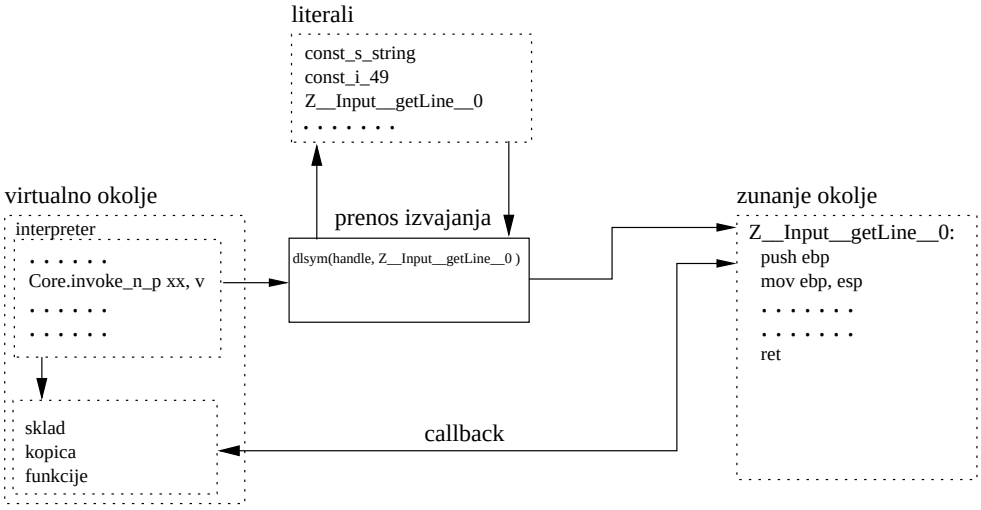
Z__ + ime razreda + __ + ime metode + __ število parametrov + __ ime razreda 1 + __ ime razreda 2 + ... + __ ime razreda n
Ime metode `getLine` se po takšni strategiji poimenovanja prevede v naslednjo obliko:

```
Z__Input__getLine__0
```

5.2.1 Implementacija klica

Dejanski klic funkcije iz knjižnice je realiziran z uporabo standardnih mehanizmov za dostop do dinamičnih knjižnic v okolju Unix. Ob inicializaciji virtualnega stroja se pripravijo potrebne dinamične knjižnice, iz katerih lahko med izvajanjem kličemo funkcije. Jedro klica je gotovo funkcija `dlsym`, ki prejme ime funkcije in vrne kazalec na izvršljiv kod te funkcije v pomnilniku. V tem trenutku se izvajanje prenese v zunanji prostor, kjer nimamo več neposrednega dostopa do notranjih struktur virtualnega stroja. Zunanja funkcija mora imeti dostop do vseh argumentov, ki jih prejme ob invokaciji v virtualnem okolju; tudi do prejemniškega objekta, nad katerim se izvaja. To omogočimo tako, da funkciji damo dostop do kazalca okvirja, ki je identičen

tistemu znotraj virtualnega stroja. Tako se izognemo nepotrebnemu kopiranju vrednosti okvirja v neko drugo strukturo. Druga stvar, ki jo je potrebno zagotoviti, je vsaj omejen dostop do okolja virtualnega stroja. Ni nenavadno, da zunanja funkcija želi ustvariti objekte, kateri morajo biti dostopni drugim objektom znotraj virtualnega okolja. Ker se vsi objekti ustvarjajo preko virtualnega stroja, potrebujemo še povratni (callback) mehanizem, s katerim se poslužujemo teh funkcionalnosti. Shema poteka klica zunanje funkcije je prikazana na sliki 4. Vsaka zunanja funkcija prejme natanko dva argumenta. Prvi je kazalec na okvir in drugi je kazalec na izvoženo okolje virtualnega stroja. Argumenti so dostopni preko okvirja in mehanizmi virtualnega stroja preko tega okolja. Ker ima programer zunanje funkcije neprimerno več vpogleda v sistemske lastnosti, mora zagotoviti verodostojnost programskega koda.



Slika 4: Shema invokacije zunanje metode.

5.2.2 Funkcionalnost zunaj virtualnega okolja

Zunanje funkcije morajo biti sposobne ustvarjati objekte znotraj virtualnega okolje, kar nadalje pomeni, da morajo imeti možnost klicati metode nad objekti, ki obstajajo samo v mejah virtualnega stroja. To omogoča povratni mehanizem, umeščen med virtualno in zunanje izvršilno okolje. Mehanizem programerju omogoča, da preko njegovih funkcionalnosti posega v sistemske lastnosti virtualnega stroja. Programer zunanjega koda

mora biti natančno seznanjem s strukturo okvirja dotične metode in pravili vračanja vrednosti iz funkcij. Neposreden dostop do okvirja metode ima to prednost, da vrednosti ostajajo nespremenjene in jih ni potrebno kopirati. V primeru dostopanja do skalarnih ali znakovnih vrednosti objektov vgrajenih primitivnih tipov, ni potrebno posebno poznavanje notranje strukture le-teh, saj so za to na voljo funkcije, ki to opravijo namesto programerja.

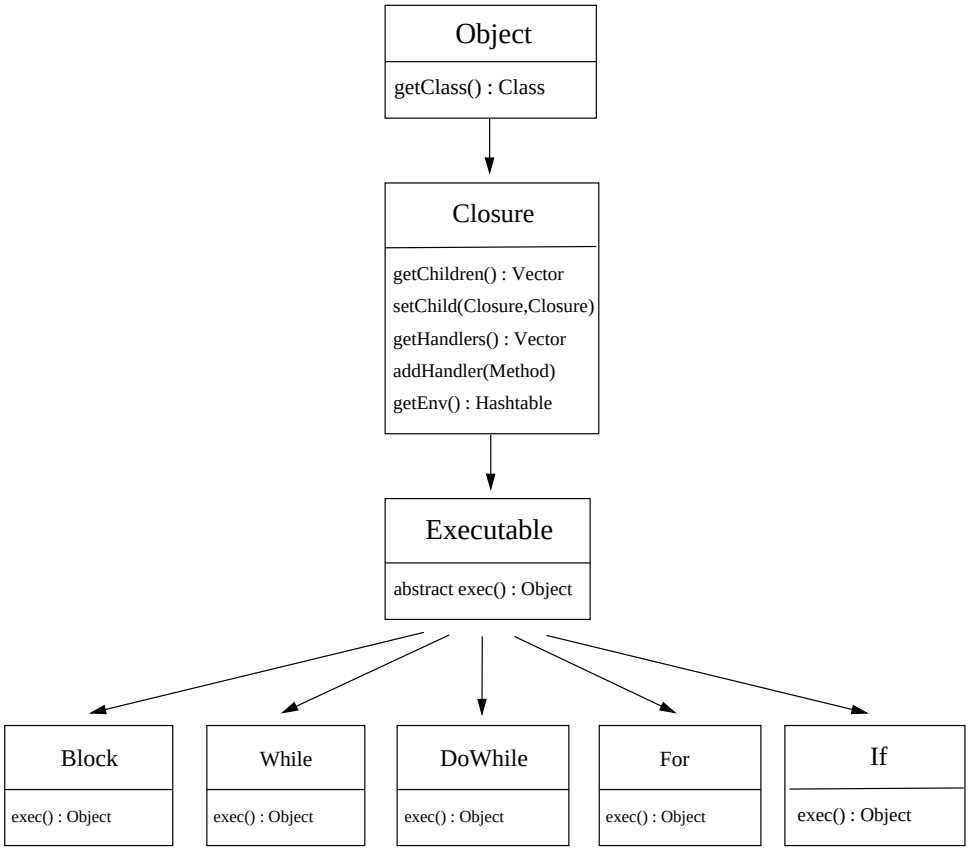
Da bi lahko invocirali virtualno metodo nad "notranjim" objektom, potrebujemo dostop do interpreterja. Metoda tukaj ni več povezana preko indeksa v virtualni tabeli, temveč jo moramo najti posredno z upoštevanjem njene celovite signature. Vsaka metoda razreda ima svoj opis v sekciji metarazrednih podatkov. Metanivo arhitekture in vmesnik za uporabo zunanjega koda sta ciklično povezana in dokaj tesno sklopljena. Mehanizmi za manipulacijo z metapodatki izvajalnega okolja namreč uporabljajo omenjeni vmesnik za svojo realizacijo v okviru jezika Zero, hkrati pa so potrebni za izpopolnitev funkcionalnosti samega vmesnika. V trenutku, ko imamo na voljo sveženj metapodatkov vseh metod, ni več težav pri invokaciji posamezne razredne metode, saj lahko njen indeks v virtualni tabeli in iz tega tudi njen pomnilniški naslov enoumno poiščemo posredno preko vnosov v tabeli metainformacij za signature vseh metod. Preostane le, da interpreterju podamo pomnilniški naslov in kod metode izvedemo.

5.3 Metainformacije obstoječih razredov

Temelj nadgradnje jezika Z_0 je vpeljava metaprogramskega modela programiranja. Jezik Z_0 je bil zasnovan na osnovi čiste objektne abstrakcije, kjer je vsaka vrednost predstavljena kot objekt oz. instanca razreda. Na tej zasnovi smo v jezikovni objektni model vpeljali metarazredno hierarhijo, ki služi za ogrodje metaprogramskemu modelu. Metarazredna hierarhija ne spreminja jezikovnega objektnega modela, temveč ohranja konsistenco obstoječih razredov. Že jezik Z_0 je postavil temelje določene metafunkcionalnosti. Spreminjanje metod je namreč pomenil strukturalni poseg in spremembo v instanci razreda, kar lahko klasificiramo kot metaoperacijo.

Ohranitev objektnega model jezika Z_0 smo dosegli s posegom v korenski del razredne hierarhije – v razred `Object`. Razredu smo dodali metodo `getClass()`, ki vrne instanco

metarazreda `Class`. Razred `Class` je osnova za vse meta posege v katerikoli razred hierarhije. Ker je metoda `getClass()` definirana v korenskem razredu hierarhije, lahko po svojem razredu poizveduje vsak razred. Kar zadeva strukturo in manipulacijo, je razred `Class` povsem enak kateremukoli "navadnemu" razredu hierarhije. Funkcionalno pa sodi v svoj segment, saj omogoča metaoperacije, ki jih drugi razredi ne omogočajo. Ko bi hoteli razred `Class` uporabiti za metaoperacije vseh konstruktov, bi razred postal prekompleksen in predvsem nehomogen glede funkcionalnosti. Nekateri jezikovni konstrukti so namreč dovolj zapleteni, da lahko njihovo metafunkcionalnost implementiramo v ločenih razredih. Iz tega razloga smo vsak krmilni konstrukt jezika implementirali z dokaj razdrobljeno hierarhijo 4 razredov, kot jo prikazuje slika 5.



Slika 5: Razredna hierarhija krmilnih konstruktov.

Kot je razvidno s slike, temeljijo krmilne strukture jezika na razredu `Closure`. `Closure` je zaradi svoje funkcionalnosti metarazred, saj omogoča ne samo vpogled v strukturo

temveč tudi njeno spreminjanje. Več o metafunkcionalnosti krmilnih struktur bomo povedali v sledečih poglavjih. Kot bo razvidno v sledečih poglavjih, je večina metafunkcionalnosti zapisana v jeziku C++. To je nakazano v signaturi metod metarazredov s posebnim specifikatorjem `native`. Primarni razlog temu je, kot smo omenili v prejšnjem poglavju, učinkovitost.

5.3.1 Metarazred `Class`

Razred `Class` je temeljni metarazred vsakega razreda v jeziku. Do njega lahko dostopamo preko invokacije metode `getClass()`, definirane v razredu `Object`. Vsak razred jezika opišemo z instanco razreda `Class`, ki je specifična glede na razred, katerega opisuje. Tako ima instanca razreda vselej iste metode, vendar te vračajo druge informacije. Razred `Class` imenujemo eksplicitni metarazred, saj ni implicitno podedovan v nobenem razredu. Njegovo instanco je mogoče ustvariti samo eksplicitno – kot že rečeno, z invokacijo metode `getClass()`. Razred `Class` omogoča metaprogramski model z uporabo refleksije. Refleksivna funkcionalnost razreda se deli na statično ali introspektivno in dinamično. Statična refleksija ne spreminja tekočega stanja. Njena zmožnost je zgolj poizvedovati po informacijah, zato temu pravimo tudi introspekcija ali "vpogled vase". Dinamična refleksija ima moč spremeniti stanje tekočega sistema. Njena izraznost določa, do katere stopnje lahko vrši spremembe v objektih.
Metafunkcionalnost razreda `Class` je definirana z refleksivnimi metodami. Razred `Class` smo definirali na sledeč način:

```
class Class {
  // protected constructor
  protected Class;
  restricted native getName : String;
  restricted native setName(String name);
  restricted native isClassOf(Object o) : Boolean;
  restricted native isSubclassOf(Class c) : Boolean;
  restricted native isSuperclassOf(Class c) : Boolean;
  restricted native getDeclaredConstructors : Vector;
  restricted native getDeclaredMethods : Vector;
```

```
    restricted native getDeclaredMethod(String n, Vector p) : Method;
    restricted native getDeclaredMethod(String n) : Method;
    restricted native getSuperClasses : Vector;
    restricted native setMethod(Method from, Method to);
    restricted native setSuper(Class from, Class to);
}
```

Metafunkcionalnost razreda `Class` je definirana na dokaj "grobem" nivoju. To je pogojeno z objektnim modelom jezika Zero. Razred `Class` je "vstopna točka" metaprogramiranja. Specifična metafunkcionalnost se nahaja na nižjih, bolj razdelanih nivojih. Na nivoju razreda `Class` lahko operiramo z neposrednimi nadrazredi in metodami razreda. Metode, s katerimi pridobimo informacije o definiranih metodah razreda, služijo za prehod na nižji in bolj specifični nivo metafunkcionalnosti. Objektni model s takšnim ločevanjem metafunkcionalnosti postane preglednješi, "čistejši" in lažje razumljiv.
Metode statične refleksije razreda `Class` so:

- `getName : String`
  vrne ime razreda,
- `isClassOf(Object o) : Boolean`
  vrne `true`, če je objekt `o` instanca tega razreda,
- `isSubclassOf(Class c) : Boolean`
  vrne `true`, če je razred `c` nadrazred tega razreda,
- `isSuperclassOf(Class c) : Boolean`
  vrne `true`, če je razred `c` podrazred tega razreda,
- `getDeclaredConstructors : Vector`
  vrne vektor konstruktorskih metod definiranih v tem razredu,
- `getDeclaredMethods : Vector`
  vrne vektor metod tega razreda,

- `getDeclaredMethod(String n, Vector p) : Method`
  vrne razredno metodo s podano signaturo,
- `getDeclaredMethod(String n) : Method`
  vrne razredno metodo s podano signaturo (v imenu),
- `getSuperClasses : Vector`
  vrne vektor lineariziranih neposrednih nadrazredov tega razreda.

Metode strukturalne refleksije razreda `Class` so tri:

- `setName(String name)`
  nastavi ime razreda,
- `setMethod(Method from, Method to)`
  zamenja metodo razreda,
- `setSuper(Class from, Class to)`
  zamenja neposredni nadrazred.

Metode statične refleksije omogočajo vpogled oz. introspekcijo razreda. Metode introspekcije ločimo od metod, ki omogočajo spremembe, saj so prve povsem varne, medtem ko je pri drugih potrebna previdnost. V razredu `Class` definirane metode statične refleksije se uporabljajo za pridobivanje informacij o razredu. Metoda `SetName` spremeni ime razreda, kar presega introspekcijo. Metodi `setMethod` ter `setSuper` spreminjata notranjo strukturo razreda in ju zato uvrščamo med metode strukturalne refleksije.
Metoda `setMethod` nadomesti razredno metodo. Ker se za refleksijsko operacijo zahteva, da ohrani konsistenco in varnost tipiziranja v času izvajanja, mora novo metoda imeti identično signaturo kot originalna. Na ta način ostane protokol nespremenjen, čeprav nova metoda ne v funkcionalnem ne v strukturalnem smislu ni vezana na staro metodo. Metoda, ki nadomesti staro, je lahko metoda drugega razreda, anonimna metoda ali instančna metoda. V prvem primeru razredna metoda nadomesti razredno metodo, v drugih dveh primerih je nova metoda že vezana na specifičnega prejemnika. Tako anonimna kot instančna metoda sta namreč definirani v kontekstu nekega prejemnika. V tem primeru je nova razredna metoda pravtako vezana na svoj originalni

prejemnik. To je pomembno s stališča varnosti izvajanja, saj metode brez prejemnika ne bi mogli invocirati. Poudarimo, da kompatibilnost signature pri nadomeščanju metod ne pomeni zgolj konformanco tipa po relaciji podtipa, temveč identiteto signatur. V nasprotnem primeru bi lahko prišlo do neskladja med redefiniranimi metodami v razredni hierarhiji.

Metoda `setSuper` deluje po podobnem principu, vendar na nivoju razreda. S `setSuper` je mogoče nadomestiti kateregakoli izmed neposrednih nadrazredov. Posredne nadrazrede, torej tiste, ki se nahajajo na višjih ravneh razredne hierarhije, je mogoče nadomestiti rekurzivno. Kar zadeva tipiziranje, ima metoda `setSuper` enako zahtevo kot `setMethod` – kompatibilnost signature. Razlika je samo v tem, da se sedaj signatura metode razširi na signaturo razreda, ki pa ni nič drugega kot skupek signatur vseh, v razredu definiranih metod. Tako mora nov razred, ki zamenjuje starega, imeti metode, katerih signature so identične tistim iz starega razreda. Samo na ta način je namreč mogoče zagotoviti kompatibilnost tipov in s tem varnost tipiziranja.

Kot je razvidno iz definicije metarazreda `Class`, ni možno dodajati ali odvzemati razrednih metod. To je razumljiva omejitev statično tipiziranega in prevedenega programskega jezika. Če bi bilo v razred mogoče dodati novo metodo, bi to lahko porušilo kompatibilnost razredne hierarhije. Prikažimo to na primeru dveh preprostih razredov.

```
class Base {
  restricted Base { ... };        // konstruktor
}

class Derived  inherits Base {
  restricted Derived { ... };     // konstruktor
  restricted print(String) { ... };
}
```

Če v gornji hierarhiji vzamemo nadrazred `Base` in mu želimo dodati novo metodo, npr. `print`, bi morala nova metoda imeti kompatibilno signaturo z istoimensko metodo definirano v podrazredu `Derived`. To bi načelno lahko preverili, vendar samo, ko bi poznali vse razrede, ki dedujejo iz razreda `Base`. Praktično je to nemogoče preveriti, saj nikoli ne moremo poznati vseh razredov, ki bodo "kdajkoli" dedovali iz tega razreda. Definirajmo še en razred, ki deduje iz `Base` takole:

```
class Derived1 inherits Base {
  restricted Derived1 { ... };   // konstruktor
  restricted print(Integer) { ... };
}
```

Kakšno signaturo bi metoda `print`, ki bi jo dodali v skupen nadrazred `Base`, morala imeti v tem primeru? Iz zahteve statično preverljivega variantnega sistema tipov sledi, da bi argument metode `print` moral biti podtip obeh argumentnih tipov istoimenske metode v podrazredih. Lahko najdemo tak tip? V našem primeru ne, pa tudi ko bi ga, se porodi vprašanje, ali bi tak tip ustrezal funkcionalnosti naše nove metode. Pojavi pa se še ena težava. Jezik Zero je namreč preveden v zložni kod, tj. v instrukcije virtualnega stroja. Invokacije metod se izvajajo s pomočjo indeksa v virtualno tabelo metod. Indeks v to tabelo je potreben v času generiranja instrukcij, tj. v času prevajanja in je odvisen od celotne nadrazredne hierarhije prevajanega razreda. Ko bi metodo v času izvajanja dodali v nadrazred, bi bilo treba preračunati indekse vseh razredov, izpeljanih iz tega nadrazreda. Toda ker so razredi že prevedeni, bi to pomenilo poseg na binarni nivo prevedenega koda. Povrhu vsega se ponovi prva težava – kateri razredi so izpeljani iz tega razreda? Podobne ovire nastanejo pri dodajanju metode v razred. Iz teh izsledkov zaključimo, da tovrstna dinamična manipulacija s stališča statičnega tipiziranja ni dopustna. Metafunkcionalnost se v jeziku Zero mora prilagoditi obstoječemu sistemu tipiziranja, čeravno to pomeni, da ji pristrižemo krila.

5.3.2 Metarazred `Closure`

Metarazred `Closure` je implicitni metarazred vseh krmilnih konstruktov jezika. Njegovo umeščenost v razredno hierarhijo prikazuje slika 5 (glej stran 85). `Closure` imenujemo implicitni metarazred zato, ker njegovo funkcionalnost dedujejo vsi razredi krmilnih struktur `Block`, `While`, `DoWhile`, `For` in `If`. To pomeni, da je metafunkcionalnost razreda dostopna pri manipulaciji vsake krmilne strukture. Za takšno zasnovo smo se odločili zaradi preprostosti in neposrednosti uporabe metafunkcionalnosti, saj želimo v jeziku Zero metaprogramiranje zvesti na nivo klasičnega programiranja. To želimo storiti brez vpeljave kakih posebnonamenskih konstruktov. Najlažja rešitev se ponudi v obliki implicitno dedovanega nadrazreda.

Razred `Closure` je definiran na sledeč način:

```
class Closure {
   // protected constructor
   protected Closure;
   restricted native isTopLevel : Boolean;
   restricted native getChildren : Vector;
   restricted native setChild(Closure from, Closure to);
   restricted native getHandlers : Vector;
   restricted native addHandler(Method h);
   restricted native getEnv : Hashtable;
}
```

Ker je razred `Closure` po svojem namenu metarazred, se vse njegove metode nanašajo na metafunkcionalnost. Vpogled v strukturo kateregakoli zaprtja nam omogoča metoda `getChildren`, ki vrača seznam vseh neposredno vgnezdenih zaprtij. Do zaprtij, torej blokov in drugih krmilnih struktur, na nižjih nivojih, lahko dostopamo rekurzivno preko te iste metode. Na ta način pridemo do strukturalne zgradbe zaprtja, ki je lahko katerakoli krmilna struktura jezika. Vendar kljub temu, da vgnezdena struktura nikoli ni "samo" zaprtje, temveč vedno njegova konkretizacija (npr. zanka *while* ali stavek *for*), je pomembno, da metoda `getChildren` vrača vektor zaprtij, tj. objektov tipa `Closure`. V jeziku Zero je tudi metoda, strukturalno gledano, zaprtje. Toda zaprtje ima dve stanji. Lahko predstavlja konkretno krmilno strukturo, ki obstaja kot objekt, lahko pa predstavlja krmilno strukturo, ki še ni objekt. Do slednjega primera pride pri vpogledu v strukturo razredne metode, ko razred še ni instanciran. V tem primeru imamo samo kod metode, vključno z njenimi vgnezdenimi strukturami. Razumljivo in smiselno je, da strukturalna dekompozicija z metodo `getChildren` omogoča ločeno obravnavo posameznih zaprtij. Tako lahko obravnavamo poljubno vgnezdeno krmilno strukturo znotraj metode, in jo, kot prvorazredno vrednost, podamo kot argument drugi metodi ali jo vrnemo iz metode. V primeru, ko smo takšno krmilno strukturo pridobili iz razredne metode, je ne moremo samostojno izvršiti. Razlog temu je preprost – struktura lahko zavisi od okolja svoje starševske strukture, ki pa še ne obstaja. Navedimo kratek primer za ilustracijo tega problema:

```
class Primer {
```

```
  restricted Primer {              // konstruktor
   Closure c = ( method mojaMetoda() ).getClosure();
   Closure zanka = c.getChildren().at(0);
  }

  restricted mojaMetoda {
    Integer a = 1;
     while{ a < 10; }
     {
        (‘‘a je ’’ + a).print();
        a++;
     };
  };
}
```

V konstruktorju metode lahko dostopamo do zaprtja razredne metode `mojaMetoda`; to predstavlja njeno telo. Dostop do zaprtja metode omogoča, da pridemo do zanke *while* v telesu metode. Toda te zanke ne moremo izvršiti iz dveh razlogov. Zanka dejansko ne obstaja kot objekt, pa tudi njeno starševsko okolje, ki vsebuje spremenljivko `a`, še ni ustvarjeno. Okolje metode `mojaMetoda` se ustvari šele ob njeni invokaciji.
S tem, ko metoda `getChildren` vrača seznam tipa `Closure`, preprečimo, da bi takšna zaprtja sploh bila izvedljiva, saj razred `Closure` nima metode, ki bi izvršila njegov kod. Tak razred je njegov podrazred `Executable`, ki definira abstraktno metodo `exec`, ki kod zaprtja izvrši. Vendar imajo tip `Executable` le zaprtja, ki predstavljajo konkretne krmilne strukture. V tem primeru `getChildren` seveda še vedno vrača zaprtja kot tip `Closure`, vendar je te objekte mogoče pretvoriti nazaj v njihove konkretne primerke. To pomeni, da lahko takšno zaprtje brez težav izvršimo tudi zunaj konteksa, v katerem je bilo definirano. To je razumljivo, saj je zaprtje struktura, ki vsebuje kod in okolje in je kot tako sposobno zapustiti originalen kontekst. V prejle opisanem primeru razredne metode to ni mogoče. S tako zasnovano razredno hierarhijo se izognemo marsikateremu zapletu, ki bi sicer nastal pri omejevanju izvrševanja zaprtij zunaj konteksta definicije.

Vpogled v strukturo je le prvi korak manipulacije. Možnost dejanske spremembe strukture je dosegljiva z metodo `setChild`. Metoda opravi podobno nalogo kot že omenjeni metodi metarazreda `Class` – `setMethod` in `setSuper`, torej zamenjavo. Metoda `setChild` nadomesti zaprtje z drugim zaprtjem. Zaprtje, ki ga želimo nadomestiti, referencira prvi argument, novo zaprtje je podano v drugem argumentu. Do zaprtja, ki ga želimo nadomestiti, lahko pridemo samo preko metode `getChildren`. Novo zaprtje je lahko obstoječe zaprtje iz metode, del krmilne strukture ali povsem uporabniško definirano zaprtje v obliki krmilne strukture. V praksi je najpogostejši prav slednji primer, kjer novo zaprtje predstavlja prvorazreden blok. Podobno kot pri metodi `getChildren`, lahko tudi tukaj do globlje vgnezdenih zaprtij dostopamo rekurzivno. Na ta način je mogoče spremeniti celotno strukturo krmilnega konstrukta.

"Zrnatost" manipulacije v jeziku Zero je blok. Pod to mejo ne moremo, saj bolj elementarna entiteta nima svoje objektne lupine, na kateri sicer temelji možnost manipulacije. Entiteta, ki je, kar zadeva elementarnost, osnovnejša od bloka, je izraz. Če vzamemo nadvse preprost primer izraza inkrementacije `a++;`, lahko zanj trdimo, da je bolj "temeljna" oz. bolj elementarna entiteta kot blok, ki lahko sam vsebuje mnoštvo izrazov. Toda izraz sam po sebi nima refleksijske zmožnosti. To pomeni, da ga ne moremo manipulirati v smislu metaprogramiranja, na način, kot to počnemo z bloki. Vzrok temu je "neobjektna" predstavitev izraza. "Najnižja" entiteta, ki je še objekt, je blok. Iz tega sledi, da če hočemo manipulirati izraz, ga moramo umestiti v objektno lupino, to pomeni, dati ga v blok. Če bi na nekem mestu v programu želeli spremeniti strukturo na nivoju posameznega izraza, bi ta izraz morali umestiti v svoj blok. Nato bi funkcionalnost izraza spremenili tako, da bi spremenili njegov blok.

Seveda se na tem mestu porodi še druga zamisel. Kako je z manipulacijo koda na binarnem nivoju? Tam bi namreč lahko manipulirali izraz sam po sebi, saj je ta le zaporedje instrukcij. Z možnostjo manipulcije le-teh dobimo zmožnost manipulacije izraza. To drži. Na nivoju instrukcij je mogoče marsikaj. Vendar govorimo v jeziku Zero o visokonivojskem metaprogramskem modelu. Ko bi omogočili manipulacijo instrukcij, bi več ne potrebovali razdelanega metaprogramskega modela v obliki razredne hierarhije. Povrhu tega je na nivoju prevedenega koda mnogo težje zagotavljati varnost izvedenih sprememb. In nenazadnje, manipulacija prevedenega koda zahteva bistveno

več znanja kot manipulacija na nivoju programskega koda.

Metoda `getHandlers` vrne seznam metod, ki so "pripete" na zaprtje. Kaj pomeni, da so pripete? Zaprtju lahko določimo funkcionalnost, ki se izvede vsakič, ko se izvede kod zaprtja. Tako funkcionalnost opišemo v obliki metode. Zaprtje ima lahko poljubno mnogo takšnih metod, ki so shranjene v seznamu. Metoda `getHandlers` torej vrača ta seznam. Ob izvedbi zaprtja se pripete metode izvedejo v istem vrstnem redu, kot si sledijo v seznamu. Če je funkcionalnost izražena v obliki metode, se postavlja vprašanje, kakšen metodni tip opiše signaturo metode? Ker zaprtju ni moč podati parametra, mora ta metodni tip biti generičen tip `Method`, brez parametra in brez vračanja vrednosti. Toda zaprtje lahko predstavlja tudi telo metode. V tem primeru ima smisel, da se more parameter metode prenesti v pripeto metodo. To omogočimo z zahtevo, da se mora signatura pripete metode ujemati s signaturo metode, na katere zaprtje metodo pripenjamo. Argument, ki je podan metodi, se avtomatsko prenese v pripeto metodo, kar funkcionalnosti te metode omogoča obravnavo podanih argumentov. S tem je omogočena finejša in parametrizirana manipulacija obnašanja zaprtja. Metoda `addHandler` pripne metodo zaprtju. Metoda prejme generičen metodni tip.
S pripenjanjem metode na zaprtje je mogoče manipulirati obnašanje tega zaprtja. V splošnem je obnašanje mogoče doseči že z manipulacijo strukture, toda v primeru pripete metode gre za manipulacijo obnašanja brez posega v strukturo. Takšen pristop je gotovo manj invaziven in iz tega razloga bolj zaželen, kadar je moč doseči spremembo obnašanja brez potrebe po spremembi strukture.
Metoda `addHandler` torej omogoča behavioralno refleksijo v jeziku. Toda do katere stopnje? Ker metoda deluje na nivoju zaprtja, je s tem mogoče manipulirati obnašanje vsakega bloka. To ne pomeni zgolj krmilnih struktur v celoti, temveč tudi selektivne dele le-teh, npr. samo pogojni blok zanke *while*. Toda ker so zaprtja v jeziku Zero tudi osnova metod, sledi, da je mogoče spreminjati obnašanje vsake metode. Če vzamemo za primer metodo `exec` razreda `Executable`, ki je nadrazred vseh izvršljivih krmilnih struktur, lahko s tem, da metodi pripnemo specifično funckionalnost, spremenimo obnašanje. Tako bi lahko npr. spremenili obnašanje izvršitve zanke *while*.
Zero torej nima kakih specialnih konstruktov behavioralne refleksije. Vsa manipula-

cija obnašanja se vrši preko metod. Funkcionalnosti programa, ki ni izražena v obliki metode, z obstoječo shemo ne moremo spreminjati. Med tako funkcionalnost spada shema dedovanja, ki je v jeziku Zero vnaprej določena in je zato ni moč spreminjati. Ko bi tudi ta shema bila izražena z metodo, bi nanjo lahko vplivali z behavioralno in strukturalno refleksijo. Vzrok za to, da je shema dedovanja v jeziku fiksirana, pa je zdaj že jasen.

Trenutna zasnova pripetih metod ima omejitev. Metode se izvršijo samo pred izvedbo zaprtja. Bolj dodelano manipulacijo obnašanja bi dosegli z možnostjo specifikacije točke, kje se bo metoda izvedla. V aspektno usmerjenem programiranju se taka točka imenuje stična točka (join point) in predstavlja mesto v programu, kjer je njegovo funkcionalnost moč razširiti. V jeziku Zero je ta točka vselej neposredno pred zaprtjem. Našo trenutno shemo bi lahko brez večjih težav dopolnili z dodatnimi točkami "na koncu" in "namesto", kjer slednja pomeni, da se metoda izvede namesto zaprtja. Na ta način bi lahko obnašanje spremenili do te mere, da se originalna funkcionalnost sploh ne bi izvedla. Toda ob tem se že postavlja vprašanje smiselnosti takih zmožnosti. Kar pa zadeva metode, Zero takšno moč "popolne" manipulacije že ima, saj omogoča nadomeščanje metod v celoti.

Metaprogramiranje se vrši na razrednem in instančnem nivoju. To je pomembno tudi v oziru razreda `Closure`. Ko smo že omenili, je mogoče zaprtje obravnavati na dva načina: kot del obstoječe izvršljive strukture ali kot del neizvršljive strukture, do katere pridemo v razredni metodi. Behavioralna manipulacija na nivoju instance ima vpliv samo na to instanco. Manipulacija na nivoju razreda učinkuje na vse instance tega razreda. Tako bi lahko spremenili obnašanje vseh zank *while* s tem, da bi spremenili obnašanje metode `exec` v razredu `While`. Taka sprememba bi učinkovala na vse omenjene zanke, ustvarjene od trenutka spremembe metode `exec`.

Zadnja metoda definirana v razredu `Closure` je `getEnv`, ki vrne okolje zaprtja v obliki sekljalne tabele. Sekljalna tabela vsebuje pare *ključ, vrednost*, kjer ključ predstavlja ime spremenljivke in vrednost referenco te spremenljivke. V prejle definiranem raz-

redu `Primer` bi zaprtje metode `mojaMetoda` vsebovalo spremenljivko `a`, do katere vrednosti v sekljalni tabeli bi dostopali z metodo `getByKey(‘‘a’’)`, definirano v razredu `Hashtable`. Zakaj je potreben poseben mehanizem za dostop do okolja zaprtja? Do spremenljivk zaprtja lahko neposredno dostopamo samo v primeru, ko jih poznamo, tj. kadar zaprtje manipuliramo v njegovem kontekstu. Kadar pa zaprtje zapusti svoj kontekst, če ga npr. podamo kot argument neki metodi, potem je jasno, da v tej metodi ne moremo do spremenljivk zaprtja več dostopati preprosto s sklicevanjem na njihova imena. Uporabiti moramo metodo `getEnv`, ki vrne okolje zaprtja, v katerem se nahajajo vse spremenljivke, uporabljene v kodu zaprtja.

5.3.3 Metarazred `Method`

Razred `Method` je, podobno kot razred `Closure`, implicitni metarazred. Med metarazrede ga prištevamo, ker omogoča dostop do zaprtja metode in vpogled v informacije o metodi. Metarazred `Method` pa ima ob svoji metafunkcionalnosti še vlogo metodnega tipa. Metoda, ki je v jeziku Zero prvorazredna referenčna vrednost, mora namreč imeti svoj tip. Vsako metodo lahko opiše splošen (generičen) tip `Method`. Toda metoda ni preprosta vrednost kakor število, znak ali niz. Za metodo vemo, da lahko prejme poljubno število parametrov in vrača ali ne vrača vrednosti. Iz tega razloga je metodni tip v jeziku Zero parametriziran. Več bomo o tem povedali v poglavju o metodah višjega reda.
Posebnost metarazreda je torej v tem, da ga je mogoče parametrizirati. Razred smo definirali takole:

```
class Method {
  // protected constructor
  protected Method;
  restricted native getName : String;
  restricted native setName(String name);
  restricted native getReturnType : Class;
  restricted native getParameterTypes : Vector;
  restricted native getClosure : Closure;
  restricted native invoke(Object receiver, Vector params):Object;
```

```
    restricted native invoke(Vector params): Object;
    restricted native isConstructor : Boolean;
    restricted native isFunction : Boolean;
    restricted native isStatic : Boolean;
    restricted native isNative : Boolean;
    restricted native isPublic : Boolean;
    restricted native isProtected : Boolean;
    restricted native isPrivate : Boolean;
    restricted native isRestricted : Boolean;
    restricted native isAnonymous : Boolean;
    restricted native toString : String;
}
```

Metoda `getName` vrne ime metode brez njene signature. `setName` nastavi ime metode. Novo ime metode ne vpliva na relacije med istoimenskimi metodami v razredni hierarhiji. To pomeni, da ostane klic metode, ki je že preveden v kod, nespremenjen. Drugače bi se soočali s podobnimi problemi, kot smo jih opisali v že omenjenem primeru, ko bi bilo mogoče metode v razred tudi dodajati in izvzemati. Od klica metode `setName` ima metoda novo ime, ki se vidi preko refleksije. Razred ima dve metodi `invoke` za invokacijo metode, eno za razredne in drugo za instančne metode. Ostale introspektivne metode razreda omogočajo vpogled v metodo in ne spreminjajo njene strukture ali obnašanja. Posebej jih ne opisujemo, saj jih dovolj nazorno razlagajo že njihova imena. Metoda, ki zasluži posebno pozornost, je `getClosure`. Z njo je mogoč dostop do zaprtja metode, nad katerim lahko izvajamo strukturalno in behavioralno refleksijo preko metarazreda `Closure`. S stališča metaprogramiranja je metoda `getClosure` torej vstopna točka v strukturo metode.

Metode tako ne manipuliramo neposredno v razredu `Method`, temveč je potrebno najprej priti do njenega zaprtja. Statična varnost tipiziranja zahteva skladnost tipov ves čas izvajanja. To pomeni, da refleksija ne omogoča manipulacije tipov argumentov in tipa vračanja. Signatura metode mora ostati enaka.

Povedali smo že, da zaprtja metode ni mogoče izvesti, saj razred `Closure` nima metode `exec`. Metodo je mogoče izvesti samo celovito, kot metodo, ne kot zgolj zaprtje. To

omogočata metodi `invoke`. Jezik Zero ima zaradi prvorazrednosti metod še dodaten mehanizem izvedbe metod. Metodo je mogoče izvesti preko t.i. metodne spremenljivke, kateri preprosto podamo argumente. Več o taki izvedbi v poglavju o metodah višjega reda.

5.4 Metaprogramiranje v jeziku Zero

Pomemben aspekt jezika Zero, ki smo ga uvrstili v sam vrh željenih lastnosti, je enostaven metaprogramski model, kot ga vidi programer. V jeziku Zero bi se metaprogramiranje naj približalo klasičnemu programiranju. Že od jezika Z_0 dalje je metaprogramiranje implicitno, saj je vsaka nadomestitev metode po svojem bistvu metaoperacija. Nadomeščanje metod je v našem jeziku edini mehanizem manipulacije stanja objekta. Metaprogramiranje tako postane "nujno". Jezik Zero razširja metafunkcionalnost še na druge koncepte. Idejna zasnova jezika je metaprogramski model, prirejen statično tipiziranemu jeziku. V pričujočem podpoglavju želimo prikazati praktični vidik uporabe tega metaprogramskega modela.

5.4.1 Statična refleksija

Velik del refleksijske funkcionalnosti jezika Zero je namenjen vpogledu v stanje, tj. introspekciji. Zgolj vpogled ne spreminja obnašanja in strukture posameznih konstruktov. S tega vidika je takšna refleksija povsem varna. Refleksija metarazredov omogoča dostop do vseh pomembnih informacij. Prikažimo tak vpogled v razred na primeru preprostega razreda `Raziskovalec`:

```
class Raziskovalec  inherits Oseba, Zaposleni{
  restricted Raziskovalec {            // konstruktor
    ...
  }
  // vrni status raziskovalca
  restricted getStatus : String {  ... };
  // nastavi status raziskovalca
```

```
  restricted setStatus(String s) {  ... };
  // vrni tevilko raziskovalca
  restricted getStevilka : Integer {  ... };
  // nastavi stevilko raziskovalca
  restricted setStevilka(Integer s) {  ... };
  // vrni raziskovalca kot niz
  restricted toString : String {  ... };
  ...
  // metoda main
  restricted main {
    Raziskovalec raz =  new Raziskovalec;
    Class c = raz.getClass();   // pridobi razred raziskovalca

     // izpisi podatke o nadrazredih
    Vector razredi = c.getSuperClasses();
    Integer i;
     for{ (``Narazredi razreda '' + c.getName() + `` so:'').print(); i=0;}
          { i < razredi.size();} { i++; }
       {
           (``razred '' + (razredi.at(i)  as Class).getName().print();
       };  // for

     // izpisi podatke o metodah
    Vector metode = c.getDeclaredMethods();
     for{ (``Metode razreda '' + c.getName() + `` so:'').print(); i=0;}
          { i < metode.size();} { i++; }
       {
           (``metoda '' + metode.at(i).toString()).print();
       };  // for
  }   // main
}
```

Metoda `toString`, definirana v metarazredu `Method`, uporablja metafunkcionalnost

ostalih metod, definiranih v tem istem razredu. Za posamezno metodo je mogoče dobiti informacije o tipu vračanja, tipih parametrov, pravicah dostopa in drugih specialnih identifikatorjih ter ime metode. To je dovolj, da lahko metodo invociramo preko metode `invoke`.

Metoda `getSuperClasses` vrne seznam neposrednih predhodnikov našega razreda. Seznam vsebuje nadrazrede, kot si sledijo od leve proti desni. Do nadrazredov na višjih nivojih razredne hierarhije je mogoče dostopati rekurzivno.

5.4.2 Dinamična refleksija

Omenili smo že, da je prvi korak strukturalne refleksije omogočen z dostopom do metarazreda `Closure`, katerega funkcionalnost je implicitno podedovana v vseh razredih krmilnih konstruktov. Za metaprogramiranje na nivoju krmilnih konstruktov tako ni potreben noben specialni mehanizem, temveč se poslužujemo kar podedovane funkcionalnosti. Pokažimo to na preprostem primeru selekcije, torej stavka *if*, katerega strukturo spremenimo v času izvajanja:

```
Integer n = 44;
// shranimo stavek if v spremenljivko
Executable myIf =
    if { n % 2 == 0; }  then
       { (n + '' je sodo'').print(); }
    else
       { (n + '' je liho'').print(); };
   ...
// izvrsi stavek
myIf.exec();
// spremenimo pogojni blok, ki naj vsebuje negiran pogoj
myIf.setChild( myIf.getChildren().at(0), { n % 2 != 0; } );
// izvrsi s spremenjenim pogojem
myIf.exec();
// nadomestimo telo stavka
myIf.setChild( myIf.getChildren().at(1),
```

```
                    { (n + '' je zdaj liho'').print(); }
            );
// drugi blok
myIf.setChild( myIf.getChildren().at(2),
                    { (n + '' je zdaj sodo'').print(); }
            );
myIf.exec();
```

Stavek *if* je kot krmilni konstrukt podrazred razreda `Executable`, ki je sam podrazred razreda `Closure`. Metoda `getChildren`, ki jo invociramo nad konstruktom, vrne vektor treh zaprtij: pogojni blok in dva bloka, od katerih se izvrši vedno samo en. Zaprtje pogojnega bloka je torej na prvem mestu v vektorju, kar povemo s stavkom `myIf.getChildren().at(0)`. S pomočjo metode `setChild` nadomestimo ta blok z novim blokom, katerega podamo v prvorazredni obliki. Gre samo za obrnitev pogoja, saj operator `==` samo nadomestimo z operatorjem `!=`. Stavek *if* po tej operaciji izpiše ravno obratno kot v svoji prvotni obliki. Preostala dva bloka nadomestimo na podoben način. V vektorju se nahajata na lokacijah 1 in 2. Bloka nadomestimo z novima blokoma, katera pravtako podamo kot prvorazredni vrednosti. Po zadnjih dveh zamenjavah ima stavek sledečo strukturo:

```
if { n % 2 != 0; }  then
    { (n + '' je zdaj liho'').print(); }
else
    { (n + '' je zdaj sodo'').print(); };
```

Stavku *if* smo spremenili strukturo, s tem, da smo zamenjali vse tri bloke. Takšna strukturalna manipulacija krmilnega konstrukta je povsem varna. V pričujočem primeru smo prikazali manipulacijo stavka *if*, ki pa deluje enako tudi za ostale konstrukte. Postopek strukturalne manipulacije kateregakoli konstrukta se prične z njegovo dekompozicijo, z metodo `getChildren`. Nad zaprtji, ki jih ta metoda vrne, lahko nato vršimo akcije, ko so zamenjava, dodajanje funckionalnosti in vpogled na globlje ravni strukture.

Strukuralno gledano je stavek *if* objekt, ki vsebuje vse svoje bloke kot objekte, tj. reference teh objektov. Posamezni bloki v stavku obstajajo kot ločeni objekti, do katerih lahko dostopamo s pomočjo refleksije. Takšna zasnova čistega objektnega modela je

temelj za implementacijo sprememb, ki smo jih pravkar opisali. Če bi se poglobili v podrobnosti implementacije, bi videli, da gre pri takšnih zamenjavah blokov dejansko za zamenjavo referenc in preverjanje konformance tipov. Posamezne bloke krmilnega konstrukta si zamislimo kot samostojne, prevedene fragmente programskega koda. V času izvajanja se krmilni konstrukt ustvari kot skupek teh fragmentov, ki mu dajo logično vrednost. Fragment, ki predstavlja pogojni blok, sam zase ni logična celota, temveč postane del te šele, ko ga postavimo v kontekst krmilnega konstrukta. Varnost je s tem zagotovljena. Kako pa je z "logičnostjo" konstruktov? Povsem mogoče je, da nadomestimo pogojni blok s takim, ki pogoja preprosto ne preverja. V takem primeru se sledeča zanka ne bi nikoli zaključila:

```
Integer n = 10;
// shranimo zanko v spremenljivko
While myWhile =
    while { n > 0; } {
       (``n je '' + n).print();
       n--;
    }
   ...
// spremenimo pogojni blok
myWhile.setChild( myWhile.getChildren().at(0), { n++; } );
// izvri s spremenjenim pogojem
myWhile.exec();
```

Vsak blok namreč implicitno vrača vrednost `true` ali `false`. Ob izvedbi se privzeto vrne vrednost `true`, katero lahko pogojni izraz znotraj bloka spremeni, kot to ustreza pogoju. Na takšni zasnovi namreč temeljijo pogojni bloki v jeziku Zero. Ker nov pogojni blok v gornjem primeru nima pogojnega izraza, ki bi vračal logično vrednost `true` ali `false`, se vrne `true`, kar implicira, da je pogoj vedno izpolnjen. Logične pravilnosti posameznega konstrukta seveda ne preverjamo, saj je tudi prevajalnik ne preverja. Zato je možno zapisati sintaksno pravilen program, ki pa je logično napačen. Toda preprečevanje tega ostaja v domeni programerja.

Metarazred `Closure` ni osnova samo strukturalne temveč tudi behavioralne refleksije. Toda behavioralna refleksija je povezana še z metarazredom `Method`. Obnašanje konstrukta je moč manipulirati s funkcionalnostjo, katero podamo v obliki metode. Temu pravimo, da konstruktu metodo "pripnemo". Prikažimo to na primeru zanke *while*:

```
Integer n = 1;
// zanka while, ki izpise stevila od 1 do 16
While w =  while { n <= 16; } { n.toString().print(); n++; };
// dodaj akcijo
w.getChildren().at(0).addHandler(
                    method { (``pogoj pri n = '' + n).print(); }
                      );
```

Funkcionalnost, podano kot metodo, je mogoče dodati zaprtju. Ker gornja zanka sestoji iz dveh zaprtij, pogojnega bloka in telesa, lahko funkcionalnost dodamo kateremukoli teh dveh zaprtij. Točka, kjer je funkcionalnost konstrukta mogoče razširiti, je začetek zaprtja. To pomeni, da se dodana funkcionalnost izvede neposredno pred izvedbo zaprtja. V gornjem primeru pogojnemu bloku zanke dodamo funkcionalnost, ki izpiše vrednost spremenljivke `n` ob vsaki izvedbi pogojnega bloka. Funkcionalnost smo metodi `addHandler` podali kot argument v obliki anonimne prvorazredne metode. Anonimna metoda pa ni edina možnost, kako podati funkcionalnost. V primeru, kadar želimo z identično funkcionalnostjo razširiti obnašanje različnih konstruktov, se izplača funkcionalnost definirati kot poimenovano metodo ali uporabiti kar instančno metodo objekta.

Manipulacijo obnašanja zaprtja razširimo na metode. To je mogoče, saj je metoda pravzaprav le poimenovano zaprtje. Manipulacija obnašanja metode je tako dosežena z manipulacijo obnašanja vrhnjega zaprtja, tj. telesa metode. Do zaprtja, ki predstavlja telo metode, dostopamo preko metode `getClosure`, definirane v razredu `Method`. Toda tukaj se pojavi težava. Bloki v jeziku Zero ne prejmejo argumentov, ker jih ne moremo parametrizirati. V ta namen uporabljamo metode, katerih signaturo lahko preverimo že v času prevajanja. Rekli smo, da obnašanje metode razširimo z metodo, ki jo pripnemo na zaprtje oz. telo te metode. V kolikor metoda, katere obnašanje želimo razširiti, prejme argumente, mora obstajati nek način dostopa do teh argumentov tudi

znotraj metode, ki obnašanje razširja. Znotraj pripete metode morajo torej biti na voljo argumenti, podani originalni metodi. V ta namen postavimo zahtevo, da mora pripeta metoda, v kolikor jo pripenjamo na zaprtje, ki predstavlja telo neke metode, imeti identično signaturo kot metoda, na katero jo pripenjamo. To je treba preveriti v času izvajanja, saj v času prevajanja v splošnem ni moč ugotoviti ali je posamezno zaprtje telo metode ali zaprtje kakšnega nižjega nivoja. Prikažimo to na primeru razširitve obnašanja metode `append`, objekta `Vector`:

```
class MyApplication {
  restricted MyApplication {
    // seznam uporabnikov
    Vector users =  new Vector;
    // ustvari vizualne objekte
    ProgressBar progress = new ProgressBar;
    Label label = new Label;
    ...
    // razsiri funkcionalnost z anonimno metodo
    ( method users.append(Object)).getClosure().addHandler(
                      method(Object o) {
                          progress.advance();
                          label.setName(``Dodajam '' + o.toString());
                      }
    );
  }
}
```

Da bi metodi `append` pripeli dodatno funckionalnost, najprej to metodo "pridobimo" z rezervirano besedo **method** s polnim imenom metode in njeno signaturo, torej `users.append(Object)`. Nato pridobimo zaprtje te metodo z metodo `getClosure()`, na katerega pripnemo funkcionalnost, katero definiramo v obliki anonimne metode. Signatura anonimne metode je enaka signaturi metode `append`. Le tako je mogoče znotraj anonimne metode dostopati do argumenta tipa `Object`, ki je podan metodi `append`. V telesu anonimne metode dostopamo do spremenljivk, definiranih v starševskem okolju: `progress` ter `label`.

V gornjem primeru smo obnašanje metode razširili na instančnem nivoju. To pomeni, da se je funkcionalnost naše anonimne metode izvedla ob klicu metode `append` samo nad specifično instanco razreda `Vector`, nad tisto, ki smo jo poimenovali `users`. Refleksija jezika Zero pa posega tudi na razredni nivo. Ko bi želeli razširiti obnašanje metode `append` nad vsemi instancami razreda `Vector`, bi metodo pripeli na razrednem nivoju:

```
class MyApplication {
  restricted MyApplication {
    ...
    // pridobi razredno metodo append
    Method mAppend=users.getClass().getDeclaredMethod(‘‘append(Object)’’);
    // razsiri funkcionalnost z anonimno metodo
    mAppend.getClosure().addHandler(
                    method(Object o) {
                        progress.advance();
                        label.setName(‘‘Dodajam ’’ + o.toString());
                    }
    );
  }
}
```

Vse kar je potrebno, da bi metodo razširili na razrednem nivoju, je to, da pridobimo metodo iz razreda, v tem primeru iz razreda `Vector`. Nadaljnji postopek je identičen razširanju obnašanja instančne metode.

5.5 Metode višjega reda

Funkcije višjega reda so domena funkcijskih programskih jezikov kot so ML [67], LISP [64], Scheme [43]. Jezik Zero, kljub svoji imperativni naravi, omogoča uporabo tovrstnih funkcij na način, ki zagotavlja harmonijo z obstoječimi jezikovnimi konstrukti. Funkcije višjega reda so za naš jezik primerne zaradi enovite objektne predstavitve, v čemer se ne razlikujejo od drugih vrednosti. V jeziku Zero jih imenujemo *metode*

višjega reda. Metode višjega reda imenujemo tudi "curry" metode. V literaturi se običajno uporablja izraz "curry" funkcije.

5.5.1 Teoretične osnove

Teoretične začetke funkcij s področja računalništva in matematične logike najdemo v računu *lambda* (*lambda* calculus), ki sta ga v tridesetih letih zasnovala Alonzo Church in Stephen Kleene z namenom utemeljiti matematiko na funkcijah namesto množicah. Gre za formalno logiko, ki preučuje definicijo, aplikacijo in rekurzijo funkcij. Račun lambda sicer ni uspel odpraviti nekaterih protislovij, ki so se pojavljala pri množicah, a se je kljub temu uveljavil kot učinkovito orodje za obravnavo izračunljivih in rekurzivnih funkcij. Prav te funkcije so za teoretično računalništvo najbolj zanimive. Tako je funkcijsko programiranje skoraj v celoti zasnovano na računu lambda. Na račun lambda lahko gledamo podobno kot na Turingov stroj – oba lahko služita formalnemu zapisu algoritma. Medtem ko Turingov stroj uporablja imperativno logiko s stanjem, račun lambda uporablja čisto funkcijsko logiko, ki nima stanja in tudi ne stranskih učinkov.

5.5.2 Formalen zapis računa lambda

Račun lambda najlažje opišemo z abstraktno sintakso na sledeč način:

$$
\begin{array}{lll}
v \in Identifikator & & \\
e \in E & & izraz \\
e := v & & spremenljivka \\
\quad | \quad e_1 e_2 & & aplikacija \\
\quad | \quad \lambda v.e & & abstrakcija
\end{array}
$$

Aplikacija $e_1 e_2$ operira nad poljubnima lambda izrazoma e_1 in e_2 in pomeni klic funkcije e_1 z argumentom e_2. Abstrakcijo uporabimo za definicijo anonimnih funkcij. Vsaka spremenljivka računa lambda je prosta ali vezana. V izrazu $\lambda x.x + y$ je spremenljivka y prosta in spremenljivka x vezana. Semantiko računa lambda opišemo s pretvorbami:

- pretvorba α omogoča preimenovanje imen vezanih spremenljivk,
- pretvorba β ali *beta* redukcija pomeni aplikacijo funkcije. β redukcija izraza $(\lambda v.e_1)e_2$ je substitucija oblike $e_1[v := e_2]$,

- pretvorba η se uporablja pri enakosti dveh funkcij, ki velja, če funkciji data enak rezultat za vse argumente: $\lambda v.fv \Leftrightarrow f$, če spremenljivka v ni prosta v f.

5.5.3 Metode višjega reda v imperativnem jeziku

Kadar govorimo o metodah višjega reda, ne moremo mimo vprašanja "reda". Kaj pravzaprav pomeni red metode oz. funkcije? Metoda n-tega reda je metoda, ki prejme in vrača vrednosti reda $n - 1$. Metode prvega reda so torej "klasične" metode, ki prejemajo in vračajo podatke, ki sami po sebi niso metode. Metode višjega reda so torej vse tiste, za katere velja $n >= 2$. Metoda drugega reda po tej definiciji prejme ali vrne metodo prvega reda. Bistvena prednost uporabe tovrstnih metod je v višjem nivoju abstrakcije. Funkcijski programski jeziki kot npr. Lisp, so znani po tem, da izkoriščajo to moč abstrakcij na vsakem koraku. Programski kod, izražen z metodami višjega reda, je večkrat tudi laže razumljiv, saj so v njem metode uporabljene kot prvorazredne vrednosti. V jeziku Z_0 smo zasnovali objektne temelje, na katere v jeziku Zero postavljamo metode višjega reda. Ideja metod višjega reda temelji na dinamiki zaprtij. Zaprtje, ki vsebuje kod metode in njeno lokalno okolje referenc, je osnovni gradnik ne le metod, temveč tudi vseh krmilnih struktur. Metoda, ki je predstavljena kot zaprtje, ima karakteristike prvorazredne vrednosti. Za takšne vrednosti vemo, da jih lahko prenašamo v argumente, vračamo kot rezultat iz metod in jih shranjujemo v lokalne spremenljivke. Jezik Zero torej samo razširja koncept prvorazrednosti na metode. Provorazrednost krmilnih struktur je bila namreč realizirana že v jeziku Z_0.

Metode višjega reda omejujeta dva ključna koncepta jezika Zero – imperativna narava in statičen sistem tipov. "Imperativnost", ki za opis programa uporablja zaporedje stavkov s stranskimi učinki, je manj skladno z "deklarativnostjo" – idejnim funkcijskim programiranjem, ki program opisuje z matematičnimi funkcijami brez stanja in stranskih učinkov. Težava s statičnim tipiziranjem postane očitna pri izrazni moči jezika. Iz tega razloga je večina funkcijskih jezikov dinamično tipiziranih. Funkcijo lahko v dinamičnem sistemu tipov definiramo bolj univerzalno, kot če smo omejeni na katerikoli, še tako širok, tip. Tako definirano funkcijo lahko apliciramo na "poljuben" tip, ki zadošča funkcionalnosti definicije. Funkcijo drugega reda, ki vrne funkcijo, katera

sešteje dve vrednosti, lahko torej apliciramo na dve vrednosti poljubnih tipov, za katera je seštevanje smiselno. Kako to zapreko rešiti v statično tipiziranem jeziku? Ena od možnosti je, da metode izvzamemo iz statičnega tipiziranja in jih preverimo neposredno pred aplikacijo. To bi sicer delovalo, vendar bi vneslo precejšnjo mero neskladja v sistem tipiziranja. Kam bi konec koncev uvrstili jezik – med statično, dinamično ali hibridno tipizirane? Povrhu tega ima tak pristop vrsto implementacijskih težav, katere gotovo želimo zaobiti.
Enovitost objektnega modela smo postavili za temelj že v jeziku Z_0. To enovitost lahko ohranimo samo tako, da uporabljen princip tipiziranja prenesemo tudi na metode. Poudarimo, da so metode jezika Zero, abstraktno in strukturalno, že objekti. Potrebno je razširiti samo tipiziranje.
Pri metodah vseh redov gre za obravnavo funkcijskih (metodnih) tipov. Teoretične vidike funkcijskih podtipov smo obdelali v poglavju 2.3.3. Zdaj je potrebno to znanje prenesti v statičen sistem tipov objektno usmerjenega jezika. Metodni tipi morajo ohranjati koherenco z obstoječim tipiziranjem. K temu spadajo enovita razredna hierarhija s korenom v razredu `Object`, možnost dvosmerne pretvorbe med posameznimi metodnimi tipi in preverljivost v času prevajanja.

Predstavimo torej metodo kot objekt tipa `Method`. Razred `Method` je metarazred, ki opisuje posamezno metodo. Takoj se pojavi težava, kako predstaviti neskončno mnogo metodnih tipov s tipom, opisanim z enim samim razredom. Metodni tip je definiran s tipi parametrov in tipom vračanja. Oboji so lahko poljubni in jih je lahko poljubno mnogo. Tako bi vsak metodni tip moral imeti svoj razred za opis. Toda to bi bilo skrajno neučinkovito. Boljša rešitev je implicitna parametrizacija metarazreda `Method`. Tako lahko vsak metodni tip predstavimo s tem razredom in specifičnimi parametri. Razred pri tem ostane enak, objekt pa vsebuje parametre specifičnega metodnega tipa.

Strukturalno gledano je metoda parametriziran blok, ki temelji na zaprtju. Metodo lahko iz tega stališča obravnavamo enako kot bloke, katerih funkcionalnost je bila implementirana že v jeziku Z_0. Prav metode so razlog, zakaj jezik Zero ne omogoča posebnega konstrukta za parametrizirane bloke. Ko bi takšen konstrukt obstajal, bi

se v jeziku pojavila neortogonalnost, saj bi bila dva konstrukta za opis iste zadeve. Tako imamo neparametrizirane bloke, kadar za opis funkcionalnosti ne potrebujemo parametrov in prvorazredne metode, ko funkcionalnost zavisi od vrednosti parametrov. Tudi s stališča optimalnosti je smotrno ločiti metode od blokov. Slednji namreč niso tipizirani in so zato hitrejši.

5.5.4 Pretvorba med metodnimi tipi

Zaradi parametrizacije se pretvorba med posameznimi metodnimi tipi premakne z razredne na instančno raven. Metarazred `Method` opisuje vsak metodni tip, ne glede na tipe parametrov in tip vračanja. Metodne tipe bi lahko napravili povsem statično varne, tj. preverljive času prevajanja, vendar bi takšen pristop občutno zmanjšal izrazno moč tipiziranja metod in s tem celotnega jezika. Metodni tipi postanejo resnično uporabni šele z varianco. To pomeni, da lahko posamezen tip pretvorimo v drugega, če sta kompatibilna.

Formalno metodni tip zapišemo kot $(T_{a_1}, \ldots, T_{a_n}) \rightarrow (T_r)$, ki pomeni, da metoda slika iz tipov parametrov v tip rezultata. Za metode brez parametrov postavimo $n = 0$. Metode, ki ne vračajo rezultata, imajo $T_r = \epsilon$, kjer ϵ predstavlja "prazen" tip. Tipiziranje metod v času prevajanja temelji na njihovi signaturi. Metodni tip metode, ki vrne produkt realnega števila, v jeziku Zero zapišemo na sledeč način:

```
Method(Double) : Number m =
                        method(Double a) : Double {  return a*a; };
```

Desna stran deklaracije, **method**`(Double a) : Double`, predstavlja definicijo anonimne metode tipa $Double \rightarrow Double$, kar zadošča tipu $Double \rightarrow Number$, če je $Double <: Number$.

Če tip argumenta označimo s τ_a in tip vračanja s τ_r, lahko relacijo metodnega podtipa izrazimo kot:

$$\frac{\tau_a <: \tau'_a \qquad \tau'_r <: \tau_r}{\tau'_a \rightarrow \tau'_r \;<:\; \tau_a \rightarrow \tau_r}$$

S tem definiramo kovarianco funkcijskega operatorja v drugem argumentu ($\tau'_r <: \tau_r$) in kontravarianco v prvem ($\tau_a <: \tau'_a$). Metodni tip lahko seveda deklariramo rekurzivno,

kar pomeni, da imajo lahko tipi parametrov in vračanja tip metode poljubnega reda. Metodo `m` lahko pretvorimo v katerikoli drug metodni tip, ki ustrezno opiše metodo. Lahko jo pretvorimo v generično metodo tipa `Method` brez parametrov in tipa vračanja. Toda za izvedbo take metode je potrebna pretvorba nazaj v originalen tip.

5.5.5 Invokacija

Ena izmed idej metod višjega reda v jeziku Zero je transparentnost aplikacije. To ne pomeni nič drugega kot to, da je mogoče takšno metodo poklicati enako kot metodo prvega reda. Jeziki, v katerih so metode višjega reda bile dodane kot razširitev (C++, Java) imajo nedoslednosti prav pri aplikaciji. Potrebni so "posebni" mehanizmi ali konstrukti, ki omogočajo izvedbo takšnih metod. V jeziku Zero smo metode višjega reda načrtovali kot "osnovni" koncept oz. kot eno izmed primarnih lastnosti jezika. Metodo višjega reda lahko vedno apliciramo delno ali popolno. Delna aplikacija le vrne metodo, medtem ko polna aplikacija tudi aplicira vrnjeno metodo na argumente. Metoda višjega reda se v jeziku Zero ne razlikuje od metode prvega reda. Njena invokacija je identična, pokličemo jo kot metodno spremenljivko. Če metoda `getSquareMethod()` vrne metodo tipa, ki smo ga zapisali zgoraj $Double \rightarrow Number$, to metodo invociramo na sledeč način:

```
getSquareMethod()(3.3).toString().print();
```

Klic metode `getSquareMethod()` vrne metodo tipa $Double \rightarrow Number$, katero v istem koraku apliciramo na realni argument 3.3. Na tak način lahko aplikacijo metod verižimo do poljubnega nivoja. Metodo generičnega tipa `Method` lahko invociramo samo v primeru, kadar ne prejme nobenega argumenta. Sicer bi takšne metode ne mogli preveriti v času prevajanja. Toda tu se pojavi težava, ki je v času prevajanja nerešljiva. Generična metoda tipa `Method` lahko predstavlja katerokoli metodo in jo iz tega razloga lahko pretvorimo v katerikoli metodni tip. Kadar želimo metodo izvesti, potrebujemo popolno znanje o njeni signaturi. Metodo je pred invokacijo treba pretvoriti v pravilen tip, pri čemer sodeluje dinamičen sistem tipov. Tako je invokacija "objektiziranih" metod prvega in višjega reda tipizirana hibridno. Podobno zasledimo pri pretvorbi tipov v razredni hierarhiji od splošnega k specifičnemu tipu.

5.5.6 Primerjava z drugimi statično tipiziranimi jeziki

Jezik C++ v osnovi nima funkcij višjega reda. Mogoče je uporabljati kazalce na funkcije, vendar tega ne klasificiramo kot funkcije višjega reda. Težava kazalcev na funkcije je v tem, da ne omogočajo celovite obravnave zaprtij. "Kažejo" lahko samo na obstoječe definirane funckije. To onemogoča "prave" anonimne funkcije, ki so temelj funkcij višjega reda. Da bi v jeziku C++ bilo mogoče uporabljati funkcije višjega reda, je potrebno najprej rešiti težavo z zaprtji. Tak pristop je bil prikazan v [63]. Podobna rešitev je bila predlagana za Javo [73, 74]. Jezik $C^{\#}$ v ta namen uporablja delegatorje. Toda delegator je v bistvu le varen kazalec na funkcijo in ima kot tak precejšnje omejitve izrazne moči.

6 Zaključek

Temeljni cilj pričujočega raziskovalnega dela je bil zasnovati in implementirati metaprogramski model čistega objektno usmerjenega statično tipiziranega programskega jezika. Ta jezik smo poimenovali Zero. Ideja metaprogramskega modela, ki smo ga umestili v obstoječ jezik Z_0, je sposobnost dinamičnih sprememb na način, ki ohranja varnost tipiziranja. Večina mehanizmov, ki smo jih v okviru našega načrtovanja skušali izvesti, spada v domeno dinamično tipiziranih in interpretiranih jezikov. Sekundarni cilj snovanja metaprogramskega modela je bil njegova umestitev v obstoječ jezik tako, da bi se logično, tj. sintaktično in semantično zlil z obstoječimi koncepti jezika Z_0. Ker smo se možnosti metaprogramiranja zavedali že v času načrtovanja jezika Z_0, smo koncepte tega jezika zasnovali na način, združljiv s koncepti metaprogramiranja. Med drugim cilj raziskav seveda ni bil zgolj zasnovati nek teoretičen metaprogramski model, temveč tega tudi učinkovito implementirati. Po splošnem povzetku torej ne gre samo za nadgradnjo obstoječega jezika, temveč za celovito rešitev, utemeljeno in izgrajeno na jeziku Z_0.

Načrtovanje jezika Zero smo torej zasnovali v okviru implementiranih konceptov jezika Z_0. Konceptov jezika Z_0 nismo spreminjali, saj verjamemo, da so bili zasnovani racionalno glede zmožnosti statično tipiziranega jezika. V posameznih primerih smo koncept le dopolnili z upoštevanjem situacij, ki so se odprle šele v jeziku Zero. Eden takih konceptov je spreminjanje metode, pri katerem v jeziku Z_0 nismo mogli predvideti zapletov, ki so se pojavili šele z uvedbo anonimnih prvorazrednih metod v jeziku Zero. Tako so bile ponekod potrebne spremembe implementacijske narave, drugod le sintaktične ali semantične. Poudarimo, da je večina konceptov vendarle ostala nespremenjena. Tako je jezik Zero v celoti podedoval čisti objektni model, metodno predstavitev stanja objekta, sistem tipov ter shemo dedovanja.

V okviru snovanja metaprogramskih konceptov smo preučili številne objektno usmerjene programske jezike, ki omogočajo metaprogramiranje na tak ali drugačen način. Ob "temeljnih" jezikih, iz katerih je tehnika metaprogramiranja dejansko izšla, kamor spadajo Smalltalk in razni dialekti Lispa, smo obravnavali tudi jezike, ki so svoje me-

taprogramske modele dobili šele z razširitvami. Med temi smo se posebej osredotočili na Javo, ki jo, podobno kot naš jezik Zero, omejuje statičen sistem tipov. Iz tega stališča se nam je zdela takšna primerjava z našim jezikom primerna. V okviru javanskih razširitev smo obravnavali behavioralno in strukturalno refleksijo, katerih definicije smo podali v poglavju o metaprogramiranju in refleksiji. Ta razprava je imela pomemben vidik, saj smo metaprogramski model našega jezika zasnovali prav na teh dveh konceptih refleksije. Statično refleksijo smo zasnovali po zgledu Jave, medtem ko smo se pri dinamični zgledovali predvsem po dinamično tipiziranih jezikih in seveda tistih razširitvah Jave, ki omogočajo dinamične spremembe.

Kar zadeva jezikovne koncepte, ne moremo zatrditi, da smo načrtovali kaj povsem novega ali celo revolucionarnega. Namesto v zasnovo "novih" konceptov smo načrtovanje raje usmerili v združevanje težko združljivih konceptov, takih, ki so značilni za dinamično tipizirane jezike s tistimi, v domeni statično tipiziranih. Iz primerjave sorodnih del tega področja sklepamo, da so omenjeni koncepti v jeziku Zero združeni na naš način, originalni. Dokaz za to je, da ne obstaja noben javnosti predstavljen programski jezik, ki bi vključeval in smiselno združeval koncepte, kot je to realizirano v jeziku Zero. Obstajajo seveda jeziki, ki omogočajo še višjo stopnjo dinamičnih sprememb obnašanja in strukture programov, vendar so dinamično tipizirani in navadno tudi interpretirani. Na drugi strani obstajajo jeziki, katerih programi se izvajajo hitreje, vendar omogočajo neprimerno manj manipulacije, če sploh kakšno, v času izvajanja. Osnovno idejo jezika Zero pa smo postavili v prav to točko razpotja – omogočiti visoko stopnjo manipulacije programa v času izvajanja in hkrati ostati pod okriljem statičnega ali vsaj hibridnega tipiziranja. Zahteva, ki smo ji v fazi načrtovanja jezika posvetili veliko pozornosti in preučevanja, je umestitev metaprogramskega modela v obstoječ jezik. S primerjavo razširitev Jave z metaprogramskimi koncepti smo ugotovili, da imajo vse rešitve neljubo lastnost, da se namreč slabo skladajo z obstoječimi jezikovnimi koncepti, bodisi na sintaktični in semantični ravni bodisi da sploh ne šedejo" v jezik. To je problem razširitev jezika, ki v osnovi ni bil načrtovan za take podvige. Metaprogramski koncepti realizirani kot razširitev jezika se vedno rešujejo preko bolj ali manj posrečenih programskih vmesnikov in ogrodij. Tovrstne rešitve pa s seboj vselej prinašajo neko

okornost uporabe. Mnogokrat je, kot kažejo preučene študije literature, potrebno že za najmanjšo spremembo napisati številne vrstice programskega koda, ki program komplicirajo in ga delajo nepreglednega. Prav temu smo se v jeziku Zero želeli izogniti z načrtovanjem in vpeljavo takega metaprogramskega modela, ki bi ga bilo mogoče združiti s koncepti obstoječega jezika. Ta model smo zasnovali tako, da se metaprogramiranje v jeziku Zero "zlije" s klasičnim programiranjem, tako da se meja med obema zabriše. Del tega je omogočal že jezik Z_0 s svojim nadomeščanjem metod, kar je v bistvu bila implicitna metaoperacija. Jezik Zero tako nima nikakršnih specialnih konstruktov metaprogramiranja, vsa metafunkcionalnost se rešuje preko metarazredov. To nas je pripeljalo do dveh momentov metaprogramiranja v jeziku Zero – implicitnega in eksplicitnega. Implicitno metaprogramiranje je, kot že rečeno, realiziral že jezik Z_0 z možnostjo nadomeščanja instančnih metod. Eksplicitno metaprogramiranje prinaša jezik Zero in pomeni zgolj uporabo metarazredov za manipulacijo strukture in obnašanja na razrednem in instančnem nivoju. Eksplicitno metaprogramiranje prinaša jeziku seveda bistveno večjo moč manipulacije, kot je bila dosegljiva v jeziku Z_0.

Metaprogramski model jezika Zero smo zasnovali na osnovi refleksije. Načrtovanja le-te smo se lotili na podlagi dognanj podobnih del iz literature. Refleksijsko funkcionalnost smo logično razdelili na dva dela: statično in dinamično, pri čemer prva pomeni le pridobivanje informacij o tekočem programu, druga pa spreminjanje le-tega v času izvajanja. Dinamično refleksijo smo dalje razmejili na behavioralno ter strukturalno. Behavioralna refleksija omogoča manipulacijo obnašanja, strukturalna manipulacijo strukture – t.j. vpogled v strukturo in njeno spreminjanje. Metaprogramski model jezika Zero temelji na obeh vrstah refleksije. Behavioralno refleksijo jezika smo realizirali s koncepti, podobnimi tistim, na katerih temelji aspektno usmerjeno programiranje. Funkcionalnost, ki spreminja obnašanje konstrukta, je v jeziku Zero realizirana v obliki metode. Z implementacijskega stališča se sprememba obnašanja tako prevede na spremembo funkcionalnosti. Točka razširitve funkcionalnosti je v našem jeziku vedno zaprtje, kar omogoča manipulacijo obnašanja krmilnih konstruktov kot tudi metod, saj so tudi te sestavljene iz zaprtij. Tekom načrtovanja smo se odločili točko razširitve postaviti pred izvršitev zaprtja. V primerjavi z aspektno usmerjenim programiranjem

pomeni takšna fiksacija omejitev, vendar smo arhitekturo zasnovali tako, da je nove točke razširitve mogoče dodati na enostaven način. Potrebno bi bilo le dodelati metode behavioralne refleksije metarazreda `Closure` in dopolniti virtualni stroj.
Strukturalno refleksijo jezika smo realizirali na osnovi čistega objektnega modela jezika. Že v jeziku Z_0 so bile vse entitete, tudi krmilni konstrukti, predstavljene uniformno kot objekt. Ta čisti objektni model, ki smo ga povzeli po jeziku Smalltalk, nam je omogočil relativno enostavno razširitev jezika s konceptom strukturalne refleksije. Manipulacija strukture je mogoča na nivoju zaprtij. To pomeni, da je možno vsako zaprtje v programu nadomestiti. Ker so krmilni konstrukti sami sestavljeni iz zaprtij, se možnost take manipulacije razširi na vse konstrukte. Grobo rečeno gre s stališča implementacije za nadomeščanje referenc v posameznih objektih. Seveda je potrebno pri tem upoštevati še ohranjanje varnosti tipov in pravilne strukture programa. Strukturalna refleksija jezika Zero nima mehanizma preverjanja logične strukture manipuliranega programa; to ostaja v domeni programerja.
Celoten metaprogramski model jezika smo zasnovali na ideji dvoslojne manipulacije. Model, ki smo ga implementirali, namreč omogoča manipulacijo na nivoju razreda kot tudi instance. To ponuja širše možnosti aplikacije metaprogramiranja, saj je mogoče manipulacijo aplicirati povsem lokalno na specifično instanco, ali pa na vse instance nekega razreda. Ta princip velja tako za behavioralne kot strukturalne spremembe.

Metaprogramski model jezika Zero temelji na metarazredih. Podobno zasnovo najdemo v metaobjektnem protokolu. V metaobjektnem protokolu ima vsak objekt pripadajoč metarazred, medtem ko v jeziku Zero temu ni tako. Kljub modelu zasnovanem na metarazredih, ne gre torej za čisti metaobjektni protokol. Namesto, da bi vsakemu objektu pripisali metarazred, smo načrtovali le tri metarazrede. Ti so `Class`, `Closure` in `Method`. Prvi omogoča metafunkcionalnost na nivoju razreda, drugi metafunkcionalnost krmilnih konstruktov in tretji metod. Razreda `Closure` in `Method` smo načrtovali kot implicitna metarazreda, katerih funkcionalnosti se dedujeta v krmilnih konstruktih in metodah. Razred `Class` je eksplicitni metarazred, do katerega dostopamo s posebno metodo.
Razlog za takšno zasnovo metarazredov je v cilju jezika Zero, ki želi metaprogramira-

nje agregirati s klasičnim programiranjem. To je mogoče prav z implicitno dedovanimi metarazredi in njihovo dobro zasnovano funkcionalnostjo. Metode metarazredov so dejansko tiste, ki omogočajo strukturalno in behavioralno refleksijo. Od tega rahlo odstopa le razred `Method`, ki za ta namen nima lastnih metod, saj namreč tudi metode temeljijo na zaprtjih, katerih refleksijo obravnava razred `Closure`.
Strukturalne in behavioralne refleksijske zmožnosti jezika smo prikazali na praktičnih primerih, kjer smo se najprej lotili dekompozicije programa na posamezna zaprtja in nato izvajali manipulacijo teh zaprtij z metodami metarazredov.

Čisti objektni model jezika Z_0 smo v jeziku Zero razširili na metode, ki so zdaj prvorazredne vrednosti. Implementirali smo selektivno predstavitev metod kot objektov, kar pomeni, da jih predstavljamo kot objekte samo, kadar je to potrebno. Na ta način se izognemo nepotrebni potrati pomnilniškega prostora. Prvorazrednost metod je pomembna s stališča izrazne moči jezika, saj lahko ob taki predstavitvi metode tretiramo kot shranljive vrednosti in z njimi manipuliramo kot s spremenljivkami. Prvorazrednost metod je pravtako pomemben temelj, na katerem smo zgradili koncept anonimnih metod. Teh jezik Z_0 ni poznal. Anonimne metode igrajo pomembno vlogo pri manipulaciji metod v okviru behavioralne refleksije. Hkrati imajo anonimne metode veliko praktično vrednost, saj je nova funkcionalnost, ki jo dobi razredna ali instančna metoda, v večini primerov definirana ravno v obliki anonimne metode. Z namenom poenostaviti uporabo takih metod, smo vpeljali idejo invokacije metode preko metodne spremenljivke. Ta ni nič drugega kot spremenljivka metodnega tipa, katero lahko "izvršimo" tako, da ji podamo seznam argumentov. Tako poteka klic shranjene metode identično, kot če bi klicali instančno metodo.
Ker je jezik Zero statično tipiziran, smo uvedli metode tipe, ki jih teorija imenuje funkcijske tipe. Metodni tipi so variante generičnega metodnega tipa, opisanega z metarazredom `Method`. Parametrizacija razreda `Method` je bila nujna, saj je metodnih tipov v splošnem poljubno mnogo. Metodne tipe smo zasnovali tako, da bi bili kar najlaže berljivi in razumljivi. Seveda pa lahko zapletenost tipa naraste z vpeljavo metod višjih redov, kjer se pojavljajo metodni tipi, ki vračajo druge metodne tipe itn. Za metodne tipe velja hibridno tipiziranje, tj. preverjanje v času prevajanja, kadar pa

to ni mogoče, pa v času izvajanja. Slednje zavzema večji del tipiziranja, saj večina metod, ki operira nad argumenti metodnih tipov, predpostavlja generičen metodni tip. Z vpeljavo metod kot prvorazrednih vrednosti, anonimnih metod ter metodnih tipov smo se v jeziku Zero, ki je sicer imperativne narave, približali deklarativnemu načinu izražanja. Zavedamo se, da metode višjega reda ne dosegajo izrazne moči dinamično tipiziranih jezikov, saj zaradi tipov preprosto ne morejo dosegati stopnje univerzalnosti funkcijskih jezikov. Kljub temu verjamemo, da smo uspeli metode višjega reda v naš jezik umestiti na način, ki je združljiv z imperativnim programirnim vzorcem.

Z razvojem vseh omenjenih konceptov jezika smo uspeli pokazati veljavnost na začetku postavljenih hipotez glede primernosti statičnega sistema tipov kot primerne sheme za razvoj metaprogramskega modela. Pokazali smo tudi, da tak sistem, kolikor želi ohraniti varnost tipiziranja, zahteva kompatibilnost signatur. Nenazadnje pa smo s konkretno implementacijo metakonceptov programskega jezika pokazali, da je metaprogramski model behavioralne in strukturalne refleksije povsem združljiv s statično tipiziranim jezikom.

V jezik Zero smo vključili moderne koncepte programskih jezikov. To pa ne pomeni, da je jezik s tem izpopolnjen. Nasprotno, obstaja še mnogo konceptov, ki so se tekom razvoja programskih jezikov izkazali za nadvse uporabne. Omenimo samo parametrični polimorfizem, ki ga naš jezik v trenutnem stanju ne omogoča. Jezik Zero vsebuje samo en tip, ki je zmožen parametrizacije – `Method`. Možnost parametrizacije tipov je pomembna, saj je s tem omogočeno bistveno fleksibilnejše tipiziranje, ki ima pomembno lastnost – da se ga namreč preveriti v času prevajanja. Parametrizacija tako omogoča visoko stopnjo ponovne uporabnosti in učinkovitost izvajanja. Drug pomemben koncept, ki ga jezik Zero ne omogoča, je čista abstrakcija tipa. Java v ta namen ponuja konstrukt *interface*. Ločitev tipa od implementacije igra pomembno vlogo pri načrtovanju aplikacij, saj omogoča višjo stopnjo abstrahiranja in s tem razumljivejši in čistejši objektni model aplikacije.

Seveda je poleg povsem novih konceptov zanimiva tudi dodelava in izboljšanje ob-

stoječih. Tukaj imamo v mislih predvsem koncepte behavioralne in strukturalne refleksije. Behavioralni refleksiji bi povečali moč z dodatnimi točkami razširitve po vzoru aspektno usmerjenega programiranja. Kar zadeva refleksijo zaprtij, bi bil na mestu mehanizem predpomnjenja, ki bi zmanjšal čas, potreben za ustvarjanje zaprtij in s tem čas celotnega izvajanja programa. Povejmo, da je jezik v svoji trenutni obliki primeren za nadaljnje raziskovanje. Za industrijsko rabo bi bilo po našem prepričanju potrebno dodelati predvsem virtualni stroj in upravljanje s pomnilnikom.

Literatura

[1] The Open Closed Principle
`http://www.eventhelix.com/realtimemantra/open_closed_principle.htm`.

[2] Martín Abadi. Baby modula-3 and a theory of objects. *Journal of Functional Programming*, let. 4, t. 2, strani 249–283, 1994.

[3] Martín Abadi and Luca Cardelli. An imperative object calculus: Basic typing and soundness. V *SIPL '95 - Proc. Second ACM SIGPLAN Workshop on State in Programming Languages*. Technical Report UIUCDCS-R-95-1900, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[4] Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. V *Conf. Record 20th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'93, Charleston, SC, USA, 10–13 Jan 1993*, strani 157–170. ACM Press, New York, 1993.

[5] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. V *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, strani 396–409, 1996.

[6] Ole Agesen, Jens Palsberg, and Michael Schwartzbach. Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. V *Proceedings of ECOOP '93*, strani 247–267, Springer-Verlag, 1993.

[7] Davide Ancona and Elena Zucca. An algebra of mixin modules. V *Workshop on Algebraic Development Techniques*, strani 92–106, 1997.

[8] Davide Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, let. 8, t. 4, strani 401–446, 1998.

[9] Davide Ancona and Elena Zucca. A theory of mixin modules: Algebraic laws and reduction semantics. Technical Report ISI-TR-99-05, Dipartimento di Informatica e Scienze dell'Informazione, Universita di Genova, 1999.

[10] Andrew W. Appel. *Modern compiler implementation in Java*. Cambridge Universisty Press, 1997.

[11] Giuseppe Attardi and Antonio Cisternino. Reflection support by means of template metaprogramming. V *GCSE '01: Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, strani 118–127, London, UK, Springer-Verlag, 2001.

[12] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, let. 35, t. 2, strani 97–113, 2003.

[13] Gerald Baumgartner and Vincent F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software–Practice & Experience*, let. 25, t. 8, strani 863–889, 1995.

[14] Gerald Baumgartner and Vincent F. Russo. Implementing signatures for C++. *ACM Transactions on Programming Languages and Systems*, let. 19, t. 1, strani 153–187, 1997.

[15] Lorenzo Bettini, Michele Loreti, and Betti Venneri. On multiple inheritance in Java, 2002.

[16] Robert Biddle, Angela Martin, and James Noble. No name: just notes on software reuse. *ACM SIGPLAN Notices*, let. 38, t. 12, strani 76–96, 2003.

[17] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. GJ: Extending the java programming language with type parameters, 1998.

[18] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. V *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, strani 183–200, Vancouver, BC, 1998.

[19] Kim B. Bruce. *Foundations of Object-Oriented Languages, Types and Semantics*. The MIT Press, Cambridge, Massachusetts, 2002.

[20] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. *Lecture Notes in Computer Science*, t. 952, strani 27–51, 1995.

[21] Carlos Camarao and Luclia Figueiredo. Class types, 1999.

[22] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded quantification for object-oriented programming. V *Proceedings of Functional Programming and Computer Architecture*, strani 273–280, 1989.

[23] Luca Cardelli. A polymorphic lambda calculus with type:type. Technical Report 10, Palo Alto, California, 1986.

[24] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, let. 76, t. 2/3, strani 138–164, 1988.

[25] Luca Cardelli and John C. Mitchell. Operations on records. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, strani 295–350. The MIT Press, Cambridge, MA, 1994.

[26] Martin D. Carroll. Metaprogramming and c++. *J. Prog. Lang.*, let. 4, t. 1, strani 1–20, 1996.

[27] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. V *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, t. 24, strani 49–70, New York, NY, 1989.

[28] Shigeru Chiba. A metaobject protocol for C++. *SIGPLAN Not.*, 30(10):285–299, 1995.

[29] Shigeru Chiba. Load-time Structural Reflection in Java. V *Lecture Notes in Computer Science*, t. 1850, strani 313–336, 2000.

[30] Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic program transformation with JOIE. V *ATEC'98: Proceedings of the Annual Technical Conference on USENIX Annual Technical Conference, 1998*, strani 176–178, Berkeley, CA, USA, USENIX Association, 1998.

[31] Pierre Cointe. Metaclasses are first class: The objvlisp model. *SIGPLAN Not.*, let. 22, t. 12, 156–162, 1987.

[32] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. V *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, t. 24, strani 433–444, New York, NY, 1989.

[33] Markus Dahm. Byte code engineering. V *Java-Informations-Tage*, strani 267–277, 1999.

[34] Rowan Davies and Frank Pfenning. Practical refinement-type checking, 1997.

[35] Marcus Denker, Stephane Ducasse, and Eric Tanter. Runtime bytecode transformation for Smalltalk. V *Computer Languages, Systems & Structures*, let. 32, t. 2/3, strani 125–139, 2006.

[36] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. V *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, strani 331–347, ACM, New York, NY, 1984.

[37] Karel Johannes Lodewijk Driesen. Software and hardware techniques for efficient polymorphic calls. Technical Report TRCS99-24, University of California, 1999.

[38] Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. *Lecture Notes in Computer Science*, t. 1241, strani 389–429, 1997.

[39] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the java type system sound? *Theory and Practice of Object Systems*, let. 5, t. 1, strani 3–24, 1999.

[40] Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach. Java type soundness revisited, 1997.

[41] Catherine Dubois and Valerie Menissier-morain. Certification of a type inference tool for ML: Damas-milner within coq. *Journal of Automated Reasoning*, let. 23, t. 3/4, strani 319–346, 1999.

[42] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, let. 31, t. 6, strani 262–273, 1996.

[43] R. Kent Dybvig. *The Scheme Programming Language, Second Edition*. Prentice Hall, NJ, 1996.

[44] Anton Eliens. *Principles of Object-Oriented Software Development*. Addison-Wesley Publishing Company, 1994.

[45] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: spatial and temporal selection of reification. *SIGPLAN Not.*, let. 38, t.11, strani 27–46, 2003.

[46] Erik Ernst. Dynamic inheritance in a statically typed language. *Nordic Journal of Computing*, let. 6, t. 1, strani 72–92, 1999.

[47] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. V *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, strani 171–183, New York, NY, 1998.

[48] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. *ACM SIGPLAN Notices*, let. 38, t. 11, strani 115–134, 2003.

[49] Adele Goldberg and David Robson. *Smalltalk-80 The Language and its Implementation*. Addison-Wesley Publishing Company, 1983.

[50] Michael Golm and Jürgen Kleinöder. Jumping to the Meta Level: Behavioral Reflection Can Be Fast and Flexible. *Lecture Notes in Computer Science*, t. 1616, strani 22–39, 1999.

[51] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[52] Kris Gybels, Roel Wuyts, Stephane Ducasse, and Maja D'Hondt. Inter-language reflection: A conceptual model and its implementation. *Computer Languages, Systems & Structures*, let. 32, t. 2/3, strani 109–124, 2006.

[53] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *European Symposium on Programming*, strani 6–20, 2002.

[54] Stanley Jefferson and Daniel P. Friedman. A simple reflective interpreter. *Lisp and Symbolic Computation*, let. 9, t. 2/3, strani 181–202, 1996.

[55] Andrew Kennedy and Don Syme. Design and implementation of generics for the .net common language runtime. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, Snowbird, Utah, 2001.

[56] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[57] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, t. 2072. strani 327–355, 2001.

[58] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *Proceedings European Conference on Object-Oriented Programming*, t. 1241, strani 220–242, 1997.

[59] Graham Kirby, Ron Morrison, and David Stemple. Linguistic Reflection in Java. *Software Practice and Experience*, let. 28, t. 10, strani 1045–1077, 1998.

[60] Bjoern Kirkerud. *Object-Oriented Programming with SIMULA*. Addison-Wesley Publishing Co., 1989.

[61] Guenter Kniesel. Implementation of dynamic delegation in strongly typed inheritance-based systems. Technical Report IAI-TR-94-3, Computer Science Department III, University of Bonn, 1994.

[62] Wilf R. LaLonde and John R. Pugh. *Inside Smalltalk Volume I*. Prentice-Hall Internation, Inc., 1990.

[63] Konstantin Läufer. A framework for higher-order functions in c++. V *COOTS'95: Proceedings of the USENIX Conference on Object-Oriented Technologies on USENIX Conference on Object-Oriented Technologies (COOTS)*, strani 8–8, USENIX Association, Berkeley, CA, USA, 1995.

[64] Jo A. Lawless and Molly M. Miller. *Understanding CLOS: The Common LISP Object System*. Digital Press, 1991.

[65] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, let. 1, t. 3, strani 221–242, 1995.

[66] Cristina Videira Lopes, Paul Dourish, David H. Lorenz, and Karl Lieberherr. Beyond aop: toward naturalistic programming. *ACM SIGPLAN Notices*, let. 38, t. 12, strani 34–43, 2003.

[67] Robin Milner, Mads Tofte, Robert Harper, David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

[68] Pattie Maes. Concepts and experiments in computational reflection. *SIGPLAN Not.*, let. 22, t. 12, strani 147–155, 1987.

[69] Luca Cardelli Martín Abadi. A theory of primitive objects: Second-order systems. V *In Proceedings of the European Symposium on Programming. Lecture Notes in Computer Science*, t. 788, strani 1–25. Springer-Verlag, 1994.

[70] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 1988.

[71] Stephan A. Missura. Theories = signatures + propositions used as types. *Lecture Notes in Computer Science*, t. 958. strani 144–155, 1994.

[72] Stephan Murer, Jerome A. Feldman, Chu-Cheow Lim, and Martina-Maria Seidel. pSather: Layered extensions to an object-oriented language for efficient parallel computation. Technical Report TR-93-028, Berkeley, CA, 1993.

[73] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. V *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, strani 146–159. ACM Press, New York (NY), USA, 1997.

[74] Martin Odersky and Philip Wadler. Leftover curry and reheated pizza: How functional programming nourishes software reuse. V *Fifth International Conference on Software Reuse*, IEEE, Vancouver, BC, 1998.

[75] John C. Mitchell Ole Agesen, Stephen N. Freund. Adding type parameterization to the Java language. V *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, strani 49–65, Atlanta, GA, 1997.

[76] Francisco Ortin and Juan Manuel Cueva. Non-restrictive computational reflection. *Comput. Stand. Interfaces*, 25(3):241–251, 2003.

[77] Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. V *International Conference on Typed Lambda Calculi and Applications*, number 664, strani 361–375, Utrecht, The Netherlands, 1993.

[78] Rob Pooley. *An Introduction to Programming in SIMULA*. Blackwell Scientific Publications, Oxford, 1987.

[79] Gilles Roussel R emi Forax Etienne Duris. Java multi-method framework. V *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00)*, 2000.

[80] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. V *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, strani 205–230, Springer-Verlag, London, UK, 2002.

[81] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Comput. Lang. Syst. Struct.*, let. 34, t. 2/3, strani 46–65, 2008.

[82] Sašo Greiner, Janez Brest, Viljem Žumer. Zero–a blend of static typing and dynamic metaprogramming. *Comput. Lang. Syst. Struct., Na spletu 12 April 2008*, 2008.

[83] Daniel Simon and Andy Walter. An implementation of the programming language Dml in Java, 2000.

[84] Anthony J. H. Simons. Let's agree on the meaning of 'class'. Department of Computer Science, University of Sheffield, 1996.

[85] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. V *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, strani 550–570. Springer-Verlag LNCS 1445, 1998.

[86] Yannis Smaragdakis and Don Batory. Mixin-based programming in C++. *Lecture Notes in Computer Science*, t. 2177, strani 163–178, 2001.

[87] Yannis Smaragdakis and Don S. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *Software Engineering and Methodology*, let. 11, t. 2, strani 215–255, 2002.

[88] Alan Snyder. Commonobjects: an overview. V *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, t. 21, strani 19–28. ACM Press New York, NY, USA, 1986.

[89] Sašo Greiner. *Implementacija dinamičnih konceptov čistega statičnega objektno usmerjenega jezika, magistrsko delo, FERI, Maribor.* 2004.

[90] J. Sobel and D. Friedman. An introduction to reflection-oriented programming, 1996.

[91] Diomidis Spinellis. Rational metaprogramming. *IEEE Softw.*, let. 25, t. 1, strani 78–79, 2008.

[92] Guy Steele. Common lisp: The language. *Digital Equipment Corporation*, 1984.

[93] Bjarne Stroustrup. Multiple inheritance for C++. V *Proceedings of the Spring 1987 European Unix Users Group Conference*, Helsinki, 1987.

[94] Clemens Szypersky, Stephen Omohundro, and Stephan Murer. Engineering a programming language: The type and class system of sather. Technical Report TR-93-064, Berkeley, CA, 1993.

[95] Antero Taivalsaari. Kevo – a prototype-based object-oriented language based on concatenation and module operartions. Technical Report DCS–197–1R, University of Victoria, 1992.

[96] Antero Taivalsaari. Object-oriented programming with models. V *Journal of Object-Oriented Programming*, t. 6, strani 25–32, 1993.

[97] Antero Taivalsaari. Classes vs. prototypes - some philosophical and historical observations, 1996.

[98] Antero Taivalsaari. On the notion of inheritance. In *ACM Computing Surveys*, t. 28, strani 439–477. ACM Press, 1996.

[99] Eric Tanter, Noury Bouraqadi, and Jacques Noyé. Reflex – Towards an Open Reflective Extension of Java. V *Proceedings of the Third International Conference on Metalevel Architectures and Advanced Separation of Concerns*, *Lecture Notes in Computer Science*, t. 2192, strani 25–43, Springer-Verlag, Kyoto, Japan, September 2001.

[100] Eric Tanter, Marc Ségura-Devillechaise, Noyé Jacques, and José Piquer. Altering Java Semantics via Bytecode Manipulation. V *Proceedings of the 1st ACM*

SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002), *Lecture Notes in Computer Science*, t. 2487, strani 283–298, Springer-Verlag, Pittsburgh, PA, USA, 2002.

[101] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. Openjava: A Class-Based Macro System for Java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, strani 117–133, Springer-Verlag, London, UK, 2000.

[102] David Ungar and Randall B. Smith. Self: The power of simplicity. *OOPSLA'87*, let. 4, t. 8, strani 227–242, 1987.

[103] John Viega, Bill Tutt, and Reimer Behrends. Automated delegation is a viable alternative to multiple inheritance in class based languages. Technical Report CS-98-03, University of Virginia, 1998.

[104] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. *ACM SIGPLAN Notices*, let. 35, t. 10, strani 146–165, 2000.

[105] Jan Vitek and R. Nigel Horspool. Taming message passing: Efficient method look-up for dynamically typed languages. *Lecture Notes in Computer Science*, t. 821, 1994.

[106] Ian Welch and Robert J. Stroud. Kava - using byte code rewriting to add behavioural reflection to Java. V *COOTS'01: Proceedings of the 6th conference on USENIX Conference on Object-Oriented Technologies and Systems*, strani 119–130, USENIX Association, Berkeley, CA, USA, 2001.

[107] Andrew Wright. Practical soft typing. Technical Report TR94-236, Rice University, 1998.

[108] Zhixue Wu. Reflective java and a reflective component-based transaction architecture. V *OOPSLA '98 Workshop on Reflective Programming in C++ and Java*, 1998.

[109] Dachuan Yu, Andrew Kennedy, and Don Syme. Formalization of generics for the .net common language runtime. *ACM SIGPLAN Notices*, let. 39, t. 1, strani 39–51, 2004.

Bibliografija

1.01 Izvirni znanstveni članek

1. GREINER, Sašo, TUTEK, Simon, BREST, Janez, ŽUMER, Viljem. Quick adaptation of web-based information systems with aspect-oriented features. *CIT. J. Comput. Inf. Technol.*, 2004, let. 12, t. 2, strani 103–109. [COBISS.SI-ID 8911382]

2. GREINER, Sašo, BREST, Janez, ŽUMER, Viljem. Načrtovanje porazdeljene arhitekture za simultano izvajanje programskih bremen = Designing distributed architecture for concurrent program execution. *Elektroteh. vestn.*, 2005, let. 72, t. 2–3, strani 91-96. [COBISS.SI-ID 9770262]

3. GREINER, Sašo, REBERNAK, Damijan, BREST, Janez, ŽUMER, Viljem. Z0 - a tiny experimental language. *SIGPLAN not.*, 2005, let. 40, t. 8, strani 19–28. [COBISS.SI-ID 9879574]
JCR IF: 0.175, SE (76/79), computer science, software engineering, x: 0.937

4. BREST, Janez, GREINER, Sašo, BOŠKOVIĆ, Borko, MERNIK, Marjan, ŽUMER, Viljem. Self-adapting control parameters in differential evolution: a comparative study on numerical benchmark problems. *IEEE trans. evol. comput.*. [Print ed.], 2006, let. 10, t. 6, strani 646–657. [COBISS.SI-ID 10376982]
JCR IF: 3.77, SE (3/85), computer science, artificial intelligence, x: 1.251, SE (2/75), computer science, theory & methods, x: 1.001

5. BREST, Janez, BOŠKOVIĆ, Borko, GREINER, Sašo, ŽUMER, Viljem, SEPESY MAUČEC, Mirjam. Performance comparison of self-adaptive and adaptive differential evolution algorithms. *Soft computing*. [Tiskana izd.], 2007, let. 11, t. 7, strani 617–629. [COBISS.SI-ID 11150358]
JCR IF (2006): 0.516, SE (66/85), computer science, artificial intelligence, x: 1.251, SE (69/87), computer science, interdisciplinary applications, x: 1.142

6. GREINER, Sašo, BREST, Janez, ŽUMER, Viljem. Zero - A blend of static typing and

dynamic metaprogramming. *Comput. syst. struct.*, Na spletu, 12 April 2008, 11 strani [COBISS.SI-ID 12284694]
JCR IF (2006): 0.591, SE (57/82), computer science, software engineering, x: 0.937

1.03 Kratki znanstveni prispevek

7. BOŠKOVIĆ, Borko, ZAMUDA, Aleš, BREST, Janez, GREINER, Sašo, ŽUMER, Viljem. An Opposition-based differential evolution with adaptive mechanism, applied to the tuning of a chess evaluation function. *Journal of computational intelligence*, 2008, let. 1, t. 1, strani 1–6. [COBISS.SI-ID 12127766]

8. BREST, Janez, ZAMUDA, Aleš, BOŠKOVIĆ, Borko, GREINER, Sašo, ŽUMER, Viljem. An analysis of the control parameters adaptation in the differential evolution algorithm. *Journal of computational intelligence*, 2008, let. 1, t. 1, strani 7–22. [COBISS.SI-ID 12128022]

1.08 Objavljeni znanstveni prispevek na konferenci

9. GREINER, Sašo, BOŠKOVIĆ, Borko, BREST, Janez, ŽUMER, Viljem. Security issues in information systems based on open-source technologies. V: ZAJC, Baldomir (ur.), TKALČIČ, Marko (ur.). *The IEEE Region 8 EUROCON 2003 : computer as a tool : 22–24. September 2003, Faculty of Electrical Engineering, University of Ljubljana, Ljubljana, Slovenia : proceedings*. Piscataway: IEEE, cop. 2003, let. B, strani 449-453. [COBISS.SI-ID 8241174]

10. GREINER, Sašo, REBERNAK, Damijan, BOŠKOVIĆ, Borko, BREST, Janez, ŽUMER, Viljem. Web information system based on open source technologies. V: BUDIN, Leo (ur.), LUŽAR - STIFFLER, Vesna (ur.), BEKIĆ, Zoran (ur.), HLJUZ DOBRIĆ, Vesna (ur.). 25th International Conference on Information Technology Interfaces, June 16-19, 2003, Cavtat, Croatia. *ITI 2003 : proceedings of the 25th International Conference on Information Technology Interfaces, June 16-19, 2003, Cavtat, Croatia*. Zagreb: University of Zagreb, SRCE University Computing Centre, 2003, strani 137–142. [COBISS.SI-ID 7991318]

11. BREST, Janez, ROŠKAR, Silvo, BOŠKOVIĆ, Borko, REBERNAK, Damijan, GREINER, Sašo, KRUŠEC, Robert, ŽUMER, Viljem. Informacijski sistem v proizvodnem procesu. V: ZAJC, Baldomir (ur.). *Zbornik dvanajste mednarodne Elektrotehniške in računalniške konference ERK 2003, 25. - 26. september 2003, Ljubljana, Slovenija*, (Zbornik ... Elektrotehniške in računalniške konference ERK ..., 1581–4572). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2003, strani 365-368. [COBISS.SI-ID 8229910]

12. GREINER, Sašo, TUTEK, Simon, BREST, Janez, ŽUMER, Viljem. Razvojna paradigma MVC v spletnih aplikacijah. V: ZAJC, Baldomir (ur.). *Zbornik dvanajste mednarodne Elektrotehniške in računalniške konference ERK 2003, 25. - 26. september 2003, Ljubljana, Slovenija*, (Zbornik ... Elektrotehniške in računalniške konference ERK ..., 1581-4572). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2003, strani 393–396. [COBISS.SI-ID 8232982]

13. BOŠKOVIĆ, Borko, REBERNAK, Damijan, GREINER, Sašo, BREST, Janez, ŽUMER, Viljem. Nadzorovanje virov računalniškega sistema v operacijskem sistemu Linux. V: ZAJC, Baldomir (ur.). *Zbornik dvanajste mednarodne Elektrotehniške in računalniške konference ERK 2003, 25. - 26. september 2003, Ljubljana, Slovenija*, (Zbornik ... Elektrotehniške in računalniške konference ERK ..., 1581–4572). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2003, strani 397-400. [COBISS.SI-ID 8233494]

14. GREINER, Sašo, BREST, Janez, ŽUMER, Viljem. Grozdna arhitektura za porazdeljeno izvajanje programskega bremena. V: ZAJC, Baldomir (ur.). *Zbornik dvanajste mednarodne Elektrotehniške in računalniške konference ERK 2003, 25. - 26. september 2003, Ljubljana, Slovenija*, (Zbornik ... Elektrotehniške in računalniške konference ERK ..., 1581–4572). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2003, strani 417–420. [COBISS.SI-ID 8235030]

15. GREINER, Sašo, TUTEK, Simon, BREST, Janez, ŽUMER, Viljem. Quick adaptation of Web-based information systems with aspect-oriented features. V: LUŽAR - STIFFLER, Vesna (ur.), HLJUZ DOBRIĆ, Vesna (ur.). *ITI 2004 : proceedings of the 26th International Conference on Information Technology Interfaces, June 7-10, 2004, Cavtat, (Dubrovnik), Croatia*, (IEEE Catalog, No. 04EX794). Zagreb: University of Zagreb, SRCE University

Computing Centre, 2004, strani 145–150. [COBISS.SI-ID 8808214]

16. BOŠKOVIĆ, Borko, GREINER, Sašo, BREST, Janez, ŽUMER, Viljem. The representation of chess game. V: LUŽAR - STIFFLER, Vesna (ur.), HLJUZ DOBRIĆ, Vesna (ur.). 27th International Conference on Information Technology Interfaces, June 20-23, 2005, Cavtat, Croatia. *ITI 2005 : proceedings of the 27th International Conference on Information Technology Interfaces, June 20-23, 2005, Cavtat, Croatia*, (IEEE Catalog, No. 05EX1001). Zagreb: University of Zagreb, SRCE University Computing Centre, [2005], strani 381-386. [COBISS.SI-ID 9658134]

17. BREST, Janez, GREINER, Sašo, BOŠKOVIĆ, Borko, ŽUMER, Viljem. A heuristic algorithm for function optimization. V: BUDIN, Leo (ur.), RIBARIĆ, Slobodan (ur.). *MIPRO 2005 : 28. meunarodni skup, May/Svibanj 30 - June/Lipanj 03, 2005, Opatija, Croatia : Proceedings/Zbornik radova*. Rijeka: MIPRO, 2005, strani 91–94. [COBISS.SI-ID 9590550]

18. BOŠKOVIĆ, Borko, GREINER, Sašo, BREST, Janez, ŽUMER, Viljem. Učenje računalniškega šaha z uporabo algoritma diferencialne evolucije. V: ZAJC, Baldomir (ur.), TROST, Andrej (ur.). *Zbornik štirinajste mednarodne Elektrotehniške in računalniške konference ERK 2005, 26. - 28. september 2005, Portorož, Slovenija*, (Zbornik ... Elektrotehniške in računalniške konference ERK ...). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2005, zv. B, strani 71–74. [COBISS.SI-ID 9867798]

19. BREST, Janez, BOŠKOVIĆ, Borko, GREINER, Sašo, ŽUMER, Viljem. Nastavitev parametrov pri algoritmu diferencialne evolucije. V: ZAJC, Baldomir (ur.), TROST, Andrej (ur.). *Zbornik štirinajste mednarodne Elektrotehniške in računalniške konference ERK 2005, 26. - 28. september 2005, Portorož, Slovenija*, (Zbornik ... Elektrotehniške in računalniške konference ERK ...). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2005, zv. B, strani 79–82. [COBISS.SI-ID 9868054]

20. BOŠKOVIĆ, Borko, GREINER, Sašo, BREST, Janez, ŽUMER, Viljem. A differential evolution for the tuning of a chess evaluation function. V: 2006 IEEE World Congress on Computational Intelligence, Vancouver, Canada, July 16-21, 2006. *2006 IEEE World Congress on Computational Intelligence : Vancouver, BC, Canada, July 16-21, 2006 : a joint*

conference of the: 2006 International Conference on Neural Networks, 2006 IEEE International Conference on Fuzzy Systems, 2006 IEEE Congress on Evolutionary Computation. [Piscataway]: The Institute of Electrical and Electronics Engineering: = IEEE, cop. 2006, strani 6742–6747. [COBISS.SI-ID 10657046]

21. GREINER, Sašo, BREST, Janez, ŽUMER, Viljem. Advantages of dynamic method-oriented mechanism in a statically typed object-oriented programming language Z0. V: LUŽAR - STIFFLER, Vesna (ur.), HLJUZ DOBRIĆ, Vesna (ur.). 28th International Conference on Information Technology Interfaces, June 19-22, 2006, Cavtat/Dubrovnik, Croatia. *ITI 2006 : proceedings of the 28th International Conference on Information Technology Interfaces, June 19-22, 2006, Cavtat/Dubrovnik, Croatia*, (IEEE Catalog, No. 06EX1244). Zagreb: University of Zagreb, SRCE University Computing Centre, cop. 2006, strani 433–438. [COBISS.SI-ID 10546966]

22. BOŠKOVIĆ, Borko, GREINER, Sašo, BREST, Janez, ŽUMER, Viljem. Adaptivni algoritem diferencialne evolucije za uglaševanje parametrov ocenitve funkcije računalniškega šaha. V: ZAJC, Baldomir (ur.), TROST, Andrej (ur.). *Zbornik petnajste mednarodne Elektrotehniške in računalniške konference ERK 2006, 25. - 27. september 2006, Portorož, Slovenija*, (Zbornik ... Elektrotehniške in računalniške konference ERK ...). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2006, zv. B, strani 83–86. [COBISS.SI-ID 10768150]

23. BREST, Janez, SEPESY MAUČEC, Mirjam, BOŠKOVIĆ, Borko, GREINER, Sašo, ŽUMER, Viljem. Optimizacija z omejitvami: eksperimentalni rezultati s samo-prilagodljivim algoritmom diferencialne evolucije. V: ZAJC, Baldomir (ur.), TROST, Andrej (ur.). *Zbornik petnajste mednarodne Elektrotehniške in računalniške konference ERK 2006, 25. - 27. september 2006, Portorož, Slovenija*, (Zbornik ... Elektrotehniške in računalniške konference ERK ...). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2006, zv. B, strani 91–94. [COBISS.SI-ID 10768406]

2.05 Drugo učno gradivo

24. ŽUMER, Viljem, MERNIK, Marjan, BREST, Janez, KRUŠEC, Robert, ČREPINŠEK,

Matej, KOSAR, Tomaž, GREINER, Sašo, REBERNAK, Damijan, BOŠKOVIĆ, Borko, TUTEK, Simon. *Operacijski sistem LINUX/UNIX : gradivo za nadaljevalni tečaj*, (Računalniško opismenjevanje MŠZŠ). Maribor: Laboratorij za računalniške arhitekture in jezike in Center za programske jezike, 2003. 48 f. [COBISS.SI-ID 8160790]

25. ŽUMER, Viljem, MERNIK, Marjan, BREST, Janez, KRUŠEC, Robert, ČREPINŠEK, Matej, KOSAR, Tomaž, GREINER, Sašo, REBERNAK, Damijan, BOŠKOVIĆ, Borko. *Operacijski sistem LINUX/UNIX : gradivo za začetni tečaj*, (Računalniško opismenjevanje MŠZŠ). Maribor: Laboratorij za računalniške arhitekture in jezike in Center za programske jezike, 2003. 27 f. [COBISS.SI-ID 8160534]

2.09 Magistrsko delo

26. GREINER, Sašo. *Implementacija dinamičnih konceptov čistega statičnega objektno usmerjenega jezika : magistrsko delo*. Maribor: [S. Greiner], 2004. IX, 187 f. [COBISS.SI-ID 8993302]

2.11 Diplomsko delo

27.GREINER, Sašo. Arhitektura za objektno orientirane jezike : [diplomsko delo univerzitetnega študijskega programa], (Fakulteta za elektrotehniko, računalništvo in informatiko, Diplomska dela univerzitetnega študija). [Maribor]: [S. Greiner], 2002. IX, 88 f., ilustr. [COBISS.SI-ID 7595030]

Življenjepis

Sašo Greiner se je rodil v Mariboru dne 13.7.1978. Od 1985 do 1993 je obiskoval osnovno šolo Borcev za severno mejo v Mariboru. V letih 1993-1997 je obiskoval srednjo elektro računalniško šolo Maribor. Leta 1997 je pričel s študijem na fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Univerzitetni študij je leta 2002 zaključil z uspešnim zagovorom diplome "Arhitektura za objektno orientirane jezike". Istega leta se je zaposlil kot raziskovalec v laboratoriju za računalniške arhitekture in jezike. Leta 2003 je prevzel delovno mesto asistenta s področja računalništva. Sašo Greiner je trenutno aktiven na področju načrtovanja in implementacije objektno usmerjenih programskih jezikov, dizajna virtualnih računalniških arhitektur ter implementacije informacijskih sistemov.