REFACTORING – PREOBLIKOVANJE PROGRAMSKE KODE

Uroš Grajfoner, Peter Repinc
OdaTeam d.o.o.
Meljska 36, SI-2000 Maribor
e-pošta: uros@odateam.com, pero@odateam.com
URL: http://www.odateam.com

Izvleček

Spreminjanje obstoječe programske kode zaradi dodajanja nove funkcionalnosti se lahko izvaja z natančno izbranimi in definiranimi koraki. Takemu procesu pravimo preoblikovanje programske kode. Izvaja se združeno z drugimi razvijalskimi aktivnostmi, pri ekstremnem programiranju pa je bistveni del razvojnega procesa. Z njim programska struktura postane razumljivejša, jasnejša, pravilnejša in fleksibilnejša za dodajanje nove funkcionalnosti. Proces je podprt z avtomatiziranim testiranjem. V najboljšem primeru se proces izvaja avtomatsko z orodji za preoblikovanje. Preoblikovanje zbija ceno spremembe obstoječega sistema in povečuje njegovo preglednost.

Abstract

Refactoring is a process of adding new functionality by changing the existing program code. It can be performed by exactly chosen and defined steps. It is practiced together with other design activities, while in extreme programming it is an essential part of the development process. With refactoring, the program structure becomes more understandable, clearer, more correct and flexible for adding new functionalities. Automatic testing by refactoring tools support the process. Refactoring reduces the cost of changing the existing system and improves its transparency.

1. UVOD

Ko se začne zapletati...

V življenjskem ciklu informacijskega sistema naletimo na dele informacijskega sistema, ki več ne odsevajo dejanskega stanja v realnosti. Dokler taki deli sistema delujejo pravilno in ni razlogov, da bi na njih izvajali spremembe, je na videz vse lepo in prav.

Ko se v realnosti pojavijo spremembe, se morajo te spremembe odražati tudi na programski kodi. Zaradi spreminjanja sistema skozi čas se zgradba in namembnost sistema zameglita. K temu še največ doprinesejo kratkoročni in hitri posegi v sistem, ki so posledica zahtev uporabnikov, časovne stiske, ipd. Zaradi tega se dogaja, da določeni deli sistema vsebujejo veliko začasnih rešitev, naknadnih dograditev in jih je posledično vedno težje vzdrževati, so vedno bolj nepregledni, vedno pogosteje se dogaja, da se nove funkcionalnosti ne da enostavno dodati.

Zaradi tega pridemo do logičnega zaključka, da je sistem treba preoblikovati, da bi lahko ob dodajanju nove funkcionalnosti še vedno ustrezno deloval, ali pa da bi ga po določenem časovnem obdobju sploh še lahko razumeli.

Dodajanja nove funkcionalnosti se lahko lotimo na različne načine. En način je, da določen podsistem zgradimo znova. Tak pristop pa je lahko zelo drag, dolgotrajen in zelo tvegan. Druga možnost je kopiranje in prirejanje delov sistema in s tem razširjanje sistemskih zmogljivosti. Vendar so te zmogljivosti le na videz velike, saj sistem s takimi posegi postane prehitro prevelik, napake se hitreje množijo, implementacija sistema več ne ustreza načrtu, stroški inkrementalnih popravil pa se pomnožijo. Najbolj sprejemljiva rešitev je srednja pot: preoblikovanje sistema v majhnih, obvladljivih korakih. Na tak način je vpogled v problem boljši, bolj osredotočen, arhitektura sistema se s takimi popravili izboljšuje, s tem pa je olajšano tudi dodajanje novosti.

2. Lastnosti preoblikovanja

Izhodiščne informacije

2.1 Definicija preoblikovanja

Preoblikovanje je proces spreminjanja programske kode tako, da se obnašanje sistema navzven ne spremeni, spremeni in izboljša(!) pa se notranja struktura sistema. Je sistematičen način »prečiščevanja« kode in zmanjševanja možnosti napak.

Namen preoblikovanja je spreminjanje sistema tako, da ga je lažje razumeti in dopolnjevati. Preoblikovanje

spreminja način gradnje informacijskih sistemov iz ustaljenega zaporedja »načrtovanje, gradnja, vzdrževanje« v delovni proces, kjer načrtovanje, gradnja in vzdrževanje niso več ločeni procesi, ampak se prepletajo in ponavljajo v celotnem življenjskem ciklu informacijskega sistema. S pomočjo preoblikovanja lahko sistem načrtujemo tudi, ko je že implementiran.

2.2. Kdaj preoblikovati

Idealno bi bilo imeti programski sistem z brezhibno arhitekturo, s čudovito strukturirano in jasno programsko kodo. Število podjetij, ki si lahko privoščijo tak sistem, ki hkrati deluje brez časovnih omejitev, limitira proti nič. Na žalost. Čas in denar sta vedno omejeni dobrini. Zaradi tega preoblikovanje ni proces, ki bi se nenehno odvijal skozi celotno življenjsko dobo nekega informacijskega sistema. Pomembno je prepoznati trenutke, ko je preoblikovanje kode najbolj smiselno.

2.2.1. Dodajanje nove funkcionalnosti

Najpogostejši vzrok za preoblikovanje obstoječe kode je dodajanje nove funkcionalnosti. Zahteve uporabnikov praviloma niso popolne in dokončne. Okolje delovanja programskega sistema se vedno spreminja. Znanje razvijalcev in načrtovalcev je različno. Vsi ti vzroki botrujejo rojevanju potrebe po novi fukcionalnosti, ki jo mora sistem vsebovati.

Pri dodajanju nove funkcionalnosti vedno naletimo na programsko kodo, ki jo je treba na neki način spremeniti. Verjetnost, da to kodo popolnoma poznamo, je majhna. Da bi kodo lahko bolje razumeli, jo lahko preoblikujemo. S tem izboljšamo njeno izraznost, kar vsekakor pride prav, ko naslednjič naletimo na isto kodo.

Drugi dober razlog za preoblikovanje je struktura kode, ki ne omogoča preprostega dodajanja nove funkcionalnosti. Seveda lahko vedno dogradimo obstoječo kodo in z določenimi posegi in izjemami dosežemo, da koda deluje tudi z novo funkcionalnostjo, vendar na ta način še povečamo nerazumljivost kode, naše življenje bo pa precej manj lepo, ko bomo spet naleteli na njo. Svet si lahko naredimo prijaznejši, če v takem primeru kodo najprej preoblikujemo tako, da je novo fukcionalnost enostavno dodati. In šele nato dodamo novo funkcionalnost.

2.2.2. Popravljanje napak

Dober razlog za preoblikovanje kode je popravljanje napak. Kodo najprej preoblikujemo, da bolje razumemo kontekst. Ko je sistem razumljivejši, je napako mnogo lažje najti in jo zatem popraviti.

Povedano drugače, napaka, ki jo popravljamo, je verjetno posledica nejasnosti kode. Preoblikovanje je torej tudi način, s katerim si zaradi boljše preglednosti zagotavljamo boljše razumevanje kode in s tem zmanjšujemo verjetnost napak.

2.2.3. »V tretje gre rado«

To smernico je dal Don Roberts Martinu Fowlerju [1]. Prvič napišemo kodo in upamo na najbolje. Ko drugič ustvarimo nekaj podobnega, se očitnega podvajanja zavedamo, morda celo ustrašimo, a večinoma zaradi različnih pritiskov kodo vseeno podvojimo. Ko ustvarimo podobno kodo že tretjič, je čas za preoblikovanje.

2.2.4. Preoblikovanje ob pregledih kode

Pregledi kode omogočajo prepoznavanje namer programerjev. Prepoznava se slabo načrtovanje, dobre ideje, odkrivajo se napake. Pri pregledih kode razvijalci z manj izkušnjami pridobivajo novo znanje. Pri tem se velikokrat zbirajo ideje in predlogi kako kodo spremeniti, da bi bolje služila svojemu namenu.

S preoblikovanjem lahko take predloge udejanjimo. Rezultat je drugačna struktura programa, ki jo lahko zavržemo, če se izkaže kot slabša alternativa, ali pa jo uporabimo kot izhodišče za naslednjo izboljšavo. Le-to bi si seveda težko predstavljali, če ne bi naredili prve spremembe. Poleg tega pri pregledu kode ostane samo pri predlogih, pri preoblikovanju pa lahko rezultate novih idej vidimo pred sabo zelo kmalu. Če je nov načrt dober, ga lahko obdržimo in sistem bo boljši. Če ne bo, se lahko vrnemo na prejšnje stanje in bogatejši bomo za novo izkušnjo.

Pri ekstremnem programiranju je tak način preoblikovanja potenciran, saj se programira v parih, kjer je preoblikovanje konstanten del razvojnega procesa. Razvijanje v parih zato deluje kot nenehen pregled kode s preoblikovanjem.

Seveda pa preoblikovanje ni univerzalen odgovor na vsako težavo. Prav tako ga je na nekaterih področjih težavno uporabiti, so tudi primeri, ko sploh ni priporočljivo. Problematično je lahko recimo preoblikovanje baze podatkov (oz. objektnega modela), kjer se moramo v večini primerov soočiti s konvertiranji v nova stanja podatkov. Prav tako lahko nastopijo težave, ko spremenimo vmesnik komponente, ki se pogosto uporablja.

V nekaterih primerih pa preoblikovanje enostavno ne pomaga in je najbolje kak del sistema odstraniti in implementirati znova. Po našem mnenju so to sistemi z resnimi napakami v samem načrtu ali sistemi, ki so preveč nestabilni za produkcijsko uporabo. Novo gradnjo kakega sistema si, če je to mogoče, lahko zamislimo tudi kot dolgotrajno globalno preoblikovanje.

2.3. Ko »zavohamo« priložnost...

V prejšnjem poglavju so opisane različne situacije, pri katerih je preoblikovanje potrebno ali priporočljivo.

Ni pa konkretnih navodil, na katerih mestih naj se lotimo dela. Prav tako nam pri samem začetku, ko se lotevamo preoblikovanja prvič, ni popolnoma razvidno, kje naj rešujemo težave najprej. Čez čas odkrijemo, da smo dobili »nos« za določene konstrukte kode, kjer imamo precej dobro zamisel, kako bi ta koda lahko delovala bolje in kako bi lahko bila jasnejša.

V naslednjih nekaj vrsticah so opisana nekatera mesta v kodi, kjer je (zelo) očitno, da je potrebno preoblikovanje. Veliko primerov je gotovo prepoznavnih iz vsakdanjega obdelovanja kode.

- *Podvojena programska koda* – najpogostejši razlog za preoblikovanje. Če se pojavi pri istem razredu, je rešitev ponavadi izsek podvojene kode in definiranje nove metode. Pri razredih v hierarhiji se skupno uporabljana koda premakne na ustrezen abstraktni razred. Če povezav med razredi ni, lahko naredimo celo nov razred, ki opravlja naloge podvojene kode.
- *Predolga metoda* – Merila za dolžino metod v objektnih programih so različna. V večini primerov pa se predolgih metod lahko rešimo z izsekom dela kode in definiranjem nove metode. V drugih primerih lahko veliko množico parametrov zamenjamo z objektom. Metodo lahko predstavimo tudi kot nov objekt ali hierarhijo objektov.
- *Prevelik razred* – Razred združuje preveč funkcionalnosti in informacij, ima »kompleks švicarskega noža«. Rešitev je dekompozicija na več manjših razredov, odgovornih za posamezne naloge.
- *Dolg nabor parametrov* –, ki jih potrebuje neka metoda, lahko nadomestimo s povpraševalnimi stavki, predajanjem objekta kot parametra itd.
- *Raznolike spremembe* -, ki jih opravljamo pri vedno istem razredu, sugerirajo kreiranje več različnih razredov z jasnimi odgovornostmi.
- *Poseg s šibrovko* - Pri določenih spremembah je potrebno spreminjati kodo na vedno enakih mestih v različnih delih sistema. Rešitev je v definiciji centralnega razreda ali metode, kjer se izvaja vsakokratna sprememba.
- *Zavidanje lastnosti* – se pojavlja pri razredih, ki opravljajo svoje delo na drugih objektih in jim »zavidajo« njihove lastnosti. Najpogostejša rešitev je premik kode ali dela kode k pravemu lastniku.
- *Kepe podatkov* – podatki, ki večinoma nastopajo skupaj, se morajo pravzaprav združiti v skupen objekt, ki kasneje pridobi še nekatere odgovornosti in operacije.
- *»IF« in »CASE« stavki* – v objektnih jezikih se pogojnih izrazov lahko lepo izognemo s polimorfizmom in hierarhijo.
- *Zavrnjena zapuščina* – podrazredi ne potrebujejo lastnosti nadrazredov. Očitno je problem v napačno zastavljeni hierarhiji (in pripadajočih metodah).
- *Komentarji* – pričajo o nejasni in nezgovorni kodi. Rešitev za tako kodo je preimenovanje imen metod in izrezovanje delov kode in primerno poimenovanje novih metod.

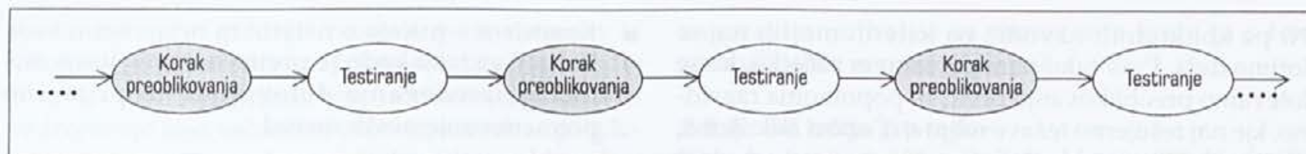
2.4. Izvajanje in načini preoblikovanja

Preoblikovanje je resen poseg v programsko kodo, ki ne spreminja njene funkcionalnosti, spreminja pa notranjo strukturo sistema. Pri vsakem preoblikovanju pa lahko pride do kake napačne konstelacije in novo ustvarjena koda več ne dela enakih stvari kot prejšnja. Zaradi varnosti preoblikovanje razdelimo na čim manjše korake, ki so točno definirani. Pri teh korakih natančno poznamo zaporedje dogodkov, ki bodo spremenili kodo. S tem zagotovimo, da ne bomo pozabili opraviti kakega pomembnega dela preoblikovanja, po drugi strani se s standardnimi operacijami varujemo pred napakami.

Taki standardni koraki se delijo na več sklopov [1], kot na primer:

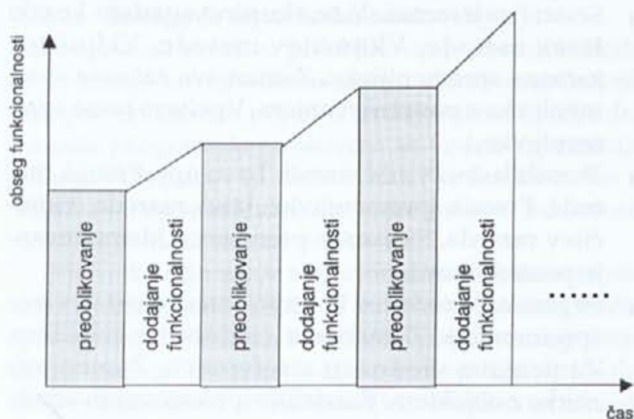
- *Sestavljanje metod.* V to skupino spadajo koraki Izsek metode, Vključitev metode, Vključitev začasne spremenljivke, Zamenjava začasne spremenljivke s povpraševanjem, Vpeljava jasne spremenljivke...
- *Premiki lastnosti med razredi.* To so npr. Premik metode, Premik spremenljivke, Izsek razreda, Vključitev razreda, Skrivanje prenosov, Odstranjevanje posrednikov...
- *Organizacija podatkov* kot npr. Interno ograjevanje spremenljivk, Zamenjava vrednosti z objektom, Zamenjava vrednosti z referenco, Zamenjava zbirke z objektom, Razdelitev prikazovanih objektov...
- *Poenostavitve pogojnih izrazov.* Razstavitev pogoja, Zamenjava pogoja z metodo, »Izpostava« podvojenih posledic pogojev, Odstranitev kontrolnih zastavic....
- *Poenostavitve klicev metod* so recimo Preimenovanje metode, Dodajanje parametra, Odstranitev parametra, Ločitev povpraševanja od spreminjanja, Parametriziranje metode, Ohranitev celega objekta...
- *Obvladovanje generalizacije* vsebuje korake kot Potegni spremenljivko gor, Potegni metodo gor, (oboje tudi dol), Premik konstruktorja gor, Izdelava nadrazreda, Izdelava podrazreda, Izdelava vmesnika, Porušenje hierarhije...

Osnova za varno izvajanje posameznih korakov preoblikovanja je avtomatizirano testiranje. Pred začetkom preoblikovanja morajo biti na voljo testni primeri, ki preverjajo pravilnost izvajanja dela sistema. Preoblikovanje se izvršuje korak za korakom, po vsakem koraku pa sledi zaganjanje testov in s tem dokazovanje, da sistem še vedno deluje enako.



Slika 1. Potek preoblikovanja

Celotna implementacija se v bistvu deli na dodajanje funkcionalnosti in preoblikovanje. Ko dodajamo funkcionalnost, dodajamo tudi nove testne primere. Ko preoblikujemo, ohranjamo enako funkcionalnost, prav tako tudi testne primere (razen v primeru, ko odkrijemo napako. Takrat se definirajo novi testni primeri, ki odkrivajo napako). Tako dodajanje funkcionalnosti kot preoblikovanje pa pomenita dodajanje določene vrednosti v programski sistem. Z dodajanjem funkcionalnosti sistem zmore več. Z preoblikovanjem je sistem razumljivejši, pravilnejši in bolj »gostoljuben« za novo funkcionalnost.



Slika 2. Vloga preoblikovanja v razvojnem procesu

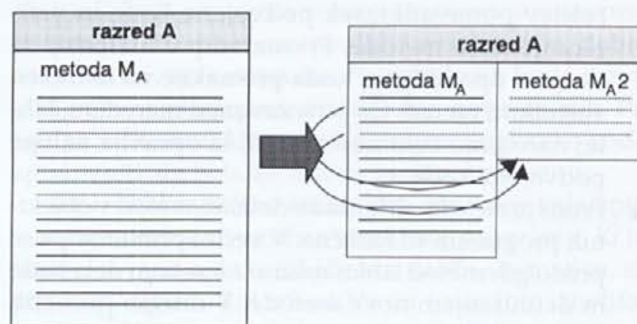
3. Primer enostavnega preoblikovanja

Preoblikovanje v praksi

Vsakdo, ki se ukvarja z razvojem programskih sistemov, je že naletel na odseke kode, ki so najprej povzročali glavobol, šele nato je sledilo njihovo razumevanje. V takih primerih si želimo, da bi koda bila jasnejša, bolje strukturirana, enostavneje zapisana itd. Z majhnimi koraki preoblikovanja lahko tako kodo prikrojimo, da bo njena izraznost občutno izboljšana.

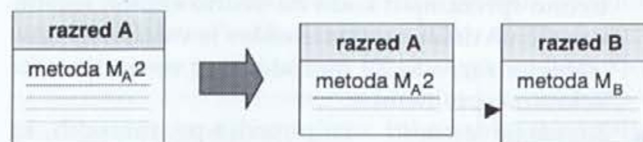
Vzemimo na primer metodo, na katero smo naleteli med dodajanjem neke funkcionalnosti. Metoda vsebuje veliko vrstic kode in je nepregledna. Zaradi neprimerno prevelike dolžine kode »zavohamo«, da bi jo lahko preoblikovali (vonj *Predolga metoda*). Privzemimo, da se metoda imenuje M_A in da pripada razredu A.

Najprej opazimo, da sta dva dela metode skoraj identična (*Podvojena programska koda*). Uporabimo korak »Izsek metode« in kodo, ki se ponavlja, izsekamo iz metode M_A ter ustvarimo novo metodo M_A2. Metoda M_A zdaj kliče metodo M_A2 na mestih, kjer je bila koda podvojena. Metoda M_A2 sedaj vsebuje podvojeno kodo. Rezultat je prikazan na sliki 3. Puščice prikazujejo klic metod.



Slika 3. Izsek podvojene kode

Na tem mestu opazimo, da metoda M_A2 pravzaprav vseskozi obdeluje le objekt razreda B, pozna objektovo notranjo strukturo in/ali celo kliče privatne metode razreda B. Večina metode M_A2 pravzaprav obdeluje samo objekt razreda B (*Zavidanje lastnosti*). Uporabimo koraka Izsek metode, kjer kodo, ki se ukvarja samo z razredom B, izsekamo iz metode M_A2, nato pa še Premik metode, s katerim novo ustvarjeno metodo premaknemo k razredu B in jo poimenujemo kot metoda M_B.



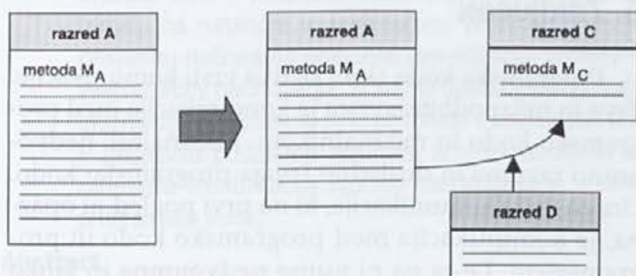
Slika 4. Premik kode k pravemu lastniku

Vrnemo se na metodo M_A, ki se še vedno zdi predolga (*Predolga metoda*). Opazimo odsek metode, ki se pretirano ukvarja z objektom razreda C (*Zavidanje lastnosti*). Omenjeni odsek izsekamo iz metode M_A (Izsek metode), nato pa ga preselimo v metodo M_C k razredu C (Premik metode), kamor pravzaprav spada.

Čeprav zdaj metoda M_A kliče metodo M_C, pa jo mora klicati z daljšim nizom parametrov (*Dolg nabor parametrov*). Poleg tega opazimo še, da ti parametri

v razredu A velikokrat nastopajo skupaj in so skupaj uporabljani še v nekaterih drugih metodah (*Kepe podatkov*).

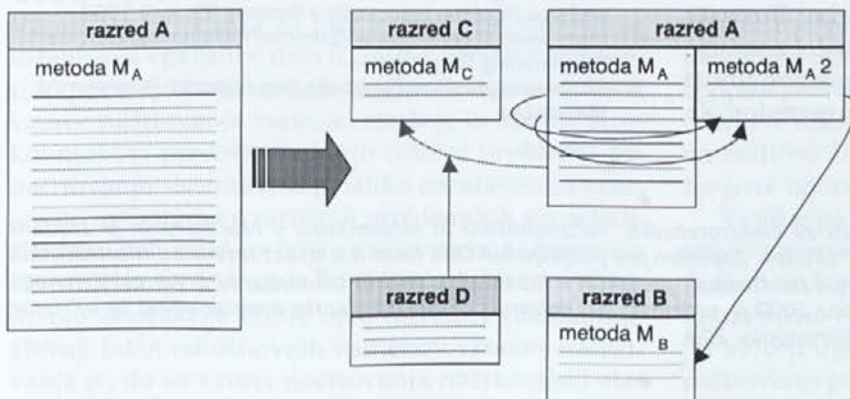
V tem primeru uporabimo korak Izsek razreda, kjer skupine objektov, ki večinoma nastopajo skupaj, združimo v skupen objekt tako, da definiramo nov razred D s skupino objektov v njegovih instančnih spremenljivkah. Zatem spremenimo metodo M_C tako, da za svoje delovanje več ne zahteva prej omenjene množice parametrov, ampak objekt razreda D (Vpeljava objekta kot parametra). Prav tako moramo spremeniti vso kodo pri razredu A (med drugim tudi metodo M_A), kjer se nanašamo na skupino parametrov, in jo zamenjamo z uporabo objekta razreda D. Preglednost in jasnost posameznih metod se je tako neprimerno povečala.



Slika 5. Objekt namesto množice parametrov

Po izvršenih korakih preoblikovanja smo končno zadovoljni z velikostjo metode M_A. Programska koda, ki je predtem bila zgoščena na enem mestu, je zdaj razdeljena po sistemu, glede na njeno namembnost. Naredili smo kar nekaj sprememb. Oglejmo si njihov skupni učinek.

Na prvi pogled je rezultat mnogo bolj zapleten. Taka sestava programa pa ima zdaj neprimerno več dobrih strani. Metoda M_A, ki je bila naš začetni problem, je zdaj krajša in preglednejša. Prav tako so tudi vsi ostali odseki kode (nove metode) zdaj jasnejši.



Slika 6. Rezultat preoblikovanja

Nove metode so manjše in obvladljivejše. Pripadajo k tistim objektom, katerih strukturo in lastnosti najbolje poznajo. Namembnost metod je bolj razdrobljena in pravilneje porazdeljena. Vsaka metoda opravlja bolj specializirano funkcijo. Vsaka sprememba kode se zdaj lahko izvrši na mestu, ki je za njo najbolj primerno, t.j. v metodi, ki je »specializirana« za izvajanje določenega opravila. Pred izvedbo preoblikovanja bi se lahko zgodilo, da bi zaradi manjše spremembe morali spreminjati metodo M_A na več mestih. Tukaj smo verjetnost takega dogodka občutno zmanjšali.

Seveda moramo vseskozi zagotavljati, da program še vedno deluje. Po vsakem koraku je potrebo izvesti ustrezna testiranja programa, ki zagotavljajo, da program še vedno deluje.

Pri preoblikovanju lahko naletimo tudi na logične napake, ki jih prej zaradi prevelike množine podatkov nismo uspeli opaziti. Ko jih odpravimo, napišemo nove testne primere, ki take napake odkrivajo.

Prikazani način dela, izmenjujoči koraki preoblikovanja s sprotnim testiranjem, ima še eno dobro stran. Po vsakem opravljenem koraku se zavedamo, da je koda razumljivejša, čitljivejša, lepša za ponovno uporabo. Po vsakem testiranju pa si zagotovimo, da je še vedno pravilna. Lahko se zgodi, da začnemo preoblikovati šele proti koncu delovnega dne. Po nekaj izvedenih korakih spoznamo, da načrtovanih sprememb ne bomo mogli izpeljati do konca. Kljub temu opravljenega dela ni potrebno zavreči. Vsak korak je natančno definiran in preverjen s testnim orodjem, zato sistem še vedno deluje pravilno in je v vsakem primeru boljši, kot je bil pred začetkom preoblikovanja. Po vsakem koraku preoblikovanja lahko torej zaključimo z delom in nadaljujemo naslednjič.

4. Orodja za preoblikovanje

Olajšanje življenja

Natančno definirani koraki preoblikovanja programske kode [1] nudijo razvijalcem gotovost, da bodo koraki izvedeni pravilno in popolno. S tem se do določene mere lahko znebimo negotovosti, ki lahko spremlja razvijalce pri preoblikovanju. Proti taki negotovosti imamo še eno orožje: avtomatizirano testiranje.

Čeprav je preoblikovanje podprto s sistematiko in testiranjem, pa se ga razvijalci vseeno pogosto izogibajo. Razlog je preprost. Preoblikovanje zahteva svoj čas. Razvijalci se ga izognejo, ker je na prvi pogled predrago.

Na tem mestu ni odveč poudariti, da je preoblikovanje proces, ki

dolgoročno poceni dodajanje funkcionalnosti. Z njim odkrivamo napake in izboljšujemo programsko kodo. Brez preoblikovanja bi vsako naknadno dodajanje funkcionalnosti bilo vedno težje izvedljivo in s tem vedno dražje. Z uporabo preoblikovanja je razvoj programja na začetku navidezno upočasnjen, vendar se s časom praksa ustali, dodajanje nove funkcionalnosti ima še vedno relativno nizko ceno, napredek je zmernejši, a konstantnejši in hitrejši.

Orodja za preoblikovanje nudijo avtomatsko izvajanje korakov preoblikovanja. S tem je vklop preoblikovanja v vsakdanji razvojni proces močno olajšan. Vsak korak preoblikovanja, ki se mu razvijalec lahko izogne zaradi pomanjkanja časa, je zdaj lahko opravljen hitro in avtomatsko, z označevanjem kode in izbiro ustreznih ukazov. S tem odpadejo tudi preverjanja pravilnosti predajanja novih parametrov, osveževanja klicateljev preimenovanih metod in drugo. Proces izvedbe posameznega koraka preoblikovanja traja nekaj sekund, medtem ko bi manualno izvajanje enakega postopka lahko trajalo kar nekaj minut. Poleg tega postane vsakokratno testiranje nepotrebno, saj se vse potrebno delo izvede avtomatsko. Kar pa še ne pomeni, da s preoblikovanjem postane testiranje odvečno. Preprosto ga ne potrebujemo tako pogosto.

Z orodji se cena preoblikovanja zniža, posredno s tem pade tudi cena razvojnih napak. Zaradi tega se lahko izognemo pogosto zapletenim, vnaprej pripravljenim razvojnim načrtom, ki nastajajo zaradi pomanjkanja zahtev. Takšni razvojni načrti vsebujejo fleksibilnosti, ki se s kasnejšo uporabo izkažejo kot nepotrebne. Tako se povečuje kompleksnost programja. Brez poznavanja preoblikovanja pa bi bilo nefleksibilne sisteme zelo drago spreminjati. Z avtomatiziranim preoblikovanjem si lahko privoščimo enostavnejše načrtovanje, saj je spremembe lažje izvajati, razširjanje načrtov in dodajanje funkcionalnosti ni več tako cenovno potratno.

Na področju preoblikovalnih orodij na žalost vlada zaenkrat velika praznina. V integriranih razvojnih okoljih kot so recimo okolja za Smalltalk, je zaenkrat razvito orodje poimenovano Refactoring Browser (Preoblikovalni brskalnik), ki pa se je pričel znatneje uporabljati šele, ko je bila njegova funkcionalnost vključena v vsakdanji brskalnik knjižnic razredov. Za okolja kot je npr. Java, kjer se programska koda vpisuje tako rekoč v preprost urejevalnik besedila, se medsebojno prepletene reference na objekte in metode ne shranjujejo, zato je izdelava orodja za preoblikovanje še posebej otežena. Nekatera naprednejša okolja, kot je npr. IBM-ov VisualAge for Java, posnemajo smalltalkovsko dinamično osveževanje programskega repozitorija.

Orodje za preoblikovanje naj bi torej na neki način obvladovalo programske reference med razredi, moralo bi delovati natančno, dovolj hitro za vsakdanjo uporabo, omogočati preklice posameznih korakov preoblikovanja (»korakanje nazaj«), še najlepše pa bi bilo, ko bi bilo tako orodje integrirano v razvojno okolje samo.

5. Zaključek

Končno

Programska koda skrbi za dve vrsti komunikacije. Prva in neizpodbitna vrsta je komunikacija med programsko kodo in računalnikom. Računalnik nedvoumno razume in dosledno izvaja programsko kodo. Druga vrsta komunikacije, ki na prvi pogled ni opazna, je komunikacija med programsko kodo in programerjem. Le-ta pa ni nujno nedvoumna in lahko funkcionira zelo slabo. Z preoblikovanjem izboljšujemo predvsem komunikacijo med programsko kodo in programerjem. Hkrati pa skrbimo, da je naš informacijski sistem jasna in nedvoumna slika realnosti.

LITERATURA

1. FOWLER, Martin, "Refactoring", Addison-Wesley, Reading, 1999
2. BECK, Kent, "Extreme programming explained", Addison Wesley, Reading, 1999
3. FOWLER, Martin, KENDALL, Scott, "UML Distilled", Addison-Wesley, Reading, 1997
4. ROBERTS Don et al., "Why Every Smalltalker Should Use the Refactoring Browser", The Smalltalk Report, letnik 6, številka 10, september 1997
5. ROSTAHER, Matevž, KLINE, Andrej "Proces skupinskega razvoja programske opreme (Extreme programming)", Zbornik srečanja Objektna tehnologija v Sloveniji 99, junij 1999
6. http://www.c2.com/cgi/wiki?ExtremeProgramming, Extreme Programming Discussion
7. extremeprogramming@egroups.com, Extreme Programming Mailing List

♦

Uroš Grajfoner je diplomiral na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru in je študent podiplomskega študija informatike na tej fakulteti. Zaposlen je v podjetju FJA Oda Team d.o.o. kot razvijalec informacijskih sistemov. V zadnjih petih letih je sodeloval pri številnih projektih v Smalltalku, Javi in pri uvajanju prvin ekstremnega programiranja v proces razvoja. V oktobru 2000 je sodeloval pri delavnici "From Moderate programming to eXtreme Programming" na konferenci OOPSLA, Minneapolis, ZDA.

♦

Peter Repinc je študent računalništva in informatike na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru. Od leta 1998 je zunanji sodelavec podjetja OdaTeam d.o.o. Sodeloval je pri projektih v Smalltalku, Javi in XP.