Uporaba metrik pri identifikaciji smiselnih preoblikovanj programske kode

Marko Tekavc, Marjan Heričko
Univerza v Mariboru, Fakulteta za elektrotehniko računalništvo in informatiko, Inštitut za informatiko
{marko.tekavc, marjan.hericko}@uni-mb.si

Izvleček

Preoblikovanje programske kode je proces spreminjanja programske kode na način, da se obnašanje sistema navzven ne spremeni, spremeni in izboljša pa se notranja struktura programskega sistema. Natančno definirani, preizkušeni in delno z orodji podprti pristopi k preoblikovanju se osredotočajo predvsem na odpravljanje slabo oblikovane kode, ne pa tudi na to, kako identificirati takšno kodo v sistemu. Korak identifikacije slabo oblikovane kode in izbire ustreznega preoblikovanja je tako v veliki meri odvisen od izkušenj razvijalca. V prispevku je predstavljen in ovrednoten na metrikah temelječ pristop k identifikaciji slabo oblikovane objektno-orientirane programske kode in smiselnih preoblikovanj.

Abstract

Metric-Based Software Refactoring

Refactoring is a process that changes software code in order to improve its structure while keeping its behavior unchanged. Current approaches and tools related to refactoring are mainly focused on the code-restructuring step. The process of code identification is rarely addressed; therefore practitioners have to rely on their personal experiences and professional judgment. In this paper the method of identifying bad code in object-oriented systems is presented and evaluated. The proposed method is based on selected OO metrics. The method assists in the identification of code parts that require refactoring. It also provides recommendations for adequate change.

1 Uvod

Programski sistemi se zaradi prilagajanja spreminjajočemu se okolju, odpravljanja napak ali kakšnega drugega razloga med razvojem pa tudi v času uporabe velikokrat spreminjajo. Vsakršno spreminjanje sistema zahteva dobro razumevanje strukture sistema, velikokrat pa povzroči tudi vnos novih, neželenih napak. Zato je skozi celoten cikel razvoja programske opreme treba namenjati ustrezno pozornost skrbi za dobro strukturo programske kode. Prav preoblikovanja lahko pomagajo na poti do izboljšanja strukture in razumljivosti programske kode. Fowler v [1] natančno opisuje postopke in primere aplikacije preoblikovanj. Uporabnost in pridobitve procesa preoblikovanja programske kode v okoljih, ki uporabljajo ekstremno programiranje, pa predstavlja tudi [2].

Aplikacija preoblikovanj je dokaj enostavna, saj je treba v večini primerov le slediti natančno podanim postopkom. Nekatera preoblikovanja so definirana tako, da jih je bilo mogoče tudi avtomatizirati in so danes podprta v številnih orodjih. Še vedno pa obstaja težava, ki je še posebej izrazita pri velikih sistemih, in sicer kako poiskati odgovore na vprašanje, kdaj in kje je smiselno določena preoblikovanja aplicirati. Vprašanje je posebej zanimivo zaradi trditve, da nobena skupina metrik ne more nadomestiti človeške intuicije in subjektivnih zaznavanj pri ugotavljanju, katera preoblikovanja aplicirati in zakaj [1]. Skladno s tem se današnje raziskave na področju avtomatizacije preoblikovanj v glavnem ukvarjajo z apliciranjem preoblikovanj, ne pa s problemom, kako v kodi identificirati mesta, kjer bi uporabili preoblikovanja. Naloga, kako locirati dele kode, primerne za preoblikovanje, nikakor ni lahka, prav tako pa je zahtevna tudi naloga samodejne identifikacije preoblikovanja, s katerim bi tako identificirano kodo na najustreznejši način izboljšali. Prav to pa sta izziva, ki jih naslavljamo v pričujočem prispevku.

Cilj raziskave je ugotoviti, v kolikšni meri si lahko pomagamo z metrikami kot orodjem za identifikacijo tako slabo oblikovane kode kot tudi preoblikovanj, ki bi odpravila takšno kodo. V raziskavi, ki smo jo opravili, smo poskusili zajeti kar se da širok nabor najbolj znanih in uporabljanih metrik za objektno-orientirane sisteme ter ugotoviti, katere izmed njih in na

kakšen način so primerne za identifikacijo slabo oblikovane kode.

V razdelku poglavju so na kratko povzete bistvene značilnosti procesa preoblikovanja objektno orientiranih programskih sistemov. Sledi pregled sorodnih raziskav, kjer podrobneje predstavimo dva pristopa k samodejni identifikaciji slabo strukturirane kode in smiselnih preoblikovanj. V četrtem razdelku najprej predstavljamo izhodišča predlaganega pristopa, ki temelji na metrikah, razvrščenih v tri skupine. Prva skupina je namenjena identifikaciji možnih izboljšav strukture sistema, druga identifikaciji in preoblikovanju razredov, tretja skupina metrik, ki smo jo oblikovali, pa se osredotoča na metode znotraj posameznih razredov. Četrti razdelek predstavi rezultate praktične uporabe in ovrednotenja predlaganega pristopa na konkretnem primeru.

2 Izboljšanje strukture programske kode s preoblikovanjem

Po definiciji je preoblikovanje proces spreminjanja programske kode tako, da se obnašanje sistema navzven ne spremeni, spremeni in izboljša pa se notranja struktura sistema [1]. Preoblikovanje programske kode se je uveljavilo kot pomembna praksa pri razvoju programske opreme predvsem v povezavi z objektno orientiranimi sistemi in vpeljavo agilnih pristopov, saj brez njene uporabe razvijalci le s težavo udejanjajo temeljne vrednote in principe agilnosti, kot sta npr. odzivnost in preprostost. Načelo preprostosti namreč pomeni, da razvijemo minimalni sistem, ki še zadovoljuje zahteve uporabnikov. Če pred dodajanjem nove funkcionalnosti obstoječe kode ne bi izboljšali, torej preoblikovali in pripravili za dodajanje nove funkcionalnosti, bi sistem po nekaj iteracijah zaradi slabe strukturiranosti postal neobvladljiv in nezmožen nadaljnjega razvoja.

Ko govorimo o preoblikovanju in o identifikaciji delov kode, kjer je treba aplicirati določeno preoblikovanje, ne moremo mimo pojma slabo oblikovane kode. Slabo oblikovana koda je definirana kot del kode, ki sicer služi svojemu namenu, torej izvršuje neko zaporedje ukazov, ki skupaj pravilno izvedejo neko funkcionalnost, vendar pa bi lahko bil ta del kode napisan bolje, razumljiveje, morda v drugem razredu, komponenti ali metodi. Tabela 1 povzema po [1] najpomembnejše tipe slabo oblikovane kode, kratek opis in vrste preoblikovanja, ki jih je smiselno uporabiti za odpravljanje tega tipa slabo oblikovane kode. Vsakega izmed identificiranih tipov slabo oblikovane kode lahko torej odpravimo z uporabo enega ali več različnih vrst preoblikovanja. Zaradi boljše razumljivosti so imena posameznih vrst preoblikovanj v tabeli pa tudi v nadaljnjem besedilu navedena z izvornim angleškim izrazom.

Žal pa tudi podrobni seznam vseh identificiranih tipov slabo oblikovane kode ne more zagotoviti enoumnega kriterija, s pomočjo katerega bi ugotovili, kdaj je preoblikovanje nujno, temveč je treba primere slabo oblikovane kode vzeti kot smernice, kje in kako izboljšati kodo s pomočjo preoblikovanj. Seveda se odločitev glede tega, katero preoblikovanje uporabiti za odpravo določenega tipa slabo oblikovane kode, od primera do primera razlikuje. Včasih lahko slabo oblikovano kodo odpravimo že z enim samim preprostim preoblikovanjem, velikokrat pa je treba aplicirati več preoblikovanj zapored.

3 Sorodne raziskave

Čeprav se lahko strinjamo s trditvijo, da pri identifikaciji slabo oblikovane kode ne moremo nadomestiti človeških izkušenj in intuicije, obstaja nekaj raziskav, ki dokazujejo, da lahko ustrezni pristopi in metode vsaj pomagajo pri zadani nalogi. Predstavili bomo rezultate dveh raziskav s tega področja, podrobnejši opis je na voljo v [3] in [5].

3.1 Preoblikovanje na podlagi metrike razdalje

Raziskava *Metrics Based Refactoring* [3] je pokazala, da lahko s pomočjo posebnih metrik podpremo subjektivne ocene o slabo oblikovani kodi in identificiramo, kje aplicirati katero preoblikovanje. Na podlagi metrik izdelamo vizualno sliko sistema, iz katere ugotovimo, kje so mesta v programu, ki bi jih bilo treba preoblikovati. Raziskava se omeji zgolj na štiri tipe preoblikovanja in sicer *Move method, Move Field, Extract Class* in *Inline Class* ter temelji na dejstvu, da je veliko preoblikovanj osnovanih na principu "*Dajmo skupaj, kar sodi skupaj*". Tako avtorji definirajo, da je podobnost med dvema elementoma v zvezi z zbirko njunih skupnih lastnosti.

$$dist_B(x,y) := 1 - \frac{|p(x) \cap p(y)|}{|p(x) \cup p(y)|}$$

pri čemer sta x in y opazovana elementa,
p(x):= {lastnosti $p_i \in B$ | x ima v lasti p_i}

Formula 1: **Razdalja med opazovanima elementoma**

Na podlagi formule 1 vpeljemo pojem razdalje oz. oddaljenosti, s katero opišemo mero vezljivosti. Velja, da je vezljivost delov z nizkimi razdaljami boljša kot vezljivost delov z velikimi razdaljami. Razdalje računamo za atribute in metode razreda, pri čemer izhajamo iz predpostavke, da dva elementa sodita skupaj, če uporabljata drug drugega. Identifikacija možnega preoblikovanja je omejena na prej omenjena štiri preoblikovanja in poteka na osnovi identifikacije velikih razdalj med člani razreda ter majhnih razdalj do članov drugega razreda.

Tabela 1: **Tipi slabo oblikovane kode**

Slabo oblikovana koda	Opis	Smiselna preoblikovanja
Komentarji Assertion	V komentarjih opisujemo, kaj dela programska koda.	Extract Method, Rename Method, Introduce
Predolga metoda	Metoda je tako obsežna, da težko ugotovimo, kakšna je njena funkcionalnost.	Extract Method, Replace Temp with Query, Introduce Parameter Object, Preserve Whole Object, Replace Method with Method Object
Predolg nabor parametrov	V metodo pošiljamo vse podatke prek parametrov, čeprav bi lahko metoda sama prišla do potrebnih podatkov.	Replace Parameter with Method, Preserve Whole Object, Replace Method with Method Object
Podvojena koda	Imamo enako kodo na več različnih mestih v sistemu, kar otežuje spreminjanje programa.	Extract Method, Pull Up Field, Form Template Method, Substitute Algorithm
Prevelik razred	Razred, ki opravlja preveč funkcionalnosti.	Extract Class, Extract Subclass
Tip vgrajen v ime	V imenih metod imamo podatke, ki so vidni že iz samega podpisa metode.	Rename Method
Nekonsistentna imena	Za isti pojem uporabljamo različna imena.	Rename Method
Špekulativna splošnost	Pretiravamo z generalizacijo kode, z namenom predvidevanja prihodnjih potreb.	Collapse Hierarchy, Inline Class, Remove Parameter, Rename Method
Obsedenost s primitivnimi tipi	Za sestavljene podatke uporabljamo primitivne tipe, namesto da bi uporabili preproste objekte.	Replace Data Value with Object, Replace Type Code with Class/Subclass State/Strategy, Extract Class, Introduce Parameter Object, Replace Array With Object
Podatkovni razred	Razred je brez funkcionalnosti – sestavljen samo z atributi in metodami get in set.	Move Method
Skupine podatkov	Skupine podatkov, ki jih vedno najdemo skupaj.	Extract class, Introduce Parameter Object, Preserve Whole Object
Zavrnjena zapuščina	Podrazredi ne potrebujejo vsega kar podedujejo.	Replace Inheritance With Delegation.
Neprimerna intimnost	Dva razreda sta preveč prepletena.	Change Bidirectional Association to Unidirectional, Extract Class, Hide Delegate
Len razred	Razred, ki ne opravlja dovolj funkcionalnosti.	Collapse Hierarchy, Inline Class
Zavidanje lastnosti	Metodo bolj zanimajo lastnosti nekega drugega razreda kot tistega, v katerem se dejansko nahaja.	Extract method, Move Method
Verige sporočil	Odjemalec mora uporabiti en razred, da pride do drugega in potem tega, da pride do tretjega itd.	Hide Delegate
Osrednja osebnost	Razred večino stvari delegira k drugemu razredu.	Inline Method, Replace Delegation With Inheritance
Raznolike spremembe	Razred se pogosto iz različnih razlogov spreminja na različne načine.	Extract Class
Poseg s šibrovko	Nasprotno od raznolikih sprememb. Sprememba povzroči veliko majhnih sprememb v različnih razredih.	Inline Class
Vzporedne hierarhije dedovanj	Posebna oblika posega s šibrovko. Vedno ko naredimo podrazred razreda, moramo narediti tudi podrazred nekega drugega razreda.	Move Method, Move Field

Po aplikaciji koncepta merjenja razdalje na osnovi vezljivosti in po izračunu vseh razdalj med metodami in atributi izdelamo metriko razdalj med opazovanimi elementi, s pomočjo katere izvedemo vizualizacijo. Vizualizacija je mogoča s pomočjo programa "3D-Spring Embedder"[4], s katerim izdelamo 3D modele v jeziku VRML. Ključnega pomena za orodje, ki omogoča vizualizacijo, je hitrost in povezanost z razvojnim okoljem. Za uporabo in izračun vrednosti metrike z razdaljami, ki je potrebna za vizualizacijo, je treba izdelati repozitorij relevantnih konceptov, kot so razredi, metode, povezave ipd. Nato izberemo konstrukte, ki jih bomo analizirali, in izračunamo razdalje. Na osnovi izračunanih pozicij in informacij iz repozitorija izdelamo model VRML. Ena izmed slabosti opisane rešitve podpornega orodja je prav trajanje izgradnje repozitorija uporabljenih konceptov iz projekta, ki se sicer izvaja samo enkrat, vendar pa se mora ob spremembah v programski kodi posodobiti. Dodatno težavo predstavlja interpretacija vizualizacije in razpoznavanje slabo oblikovane kode. S kompleksnostjo sistema namreč narašča tudi kompleksnost vizualizacije, kar otežuje identifikacijo delov s slabo oblikovano kodo. Poleg tega obstajata pri uporabi te metode za velike sisteme še dve težavi:

- Veliko razredov ne moremo obravnavati ločeno, saj so vpeti v strukturo dedovanja. Če raziskujemo določen razred brez upoštevanja konteksta dedovanja, lahko pridemo do napačnih sklepov oziroma predlogov za preoblikovanje.
- Ker lahko v velikem sistemu obstaja več tisoč razredov in je lahko podrobna analiza časovno zelo potratna, se pojavlja problem, kako v tako velikem sistemu v predstavljeni vizualizaciji opaziti tiste razrede, ki jih je treba preoblikovati.

Čeprav so nekateri izmed omenjenih problemov delno rešeni, je metoda za velike sisteme le pogojno uporabna, poleg tega pa uporabljena metrika ni podprta v obstoječih razvojnih okoljih in orodjih.

3.2 Preoblikovanje na podlagi zgodovine sprememb

V članku *Improving Evolvability through Refactoring* [5] je opisana metoda, ki uporablja zgodovinske podatke, pridobljene iz repozitorijev, kot je CVS. Metoda se osredotoča na sklope sprememb – če se nekateri deli skozi več izdaj zelo pogosto spreminjajo, potem lahko takšni podatki služijo kot pokazatelji potencialnih mest za aplikacijo preoblikovanj. Poleg koncepta slabo oblikovane kode vpeljemo še koncept sumljivih sprememb. Takšne spremembe težko opazimo v kodi, lahko pa jih identificiramo na podlagi pregleda zgodovine sprememb. Pristop omogoča zaznavanje sumljivih sprememb in s tem spodbuja razvijalce k preoblikovanju takšnih delov izvorne kode.

Definicija sklopljenosti sprememb pravi, da sta elementa logično sklopljena, če spremembe skozi dovolj veliko število izdaj vplivajo na oba elementa. Predstavljena sta dva tipa sumljivih sprememb:

1. *Osrednja osebnost* (»Man in the middle«) – osrednji razred se razvija skupaj s številnimi ostalimi razredi, ki so razpršeni po različnih modulih v sistemu. Takšen razred zavira razvoj samostojnih modulov zaradi močne povezanosti z drugimi deli sistema.
2. *Vsebnik podatkov* (»Data container«) – obstajata dva razreda, kjer prvi vsebuje vse potrebne podatke, drugi pa sodeluje z ostalimi razredi in zagotavlja funkcionalnost, ki uporablja predvsem podatke prvega razreda. S tem krši princip ograjevanja podatkov in povezanih funkcionalnosti.

Predlagana metoda je podprta tudi z orodjem, izdelanim v ta namen – EvoLens [6]. Orodje razčleni dnevniške datoteke iz repozitorija CVS in izračuna spremembo sklopljenosti med razredi. Sklopljenost se nato skupaj s strukturnimi informacijami vizualizira. Pristop je bil preizkušen na zgodovini sprememb velikega industrijskega projekta skozi obdobje petnajstih mesecev. V tem času so bila, s pomočjo analize zgodovinskih podatkov predlagana mesta za preoblikovanje. Po aplikaciji preoblikovanj in nadaljnjih petnajstih mesecih opazovanj so ocenili učinkovitost pristopa, ki je podprla hipotezo, da je kombinacija analize odvisnosti sprememb in preoblikovanja uporabna in učinkovita [5].

Žal je predstavljeni pristop uporaben zgolj v povezavi s spremljanjem in zbiranjem zgodovinskih podatkov, v praksi pa bi želeli programsko kodo izboljšati tudi v primeru, ko tovrstnih podatkov še ni, zato smo definirali pristop, ki temelji zgolj na aktualnem sistemu ter temelji na uveljavljenih metrikah objektno orientirane kode, hkrati pa lahko metrične vrednosti enostavno pridobimo in zberemo v času razvoja s pripomočki in metričnimi orodji, ki so po navadi že vključena v sodobna razvojna okolja.

4 Uporaba uveljavljenih objektno orientiranih metrik pri preoblikovanju

Pristop, ki ga predlagamo, temelji na predpostavki, da bi bilo mogoče s pomočjo uveljavljenih objektno

orientiranih metrik identificirati slabo oblikovane dele kode in predlagati preoblikovanja, s katerimi bi takšno kodo izboljšali. Na osnovi analize najbolj uveljavljenih in razširjenih metrik za objektne sisteme smo oblikovali nabor primernih metrik in jih razdelili v tri skupine glede na to, kakšen tip slabo oblikovane kode lahko z njimi identificiramo:

1. *Slaba struktura* – V to skupino smo uvrstili predvsem metrike MOOD, ki jih je predlagal Abreau [7, 8]. Z njimi lahko ugotovimo, ali program upošteva pravila in navodila dobrega objektnega načrtovanja ter implementacije.
2. *Slabo oblikovani razredi* – V to skupino smo uvrstili metrike, ki izpostavijo slabo oblikovane razrede. Tako identificiramo razrede, ki so pretesno povezani z drugimi razredi, razrede, ki imajo nizko vezljivost, in razrede, ki so preveliki ali imajo preveliko število operacij in atributov.
3. *Prezapletene metode* – V tretji skupini najdemo metrike, ki se ukvarjajo z metodami. Prezapletene metode lahko identificiramo po dolžini, preveliki ciklomatični kompleksnosti, prezapletenih vejitvah ali prevelikem številu operacij, lokalnih spremenljivk ali parametrov.

Tabela 2: **Povezanost metrik, slabo oblikovane kode in preoblikovanj**

Metrika	Slabost	Možni tip slabo oblikovane kode	Preoblikovanje
Slaba struktura			
Faktor sklopljenosti (CF)	Prevelika sklopljenost	Neprimerna intimnost, osrednja osebnost, zavidanje lastnosti	Move Method, Move Field, Change Bidirectional Association to Unidirectional, Replace Delegation With Inheritance
Faktor polimorfičnosti (PF)	Neuporaba polimorfizma	Podvojena koda	Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Conditional with Polymorphism
Skrivanje metod/atributov (AHF/MHF)	Kršitev ograjevanja		Encapsulate Field, Self Encapsulate Field, Hide Method
Dedovanje atributov/metod (AIF/MIF)	Neuporaba dedovanja	Podvojena koda, osrednja osebnost	Replace Delegation With Inheritance, Extract Subclass, Replace Type Code with Subclass/State/Strategy
Slabo oblikovani razredi			
Sklopljenost med objekti (CBO)	Prevelika sklopljenost	Neprimerna intimnost, verige sporočil	Change Bidirectional Association to Unidirectional, Extract Class, Hide Delegate
Sklopljenost klicev metod (MIC)	Prevelika sklopljenost	Neprimerna intimnost, verige sporočil	Extract Class, Hide Delegate, Move Method
Pomanjkanja vezljivosti med metodami (LCOM)	Pomanjkanje vezljivosti		Replace inheritance with delegation, Move Field, Move Method
Število atributov/operacij/članov razreda (NOA/NOO/NOM)	Prevelik razred	Prevelik razred	Extract Class, Extract Subclass
Število dodanih metod (NOAM)	Neustrezna uporaba dedovanja		Extract subclass, Replace inheritance with delegation
Uteženе metode razreda (WMPC)	Prekompleksen razred	Prevelik razred, Podvojena koda	Extract Class, Extract Subclass, Replace Type Code with State/Strategy, Replace Conditional with Polymorphism
Prezapletene metode			
Ciklomatična kompleksnost (CC)	Prekompleksna metoda	Predolga metoda, Podvojena koda	Extract method, Replace Type code with Class/Subclasses/State/Strategy
Število vrstic kode (LOC)	Predolga metoda	Predolga metoda	Extract Method
Maksimalno število vejitev v metodi (MNOB)	Prekompleksna metoda		Extract method, Replace Type code with Class/Subclasses/State/Strategy
Število parametrov (NOP)	Predolg nabor parametrov	Predolg nabor parametrov	Replace Parameter with Method, Preserve Whole Object, Introduce Parameter Object
Število lokalnih spremenljivk (NOLV)	Lokalne spremenljivke		Replace Temp with Query, Inline Temp

Tabela 2 za vsako izmed izbranih metrik prikazuje, katero slabost in tip slabo oblikovane kode lahko odkrijemo z njo ter katera so iz tega izhajajoča najpogosteje uporabljana preoblikovanja za izboljšanje takšne kode. Dejansko lahko z nekaterimi metrikami identificiramo nasprotne tipe slabo oblikovane kode ali pa metrika identificira tudi kakšen drugi tip slabo oblikovane kode, ki v tabeli ni naveden. Tudi zato tabela ne nudi popolnega pregleda, temveč le podaja najpogostejše povezave med metrikami, slabo oblikovano kodo in preoblikovanji.

4.1 Slaba struktura

Metrike iz te skupine povedo, kako dobro je sistem strukturiran in kako so uporabljeni principi objektnega razvoja (dedovanje, ograjevanje, polimorfizem). Večina teh metrik vrne vrednost za celoten sistem in zato niso primerne za lociranje mest s slabo oblikovano kodo, pač pa kažejo, v kateri smeri bodo potekala preoblikovanja.

Metrika faktor sklopljenosti (Coupling Factor – CF) prikaže odstotek sklopljenosti med razredi v sistemu. Čim večja je ta vrednost, tem bolj sklopljeni so razredi in je sistem težji za vzdrževanje. Visoka vrednost je pogosto znak za smiselnost premestitve metod in atributov (*Move Method, Move Field*) v ustreznejše razrede ali spremembo dvosmerne povezave v enosmerno (*Change Bidirectional Association to Unidirectional*). Seveda sta ti dve preoblikovanji zgolj najočitnejši možnosti, sklopljenost lahko namreč zmanjšamo tudi z drugimi preoblikovanji.

Metrika faktor polimorfičnosti (Polymorphism Factor – PF) vrne stopnjo polimorfizma v opazovanem sistemu. Nizka vrednost praviloma pomeni, da je v sistemu premalo uporabljen eden izmed osnovnih principov objektnega razvoja – polimorfizem. Običajno v sistemu namesto polimorfizma nastopajo stavki switch/case, ki jih preoblikujemo s preoblikovanji *Replace Type Code with Subclasses, Replace Type Code with State/Strategy* in *Replace Conditional with Polymorphism.*

Metriki skrivanje metod oziroma atributov (Attribute Hiding Factor – AHF / Method Hiding Factor – MHF) izpostavita kršitve ograjevanja. Problem razrešimo s spreminjanjem vidnosti in ograjevanjem atributov/metod, za kar lahko uporabimo preoblikovanja *Encapsulate Field, Self Encapsulate Field* in *Hide Method.*

Metriki dedovanje atributov oziroma metod (Attribute Inheritance Factor – AIF / Method Inheritance Factor – MIF) vrneta delež vseh podedovanih atributov/metod v sistemu. Če je vrednost nizka, to navadno pomeni, da v sistemu ni vzpostavljena ustrezna hierarhija razredov, saj je premalo uporabljan eden temeljnih principov objektnega razvoja – dedovanje.

4.2 Slabo oblikovani razredi

V tej skupini so metrike, s pomočjo katerih lahko ugotovimo, ali imamo v našem sistemu ustrezne razrede in povezave med njimi. Z metrikami iz te skupine praviloma identificiramo točno določen razred ali razrede, ki jih je treba preoblikovati, hkrati pa ugotovimo, za kakšen tip slabo oblikovane kode gre in s pomočjo katerih preoblikovanj lahko odpravimo takšno kodo.

Metrika sklopljenost med objekti (Coupling Between Objects – CBO) vrne število razredov, s katerimi je povezan opazovan razred. Prevelika sklopljenost med razredi je značilna za modularno strukturo in preprečuje ponovno uporabo ter otežuje vzdrževanje sistema. Večja sklopljenost zahteva tudi rigoroznejše testiranje. Metrika nakazuje neprimerno intimnost (*Inappropriate Intimacy*) med razredi. Sklopljenost med razredi lahko zmanjšamo s preoblikovanji *Change Bidirectional Association to Unidirectional, Extract Class* in *Hide Delegate.*

Metrika sklopljenost klicev metod (Method Invocation Coupling – MIC) vrne relativno število razredov, h katerim opazovani razred pošilja sporočila:

$$\text{MICnom} = \text{nMIC}/(\text{N-1})$$

pri čemer je nMIC število razredov, h katerim se pošiljajo sporočila, in N število vseh razredov v sistemu.

Prevelika sklopljenost med razredi ima vpliv na:

1. vzdrževanje – vzdrževanje močno sklopljenih razredov je težavnejše zaradi odvisnosti od povezanih razredov;
2. razumljivost – težja razumljivost razreda, saj je za razumevanje po navadi treba razumeti tudi povezane razrede ali njihove dele;
3. nagnjenost k napakam, težavno testiranje – napake v razredu so premo sorazmerne s številom povezav na druge razrede in posledično ima to negativen vpliv na testiranje.

Visoka vrednost te metrike torej pomeni močno povezan razred in nakazuje neprimerno intimnost (*Inappropriate Intimacy*) med razredi. Spet si lahko

pomagamo s preoblikovanji, kot so *Extract Class, Hide Delegate* in *Move Method.*

Metrika pomanjkanje vezljivosti med metodami (Lack of Cohesion Of Methods – LCOM) vrne odstotek metod, ki ne dostopajo do določenega atributa, glede na vse atribute v razredu. Visoka vrednost te metrike nakazuje pomanjkanje vezljivosti in pomeni, da je razred slabo zasnovan. Pomanjkanje vezljivosti povečuje kompleksnost.

Pri identifikaciji slabo oblikovanih razredov si lahko pomagamo tudi s tremi sorodnimi metrikami, ki preštejejo atribute, metode oziroma člane razreda. Na podlagi *metrik števila atributov, operacij oziroma članov razreda (Number of Attributes – NOA, Number of Operations – NOO, Number of Members – NOM)* lahko identificiramo (pre)kompleksen, prevelik razred, ki je eden najpogostejših tipov slabo oblikovane kode [1]. Rešitev je v razdelitvi razreda na ustreznejše razrede/podrazrede s preoblikovanjem *Extract Class/Extract Subclass.*

Metrika število dodanih metod (Number of Added Methods – NOAM) prešteje število (dodatnih) operacij (pod)razreda; podedovane in predefinirane (overidden) metode ne upošteva. Metrika ne zadeva razredov brez staršev. Prevelika vrednost te metrike pomeni, da se funkcionalnost izbranega podrazreda preveč razlikuje od starša. S pomočjo preoblikovanja *Extract subclass* lahko ustvarimo dodatni podrazred ali hierarhijo nadomestimo z delegiranjem (*Replace inheritance with delegation).*

Metrika utežene metode razreda (Weighted Methods Per Class – WMPC) je zadnja metrika, ki smo jo uvrstili v skupino metrik, primernih za identifikacijo slabo oblikovanih razredov. Prva oblika te metrike meri kompleksnost razreda na podlagi ciklomatične kompleksnosti metod v razredu, druga pa temelji na predpostavki, da več metod in večje število parametrov metod navadno pomeni tudi večjo kompleksnost.

4.3 Prezapletene metode

Uporaba metrik iz te skupine izpostavi slabo oblikovane metode. Praviloma lahko s pomočjo teh metrik identificiramo tudi tip slabo oblikovane kode v metodi in s tem preoblikovanja, ki odpravijo takšno kodo.

Metrika ciklomatične kompleksnosti (Cyclomatic Complexity – CC) pomaga pri identifikaciji kompleksnih metod, saj predstavi število ciklov v merjeni metodi. Dejansko prešteje število možnih poti skozi algoritem s pomočjo štetja stavkov *if, for* in *while.* Po definiciji dobimo vrednost metrike z enačbo:

$$CC = L - N + 2P$$

pri čemer je L število povezav v grafu kontrolnega toka, N je število vozlišč in P število nepovezanih delov. Čim višja je izmerjena vrednost metrike, tem kompleksnejša je merjena metoda.

Metrika števila vrstic kode (Lines of Code – LOC) šteje vrstice v metodi/razredu/paketu/aplikaciji, vključno s komentarji in praznimi vrsticami. Preveliko število vrstic kode običajno pomeni (pre)kompleksno, predolgo metodo. Tip slabo oblikovane kode, ki ga identificiramo, je torej *predolga metoda,* odpravimo pa ga s preoblikovanjem *Extract Method.*

Metrika maksimalno število vejitev v metodi (Maximum Number of Branches – MNOB) je definirana kot maksimalno možno število *if-else* in/ali *case* vejitev v metodi. Visoka vrednost praviloma pomeni kompleksno metodo. Preoblikovanja, s katerimi lahko odpravimo kompleksne pogojne strukture, *so Extract method, Replace Type code with Class/Subclasses/State/Strategy.*

Metrika število parametrov (Number of Parameters – NOP) vrne število parametrov v opazovani metodi. Preveliko število parametrov je dobro poznana slabo oblikovana koda, imenovana predolg nabor parametrov (*Long Parameter List*), ki jo lahko odpravimo z enim preoblikovanjem ali kombinacijo preoblikovanj: *Replace Parameter with Method, Preserve Whole Object* in *Introduce Parameter Object.*

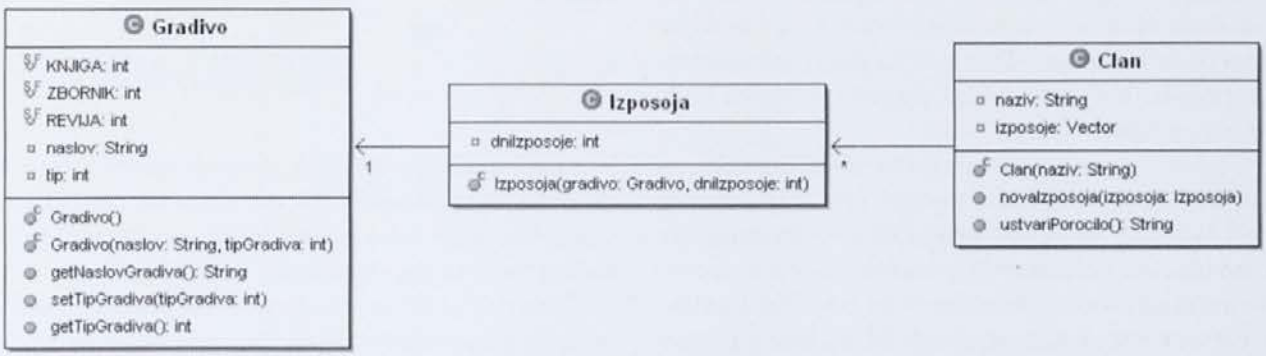
Metrika število lokalnih spremenljivk (Number of Local Variables – NOLV) je pomembna zato, ker lokalne spremenljivke otežujejo ali onemogočajo določena preoblikovanja metod. Preoblikovanji, ki ju lahko uporabimo za odpravo lokalnih spremenljivk, sta *Replace Temp with Query* in *Inline Temp.*

5 Ovrednotenje pristopa na primeru

Vrednotenje ustreznosti predstavljenih skupin metrik, ki lahko služijo kot smernice pri izvajanju preoblikovanj, smo izvedli na enostavnem sistemu za vodenje izposoje gradiva v knjižnici. Programska koda je napisana praktično brez upoštevanja konceptov objektne orientacije, z eno relativno dolgo in kompleksno metodo, in je kot takšna zelo primerena za preoblikovanje. Originalna programska koda je na voljo na naslovu http://cot.uni-mb.si/Refactoring/koda.zip. Slika 1 prikazuje razredni diagram tega programa.

Analiza programske kode je bila izvedena z orodjem, ki nudi podporo izvajanju metrik na programski kodi in avtomatizacijo nekaterih enostavnejših pre-

oblikovanj. Tabela 3 prikazuje najzanimivejše rezultate metrik, s pomočjo katerih lahko ocenimo strukturo našega programa. Faktorja dedovanja metod (MIF) in polimorfizma (PF) sta enaka 0, kar pomeni, da ta dva koncepta v našem primeru nista navzoča. Prav tako je slab tudi faktor sklopljenosti (CF), saj nakazuje precej veliko sklopljenost med našimi razredi. Sprejemljiva pa je stopnja prikritosti atributov (AHF), to pomeni, da je večinoma upoštevan koncept ograjevanja.



Slika 1: **Razredni diagram kode (pred preoblikovanjem)**

Tabela 3: **Rezultati metrik strukture kode pred preoblikovanjem**

Struktura				
	AHF	CF	MIF	PF
Knjiznica	**67**	**67**	**0**	**0**

Z analizo tega sklopa metrik smo postavili tezo, da je programska koda precej slabo strukturirana in skorajda ne upošteva dobrih praks uporabe konceptov objektne orientacije. Predvideli smo, da lahko z uporabo ustreznih preoblikovanj strukturo našega programa izboljšamo.

Pri analizi rezultatov metrik nad razredi lahko vidimo (tabela 4), da dobimo najslabše rezultate v razredu *Clan*. Velika sklopljenost razreda *Clan* (metrika CBO) nakazuje modularno strukturo razreda. Visoke vrednosti metrik za pomanjkanje vezljivosti (LCOM2 in LCOM3) kažeta na slabo zasnovo prav vseh razredov. V razredu *Clan* pa izstopata še rezultata metrike MIC, ki kaže, da razred *Clan* relativno pogosto pošilja sporočila drugim razredom, in metrika WMPC1, ki v primerjavi z drugimi razredi kaže na veliko ciklomatično kompleksnost metod razreda *Clan*.

Z analizo metrični rezultatov tega sklopa metrik lahko ugotovimo, da bo treba pri preoblikovanju največ pozornosti posvetiti razredu *Clan*. Izvesti želimo takšna preoblikovanja, da zmanjšamo močno povezanost razreda z drugimi razredi ter poenostavimo kompleksne metode v razredu.

Tabela 4: **Metrične vrednosti za razrede pred preoblikovanjem**

Razredi									
	CBO	**LCOM2**	**LCOM3**	**MIC**	**NOA**	**NOM**	**NOO**	**WMPC1**	**WMPC2**
Knjiznica	1	56	83	33	2	5	3	7	3
Knjiznica.Clan	3	50	75	100	2	5	3	12	4
Knjiznica.Gradivo	0	84	75	0	2	5	3	5	4
Knjiznica.Izposoja	1	33	100	0	2	4	2	3	2

Tabela 5: **Rezultati metrik nad metodami pred preoblikovanjem**

Metode					
	CC	**LOC**	**MNOB**	**NOLV**	**NOP**
Knjiznica	2 [9]	44 [66]			
Knjiznica.Clan	3	86			
Knjiznica.Clan.Clan	1	1	0	0	1
Knjiznica.Clan.getNaziv	1	1	0	0	0
Knjiznica.Clan.novaIzposoja	1	1	0	0	1
Knjiznica.Clan.ustvariPorocilo	9	66	13	7	0
Knjiznica.Gradivo	1	28			
Knjiznica.Gradivo.Gradivo	1	0	0	0	0
Knjiznica.Gradivo.Gradivo	1	2	0	0	2
Knjiznica.Gradivo.getNaslovGradiva	1	1	0	0	0
Knjiznica.Gradivo.getTipGradiva	1	1	0	0	0
Knjiznica.Gradivo.setTipGradiva	1	1	0	0	1
Knjiznica.Izposoja	1	19			
Knjiznica.Izposoja.Izposoja	1	2	0	0	2
Knjiznica.Izposoja.getDnilzposoje	1	1	0	0	0
Knjiznica.Izposoja.getGradivo	1	1	0	0	0

Tabela 5 prikazuje rezultate metrik nad metodami. V oglatem oklepaju so pri metrikah CC in LOC navedene tudi maksimalne izmerjene vrednosti. Že na prvi pogled je jasno, da je metoda *ustvariPorocilo* tista, ki je "krivec" za kompleksnost tega razreda. Predvsem izstopa ciklomatična kompleksnost metode in maksimalno možno število vejitev. 66 vrstic kode samo po sebi sicer ni veliko, vendar gre za zelo enostaven primer in vidimo lahko, da je to polovica kode našega primera.

S pomočjo analize vrednosti metrik tega sklopa metrik lahko identificiramo metodo, pri kateri se lotimo preoblikovanja. Rezultati kažejo, da je treba zmanjšati kompleksnost in dolžino identificirane metode, zato jo z ustreznimi preoblikovanji razbijemo na manjše dele, pri čemer bomo skušali odpraviti tudi lokalne spremenljivke.

Glede na vrednosti metrik, ki nakazujejo preveliko sklopljenost razreda, nato tako razbito metodo skušamo premakniti na ustreznejša mesta oz. drug razred in s tem zmanjšati sklopljenost in povečati vezljivost.

Z nizom različnih preoblikovanj, ki izhajajo oz analize rezultatov metrične analize in predlaganih preoblikovanj, pridemo do strukture razredov, ki jo prikazuje slika 5. Programska koda je na voljo na: http://cot.uni-mb.si/Refactoring/PKoda.zip.

Po končanem preoblikovanju programsko kodo ponovno ovrednotimo s pomočjo istih metrik. Metrične vrednosti prvega sklopa metrik, s katerim ocenjujemo strukturo programske kode, prikazuje tabela 6.

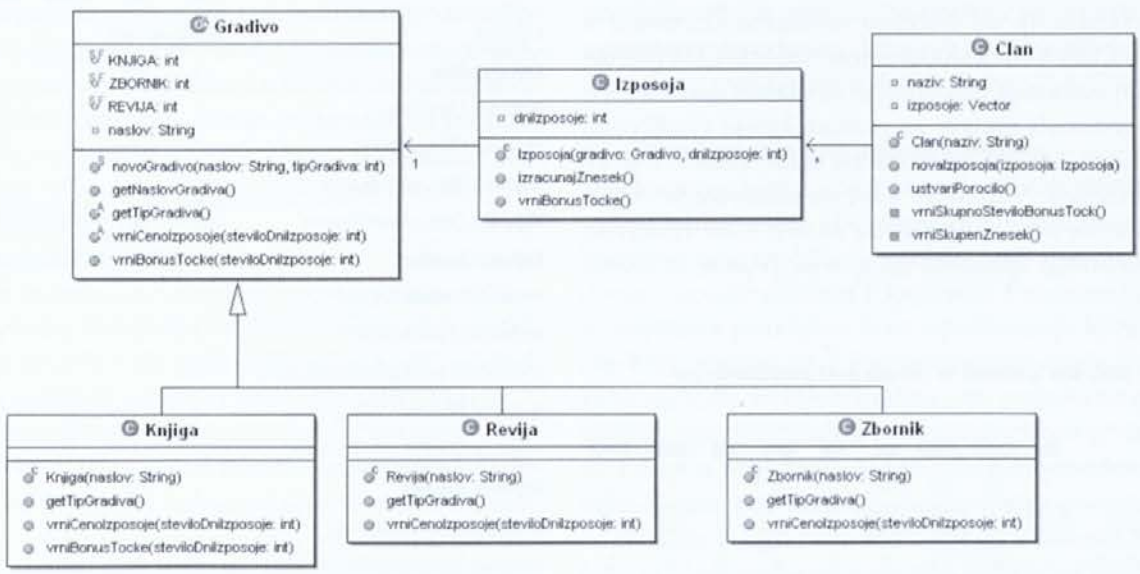
Tabela 6: **Rezultati metrik strukture kode po preoblikovanju**

Struktura				
	AHF	**CF**	**MIF**	**PF**
Knjiznica	63	23	45	39

Iz rezultatov je razvidno, da smo s pomočjo preoblikovanj zmanjšali odstotek sklopljenosti med razredi v sistemu. Prav tako pa vrednosti metrik MIF in PF kažeta na uporabo dedovanja in polimorfizma.

Tabela 7 kaže na izboljšanje vrednosti metrik v razredu *Clan*. Vidimo lahko, da smo oblikovali tri dodatne razrede glede na strukturo, ki smo jo imeli pred pričetkom preoblikovanj. Ker smo funkcionalnost večinoma prenesli v razred *Gradivo* in njegove podrazrede, so pričakovano rezultati metrik za ta razred slabši kot pred preoblikovanjem, vendar pa lahko – gledano rezultate metrik v celoti – ugotovimo, da smo izboljšali naše razrede.

S preoblikovanji smo izboljšali prej zelo kompleksno metodo *ustvariPorocilo*, saj smo funkcionalnosti iz te metode prenesli v druge metode in v ustreznejše razrede. Izboljšali smo tudi povprečno število vrstic metod v projektu. V celoti rezultate metrik nad metodami po preoblikovanju prikazuje tabela 8.



Slika 5: **Razredni diagram aplikacije po preoblikovanju**

Tabela 7: **Rezultati metrik razredov po preoblikovanju**

Razredi									
	CBO	**LCOM2**	**LCOM3**	**MIC**	**NOA**	**NOM**	**NOO**	**WMPC1**	**WMPC2**
Knjiznica	1	56	72	23	1	5	4	6	5
Knjiznica.Clan	3	50	62	40	2	7	5	9	6
Knjiznica.Gradivo	3	79	87	60	1	7	6	8	11
Knjiznica.Izposoja	1	40	66	20	2	6	4	5	4
Knjiznica.Knjiga	0			20	0	3	3	6	5
Knjiznica.Revija	0			0	0	2	2	5	3
Knjiznica.Zbornik	0			0	0	2	2	4	3

6 Sklep

V članku smo naslovili področje preoblikovanja objektne programske kode s poudarkom na raziskavi uporabnosti uveljavljenih OO metrik pri identifikaciji slabo oblikovane kode. Ugotovili smo, da obstaja le malo raziskav na temo uporabe metrik za identifikacijo slabo oblikovane kode. Povzeli smo značilnosti dveh pristopov, ki se tej tematiki najbolj približata, ju na kratko opisali ter predstavili njune cilje, prednosti in slabosti.

Predstavili in ovrednotili smo tudi pristop, ki temelji na uporabi treh kategorij metrik glede na to, kakšen tip slabo oblikovane kode lahko identificiramo z njimi. V kategorijo metrik, ki pomagajo pri identifikaciji slabe strukture programske kode, smo uvrstili izbrane metrike iz skupine metrik MOOD, v drugo kategorijo metrike, s katerimi identificiramo slabo oblikovane razrede, v tretjo pa tiste, ki izpostavijo slabo oblikovane metode. Za predstavljene metrike smo podali, kakšno vrsto slabo oblikovane kode lahko odkrijemo z njimi, in kadar je bilo mogoče, predlagali preoblikovanja, s katerimi bi takšno kodo izboljšali.

Predlagani pristop smo ovrednotili na primeru in tako spoznali prednosti ter slabosti. Metrike so se izkazale kot uporaben pripomoček pri preoblikovanju. Pokazali smo, da lahko z njimi ovrednotimo strukturo programa in lociramo nekatere slabo oblikovane dele kode. Nobenega dvoma ni, da že uporaba preprostih metrik daje koristne smernice za izboljšanje oz. preoblikovanje strukture programske kode. Žal pa se je izkazalo, da je potrebno nekaj izkušenj, da se odločimo, kako preoblikovati in izbrati najprimernejša preoblikovanja. V nadaljnjih raziskavah bi se bilo treba osredotočiti predvsem na možnost uporabe kombi-

Tabela 8: **Rezultati metrik nad metodami po preoblikovanju**

Metode					
	CC	**LOC**	**MNOB**	**NOLV**	**NOP**
Knjiznica	2 [2]	34 [27]			
Knjiznica.Clan	2	66			
Knjiznica.Clan.Clan	1	1	0	0	1
Knjiznica.Clan.getNaziv	1	1	0	0	0
Knjiznica.Clan.novaIzposoja	1	1	0	0	1
Knjiznica.Clan.ustvariPorocilo	2	27	0	5	0
Knjiznica.Clan.vrniSkupenZnesek	2	6	0	3	0
Knjiznica.Clan.vrniSkupnoSteviloBonusTock	2	6	0	3	0
Knjiznica.Gradivo	1	40			
Knjiznica.Gradivo.getNaslovGradiva	1	1	0	0	0
Knjiznica.Gradivo.getTipGradiva	1	0	0	0	0
Knjiznica.Gradivo.novoGradivo	1	6	4	0	2
Knjiznica.Gradivo.setNaslov	1	1	0	0	1
Knjiznica.Gradivo.vrniBonusTocke	3	8	3	1	1
Knjiznica.Gradivo.vrniCenoIzposoje	1	0	0	0	1
Knjiznica.Izposoja	1	27			
Knjiznica.Izposoja.Izposoja	1	2	0	0	2
Knjiznica.Izposoja.getDniIzposoje	1	1	0	0	0
Knjiznica.Izposoja.getGradivo	1	1	0	0	0
Knjiznica.Izposoja.izracunajZnesek	1	1	0	0	0
Knjiznica.Izposoja.vrniBonusTocke	1	1	0	0	0
Knjiznica.Knjiga	2	29			
Knjiznica.Knjiga.Knjiga	1	1	0	0	1
Knjiznica.Knjiga.getTipGradiva	1	1	0	0	0
Knjiznica.Knjiga.vrniBonusTocke	2	8	2	1	1
Knjiznica.Knjiga.vrniCenoIzposoje	2	6	1	1	1
Knjiznica.Revija	2	21			
Knjiznica.Revija.Revija	1	1	0	0	1
Knjiznica.Revija.getTipGradiva	1	1	0	0	0
Knjiznica.Revija.vrniCenoIzposoje	3	9	3	1	1
Knjiznica.Zbornik	1	18			
Knjiznica.Zbornik.Zbornik	1	1	0	0	1
Knjiznica.Zbornik.getTipGradiva	1	1	0	0	0
Knjiznica.Zbornik.vrniCenoIzposoje	2	6	1	1	1

nacij metrik, določiti mejne ter signalne vrednosti posameznih metrik, upoštevati tudi značilnosti projekta, v sklopu katerega apliciramo predlagani pristop, ter preizkusiti uporabnost pristopa na večjem projektu.

7 Literatura

[1] Fowler, M. (1999). *Refactoring: improving the design of code*, Addison-Wesley, 1999.

[2] Grajfoner, U., Repinc, P. *Refactoring – Preoblikovanje programske kode*, Uporabna informatika, letnik VIII, štev. 4, 2000, str. 231–236.

[3] Simon, F., Steinbrückner, F., Lewerentz, C. *Metrics Based Refactoring*, European Conf. Software Maintenance and Reengineering, IEEE Computer Society Press, 2001, str. 30–38.

[4] Simon, F., Steinbrückner, F., Lewerentz, C. *3D-Spring Embedder for Complete Graphs*, Technical Report 11/00, Computer Science Reports, Technical University Cottbus, 2000.

[5] Ratzinger, J., Fischer, M., Gall, H. *Improving Evolvability through Refactoring*, Vienna University of Technology, 2004.

[6] Ratzinger, J., Fischer, M., Gall, H. *Evolens: Lens-view visualizations of evolution data*, Technical Report, Vienna University of Technology, December 2004.

[7] Abreu, F. B., Carapuça, R., *Object-oriented Software Engineering: Measuring and Controlling the Development Process*, Proceedings of the 4th International Conference on Software Quality, ASQC, McLean, VA, USA, Oktober 1994.

[8] Abreu, F. B., Goulão, M., Esteves, R. *Toward the design quality evaluation of object-oriented software systems*, Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA Oktober 1995, str. 44–57.

Marko Tekavc je podiplomski študent na Inštitutu za informatiko Fakultete za elektrotehniko, računalništvo in informatiko Univerze v Mariboru, kjer je tudi zaposlen kot asistent za področje informatike. Sodeluje na različnih projektih, tako aplikativnih kot tudi znanstveno-raziskovalnih. Raziskovalna področja, s katerimi se ukvarja, pokrivajo objektno orientirane tehnologije, preoblikovanje programske kode, storitvene arhitekture (SOA), spletne storitve in kompozicije poslovnih procesov (BPEL).

Marjan Heričko je izredni profesor na Inštitutu za informatiko na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Je vodja laboratorija za informacijske sisteme in strokovni koordinator slovenske tehnološke platforme za programsko opremo in storitve. Njegovo raziskovalno-razvojno delo obsega vse vidike razvoja sodobnih informacijskih rešitev s poudarkom na metodologijah razvoja, metrikah in razvojnih okoljih. Svoje izkušnje je predstavil v številnih prispevkih na domačih in tujih konferencah ter revijah. Je tehnični koordinator aktivnosti centra COT ter predsednik konferenc OTS in CIS. Diplomiral, magistriral in doktoriral je na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru.