Razvoj softvera 2

Refaktorisanje

(Code Complete)

Tijana Jordanov

1011/2014

1. Filozofija evolucije softvera

"There is no code so big, twisted, or complex that maintenance can't make it worse." - Gerald Weinberg

- Ako je neophodno izmeniti kod, treba težiti tome da izmene poboljšaju kod tako da buduće izmene budu manje i jednostavnije.

- Glavno pravilo ravoja softvera (The Cardinal Rule of Software Evolution): Razvoj treba da poboljša unutrašnji kvalitet programa.

2. Refaktorisanje

Glavna strategija za zadovoljavanje glavnog pravila razvoja softvera je refaktorisanje.

Prema jednoj od definicija refaktorisanje je niz promena u unutrašnjoj strukturi programa koje omogućavaju lakše razumevanje i jednostavnije izmene ne utičući na ponašanje programa (Martin Fowler)

Reč refaktorisanje (refactoring) nastala je od reči “factoring” koja je u strukturnom programiranju označavala dekomponovanje programa na sastavne delove. (Larry Constantine)

3. Razlozi za refaktorisanje

1. Ponavljanje koda:
   - Moraju se praviti paralelne promene
   - DRY princip – Don't Repeat Yourself
   - “Copy and paste is a design error” (McConnell 1998.)
2. Dugačak metod:
   - Retko kada je potreban metod tako dugačak da se ne može ceo prikazati na ekranu, bez skrolovanja
   - Dobar sistem treba da ima više metoda od kojih svaki radi jednu stvar i to dobro.
   - Razlaganje jednog metoda na više poboljšava razumevanje, debagovanje i menjanje koda
3. Dugačka ili preduboko ugnježdena petlja:
   - Petlje su dobri kandidati za metode
   - Pravljenje metoda umesto petlje može učiniti kod razumljivijim
4. Paralelne hijerarhije nasleđivanja
   - Dve hijerarhije se održavaju paralelno – za svaku klasu jedne hijerarhij postoji odgovarajuća klasa druge
   - Rešenje: premeštanje podataka i metoda

5. Siromašna kohezija klase
   - Ako se pravi mnogo objekata neke klase različite namene, tu klasu treba razdvojiti na više manjih klasa
6. Interfejs klase ne obezbeđuje dosledan nivo apstraktnosti
   - Usled promena do kojih dolazi tokom vremena interfejs klase se transformiše
7. Dugačka lista argumenata
   - Bolje je imati više malih, dobro definisanih metoda koji nemaju dugačku listu argumenata
8. Divergentne promene
   - Klasa se menja iz više različitih razloga
   - Tada se prilikom promene klase menja samo jedan njen deo, retko oba
   - Ovo je znak da klasu treba podeliti bar na 2 klase različitih odgovornosti
9. Distribuirana apstrakcija
   - Kada jedna vrsta promene utiče na isti skup klasa, potrebno je reorganizovati te klase td. jedna promena utiče na jednu klasu
   - Primer: dodavanje novog izlaza zahteva menjanje 15 klasa

10. Switch naredba
    - Problem nastaje ako imamo više switch naredbi u jednoj klasi i svaki put kada menjamo jednu case granu jedne switch naredbe, moramo je promeniti i u ostalim switch naredbama u toj klasi
    - Ovo se rešava hijerarhijom klasa
11. Povezani podaci koji se zajedno(istovremeno) koriste nisu organizovani u klasu
    - Ako se konstantno koristi ista grupa podataka, treba razmotriti njihovo smeštanje u klasu
12. Metod više koristi delove neke druge klase nego svoje
    - Ovo sugeriše da metod treba premestiti u tu drugu klasu i pozvati ga iz stare klase
13. Poplava primitivnih podataka
    - Nekada je bolje umesto korišćenjna primitivnog tipa napraviti klasu
    - Primer: Money, Temperature

14. Lenja klasa
    - Nekada nakon refaktorisanja dobijemo klasu koja nema značajnu funkciju. Tada treba razmotriti dodeljivanje njenih funkcionalnosti nekoj drugoj klasi i njeno brisanje
15. Lanac poruka
    - Neki metodi samo prosledjuju podatke drugim metodima
    - Treba razmotriti uklanjanje metoda posrednika
16. Posrednik
    - Ako je jedina uloga klase da poziva metode drugih klasa,
    - Treba razmotriti njeno uklanjanje
17. Nepoželjna bliskost
    - Neka klasa suviše često pristupa privatnim delovima druge klase
18. Metod ima neodgovarajuće ime
    - U ovom slučaju treba promeniti ime metoda tamo gde je definisan i svuda odakle se poziva
19. Podaci koji su public
    - Ako klasa ima public podatke, to narušava enkapsulaciju i treba ih sakriti

20. Podklasa koristi samo mali deo metoda roditeljske kalse
    - Ovo ukazuje da veza između klasa postoji jer su podklasi potrebni metodi roditeljske klase, ne zato što ona logički zavisi od nadklase
    - Bolja enkapsulacija se može postići ako klasa koja je bila potomak sadrži instancu bivše roditeljske klase
21. Komentari služe da pojasne komplikovan deo koda
    - Komentare ne treba koristiti za pojašnjavanje loše napisanog koda
    - “Don't document bad code – rewrite it” (Kernighan and Plauer 1978)
22. Koriste se globalne promenljive
    - Treba izbegavati korišćenje globalnih promenljivih
    - One se mogu zameniti pristupnim metodima
23. Program sadrži kod koji će možda nekad zatrebati
    - Treba izbegavati pisanje ovakvog koda.
    - Trenutni programer teško može napraviti tačnu procenu zahteva koje će taj kod trebati da ispuni
    - Budući programeri mogu misliti da kod radi mnogo bolje nego što je slučaj

24. Metod zahteva podešavanje dela koda pre poziva ili brisanje dela koda nakon izvršavanja

- Metodu prosleđujemo objekat kom treba postaviti sva polja pre poziva metoda
- Nakon izvršavanja metoda objekat se briše
- Klasa ima specijalan konstruktor koji kao argumente prima polja koja se koriste u metodu
- Metodu prosleđujemo više polja jednog objekta umesto ceo objekat

Razlozi protiv refaktorisanja

Refaktorisanje se zasniva na pravljenju niza promena. Promene nisu uvek dobre. Da bi refaktorisanje bilo delotvorno treba da se vrše promene sa namenom.

Ne treba refaktorisati ako:

- Postoje višestruki, međusobno zavisni problemi
- Ne mogu se primeniti mali koraci u refaktorisanju bez remećenja postojećeg ponašanja koda
- Postojeći kod ne radi
- Krajnji rokovi su suviše blizu
- Važi više navedenih uslova

4. Specifična refaktorisanja

4.1 Refaktorisanje na nivou podataka

1) Zamena magičnog broja imenovanom konstantom – umesto da svuda pišemo 3.14, treba deklarisati konstrantu PI
2) Davanje jasnih i informativnih imena promenljivama – isto važi i za konstante, klase, metode
3) Umetanje izraza – umesto da nekoj promenljivoj dodelimo vrednost izraza i koristimo je kao posrednika, bolje je da odmah napišemo izraz tamo gde nam je potreban
4) Zamenjivanje izraza metodom – obično da se izraz ne bi ponavljao u kodu
5) Korišćenje promenljive kao posrednika – promenljivoj dodeljujemo vrednost izraza, a njeno ime treba da odgovara nameni vrednosti

6) Zamena promenljive koja se koristi više puta promenljivama koje se koriste jednom – svaka treba da ima odgovarajuće ime
7) Za lokalne potrebe treba koristiti lokalnu promenljivu pre nego argument – svaki argument koji je tu samo da bi se koristio kao lokalna promenljiva treba izbaciti
8) Konvertovanje primitivnog tipa u klasu – ako za podatke određenog tipa treba da postoji određeno ponašanje, treba napraviti klasu koja implementira to ponašanje
9) Konvertovanje grupe kodova u klasu – umesto navođenja samostalne grupe kodova, bolje je smestiti ih u klasu, gde se može omogućiti bolja provera tipova
10) Konvertovanje grupe kodova u klasu sa podklasama – ako različiti elementi povezani sa različitim tipovima koda mogu imati različito ponašanje, treba napraviti klasu za tip i napraviti podklase za tipove koda (Screen, File, Printer, OutputType)
11) Zamena niza objektom – bolje je koristiti objekat nego niz čiji su elementi različitih tipova. U objektu će svaki element niza biti jedno polje.

12) Enkapsulacija kolekcije – ako klasa vraća kolekciju, više kolekcija van klase može napraviti problem sinhronizacije. Treba modifikovati klasu da vraća read-only kolekciju, a da se ta kolekcija može menjati odgovarajućim metodima
13) Zameniti tradicionalni zapis klasom – klasa treba da sadrži sve elemente zapisa i da omogući definisanje obrade greške, istrajnost i druge operacije.

4.2 Refaktorisanje na nivou iskaza

1) Razlaganje logičkog izraza – pojednostavljivanje logičkog izraza uvođenjem dobro imenovanih posrednih promenljivih
2) Izdvajanje logičkog izraza u funkciju – ako je izraz komplikovan, njegovo izdvajanje povećava čitljivost koda. Ako se izraz nalazi na više mesta u programu, ovako se gubi potreba za paralelnim modifikacijama i smanjuje mogućnost greške.
3) Objedinjavanje delova koji se ponavljaju na više mesta u uslovu – ako na kraju if bloka imamo isti kod koji se nalazi i na kraju else bloka, treba ga izvući van if-then-else naredbe.
4) Break ili return umesto promenljive za kontrolu petlje – promenljive (Done) koje kontrolišu petlju treba eliminisati
5) Return umesto logike – treba izaći iz metoda čim se izračuna tražena vrednost, bez stavljanja return naredbe u if-then-else
6) Zamena uslova nasleđivanjem (naročito ponovljenih switch naredbi) – umesto da na više mesta u programu menjamo case delove switch naredbe, treba napraviti klasu sa podklasama
7) Kreiranje i korišćenje null objekata umesto testiranja da li je null – umesto testiranja, obraditi null slučajeve

4.3 Refaktorisanje na nivou metoda

1) Raspakivanje metoda – ukoliko u metodu postoji neki složen deo koda koji se nalazi na više mesta, treba ga izdvojiti u poseban metod
2) Premeštanje koda inline metoda – ako postoji jednostavan metod, treba ga ukloniti i njegov kod premestiti na ona mesta na kojima se koristi
3) Konvertovanje velikog metoda u klasu – povećava čitljivost
4) Zamena komplikovanog algoritma jednostavnijim
5) Dodavanje argumenta metodu – ako je potrebno više informacija od pozivaoca
6) Uklanjanje argumenta – ako metod više ne koristi taj argument
7) Razdvajanje upitnih operacija i operacija koje menjaju stanje objekta – ako jedan metod obavlja obe stvari, treba ga rastaviti na dva (getTotals())
8) Spajanje sličnih metoda – ako se dva metoda razlikuju samo u obradi jedne promenljive, a obavljaju isti posao, treba ih spojiti

9) Razdvajanje metoda čije ponašanje zavisi od vrednosti argumenata – ako metod izvršava različit deo koda, zavisno od vrednosti argumenta, treba ga podeliti na dva metoda
10) Prosleđivanje celog objekta pre nego nekih polja
11) Prosleđivanje određenih polja umesto celog objekta – ako pravimo objekat samo da bismo ga prosledili
12) Enkapsulacija roditeljskih odnosa - ako metod vraca objekat, treba da vrati što specifičniji tip, naročito ako su u pitanju iteratori, kolekcije, elementi kolekcije...

4.4 Refaktorisanje implementacije klase

1) Promeniti vrednosne objekte u referentne – ako se koristi mnogo kopija velikih i složenih objekata, treba promeniti pristup i dozvoliti samo jednu, glavnu kopiju, a sa ostalih mesta omogućiti pristup preko reference
2) Promeniti referentne objekte u vrednosne – ako je u pitanju mali ili jednostavan objekat, bolje je koristiti kopije nego pristup preko reference
3) Zameniti virtuelne metode inicijalizacijom podataka – ako postoji više metoda koji se razlikuju samo po konstantnoj vrednosti koju vraćaju, zavisno od toga koja podklasa ih poziva, treba sve metode spojiti u jedan u roditeljskoj klasi, a konstantnu vrednost koju vraća inicijalizovati pomoću izvedene klase

4) Promena metoda ili zamena podataka – prilikom izvođenja klasa postoje saveti koje treba razmotriti kako ne bi došlo do ponavljanja u izvedenoj klasi: smeštanje metoda, polja ili tela konstruktora u nadklasu. Takođe, promene koje podržavaju izvođenje klasa su: smeštanje metoda, polja ili tela konstruktora u podklasu.
5) Smeštanje specijalizovanog koda u podklasu – ako klasa sadrži kod koji koristi samo deo njenih instanci, treba ga smestiti u posebnu podklasu.
6) Spajanje sličnih kodova u nadklasi – ako dve podklase imaju sličan deo koda, treba ga izdvojiti i smestiti u nadklasu.

4.5 Refaktorisanje interfejsa klase

1) Premeštanje metoda u drugu klasu – pravi se nov metod u konkretnoj klasi, a stari se premešta u novu klasu i poziva se iz novog metoda. Metod se može koristiti i direktno preko nove klase
2) Razlaganje klase na dve nove – ako klasa ima 2 ili više oblasti odgovornosti, poželjno je podeliti je na manje klase td. svaka od njih ima samo jednu odgovornost
3) Brisanje klase – ako klasa ne obavlja mnogo poslova, njen kod se može dodati drugoj klasi, a ona se briše
4) Sakrivanje delegata – nekada klasa A poziva klasu B i klasu C, a zapravo treba da klasa A poziva B, a klasa B da poziva C. U ovakvim slučajevima treba proveriti redosled pozivanja klasa.
5) Zameniti nasleđivanje delegacijom – ako klasa želi da koristi drugu klasu, ali hoće da ima više kontrole nad nejnim interfejsom
6) Zameniti delegaciju nasleđivanjem – kada klasa koristi svaki javni metod druge klase

7) Uklanjanje posrednika – ako klasa A zove klasu B, a klasa B zove klasu C, nekada je bolje da klasa A direktno zove klasu C.
8) Uvođenje stranog metoda – ako je klasi potreban nov metod koji ne može da obezbedi, možemo taj metod kreirati u nekoj klijent klasi
9) Uvođenje klase sa dodacima – ako je klasi potrebno nekoliko metoda čiju funkcionalnost ne može da obezbedi, možemo napraviti novu klasu koja omogućuje implementaciju tih metoda. To se može omogućiti pravljenjem ove klase kao podklase, ili upakivanjem nove klase i izlaganjem potrebnih metoda
10) Enkapsulacija izloženih članova promenljive – promena modifikatora promenljive sa public na private i slanje vrednosti promenljive preko metoda
11) Odstranjivanje set() metoda za polja koja se ne mogu menjati – ako se vrednost polja može zadati samo pri kreiranju
12) Sakrivanje metoda koji ne treba da se koriste van klase
13) Enkapsulacija nekorišćenih metoda – ako se koristi samo deo metoda neke klase, one koji se ne koriste treba sakriti
14) Spajanje nadklase i podklase ako su im implementacije veoma slične – u podklasi nema mnogo specijalizacije

4.6 Refaktorisanje na nivou sistema

1) Zamena jednosmernog odnosa među klasama dvosmernim – ako postoje dve klase koje treba da koriste međusobne elemente, ali samo jedna klasa zna za postojanje druge klase, treba ih promeniti tako da svaka zna za onu drugu.
2) Zamena dvosmernog odnosa među klasama u jednosmerni – kada samo jedna klasa koristi delove druge.
3) Factory metod pre običnog konstruktora – koristi se kada treba kreirati objekte na osnovu tipa koda ili pri radu sa referencama objekata.
4) Zamena koda za grešku izuzetkom i obratno – zavisi od strategije za upravljanje greškama.

5. Sigurno refaktorisanje

1) Sačuvati početni kod - pre nego što se započne sa refaktorisanjem, treba sačuvati kod.
2) Neka promene budu male – refaktorisanja treba da budu mala i da bude jasno koje promene zašto nastaju i na šta utiču.
3) Jedna po jedna promena – neke promene su komplikovanije od ostalih. Promene treba izvršavati jednu za drugom. Nakon jedne promene treba ponovo kompajlirati program, pokrenuti testove, pa onda ići dalje.
4) Napraviti spisak koraka koje treba preuzeti
5) Napraviti parking mesto – ako se prilikom refaktorisanja uoči da je potrebno izmeniti još nešto, a zatim se ponovo primeti da treba izmeniti još nešto... Za promene koje nisu odmah potrebne pravi se “parking mesto” - lista promena koje treba napraviti u nekom trenutku.
6) Pravljenje čestih kontrolnih tačaka – lako je pokvariti kod refaktorisanjem. Pre nego što se sa refaktorisanjem počne, treba napraviti niz kontrolnih tačaka.

7) Korišćenje upozorenja koja vraća kompajler – treba postaviti podešavanja kompajlera na najosetljivija, tako da i za najmanju sitnicu izbaci upozorenje ili grešku.
8) Ponovno testiranje – nakon refaktorisanja kod treba testirati
9) Dodavanje test slučajeva – za nove delove koda treba dodati nove test slučajeve
10) Provera izmena – kada programer pravi promene u programu, obično je više od 50% šanse da će prvi put napraviti grešku. Međutim, ako programer menja veći deo koda, mogućnost greške se smanjuje. Mogućnost greške raste dok je broj promena 1 do 5 linija koda. Nakon toga, mogućnost greške opada.
11) Pristup zavisi od nivoa rizika – neka refaktorisanja su rizičnija od drugih. Kod refaktorisanja niskog rizika, možemo raditi i više odjednom, ali se refaktorisanjima visokog rizika mora oprezno pristupiti. Refaktorisanje niskog rizika: zamena magičnog broja. Refaktorisanja visokog rizika: promena interfejsa klase, metoda.

Kada ne refaktorisati

- Ne treba refaktorisati kod koji ne radi kako bi bio popravljen. To nije refaktorisanje jer se ono odnosi na promene koda koji radi koje ne utiču na ponašanje tog koda.
- Treba izbegavati refaktorisanje umesto ponovnog pisanja koda. Nekada kod ne zahteva male promene, već brisanje i ponovno pisanje.

6. Strategije u refaktorisanju

Pravilo 80/20: Nakon izvođenja 20% od ukupnog planiranog refaktorisanja, dobit je 80%.

1) Refaktorisanje nakon dodavanja metoda
2) Refaktorisanje nakon dodavanja klase
3) Refaktorisanje nakon popravljanja bagova
4) Refaktorisanje modula sklonih greškama
5) Refaktorisanje najkompleksnijih modula
6) Refaktorisanje tokom rada – ne treba refaktorisati kod koji se ne menja, ali ako se neki kod menja, nakon promena ga treba ostaviti u boljem stanju nego pre
7) Definisanje granice između čistog i ružnog koda i prevođenje koda preko te granice

Hvala na pažnji!