POGLAVLJE 1

Refaktorisanje

Neko me jednom pitao šta mi se toliko svidelo u refaktorisanju. Zbog čega sam se tako često vraćala takvim projektima u svom poslu? Rekala sam mu da postoji neka zavisnost u vezi toga. Možda je to jednostavno čin sređivanja, poput urednog grupisanja i ređanja začina ili je možda to radost zbog raščišćavanja nečega, poput odnošenja torbe iznošene odeće u Crveni krst. Ili je možda u mojoj glavi tanani glas koji me podseća da će ove male postepene promene biti značajno poboljšanje u svakodnevnom životu mojih kolega. Mislim da je u pitanju kombinacija svega navedenog.

U procesu refaktorisanja postoji nešto što može da se svakom svidi, bilo da pravimo nove mogućnosti proizvoda ili radimo na skaliranju infrastrukture. Svi moramo postići ravnotežu u svom radu između pisanja mnogo i pisanja malo koda. Moramo nastojati da shvatimo posledice promena koje smo napravili, bilo da su namerne ili ne. Kôd je živa stvar koja diše. Kad razmišljam o kodu koji sam napisala da traje narednih pet, deset godina, ne mogu a da se ne trgnem. I nadam se da će do tada naići neko da ga u potpunosti ukloni ili zameni nečim čišćim i, što je najvažnije, prilagođenijim potrebama aplikacije za to buduće vreme. Eto šta je refaktorisanje.

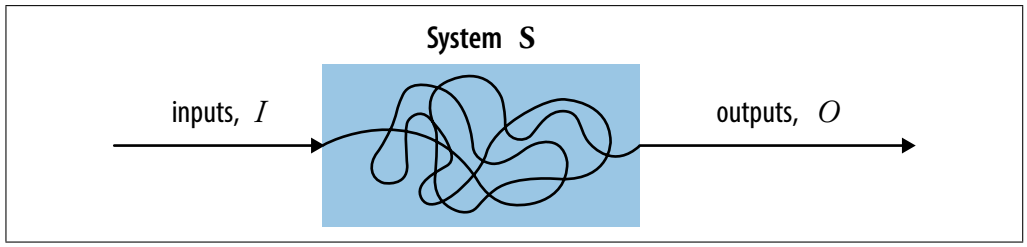
U ovom poglavlju započinjemo definisanjem nekoliko koncepata. Predložićemo osnovnu definiciju refaktorisanja za opšti slučaj i nadogradićemo se na nju kako bismo razvili posebnu definiciju za refaktorisanje po meri. Da bismo naglasili neke od motivacija ove knjige, diskutovaćemo o tome zašto bi trebalo da brinemo o refaktorisanju i koje prednosti možemo doneti našim timovima ako smo usavršili ovu veštinu. Dalje ćemo zaroniti u neke od pogodnosti koje možemo očekivati od refaktorisanja i u neke od rizika koje bismo trebali imati na umu kada razmišljamo da li da to učinimo. Uz naše znanje o kompromisima, razmotrićemo neke scenarije kada je pravo vreme, a kada pogrešno. Na kraju ćemo proći kroz kratki primer kako bismo oživeli ove koncepte.

Šta je refaktorisanje?

Jednostavno rečeno, *refaktorisanje* je proces kojim restrukturiramo postojeći kod (*factoring*) bez promene njegovog spoljnog ponašanja. Ako mislite da je ova definicija vrlo opšta, ne brinite; namerno je takava! Refaktorisanje može imati mnogo isto efikasnih oblika,

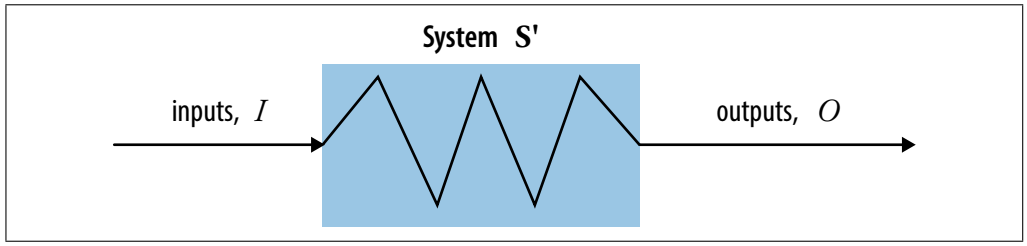
u zavisnosti od koda na koji se primenjuje. Da bismo to ilustrovali, definisaćemo „sistem“ kao bilo koji definisani skup koda koji proizvodi skup izlaza na osnovu skupa ulaza.

Recimo da imamo konkretnu primenu takvog sistema nazvanog S, prikazano na slici 1-1. Sistem je izgrađen pod pritiskom kratkog roka, podstičući autore da ne urade sve kako treba. Vremenom je to postala velika gomila zamršenog koda. Srećom, korisnici sistema nisu direktno izloženi unutrašnjem neredu sistema; oni komuniciraju sa S, koristeći definisani interfejs i oslanjaju se na njega da bi obezbedili dosledne rezultate.



Slika 1-1. Jednostavan sistem sa ulazima i izlazima

Nekoliko hrabrih programera je očistilo unutrašnjost sistema, koji ćemo sada nazvati S', prikazan na slici 1-2. Iako je to možda uredniji sistem, za korisnike S', apsolutno se ništa nije promenilo.

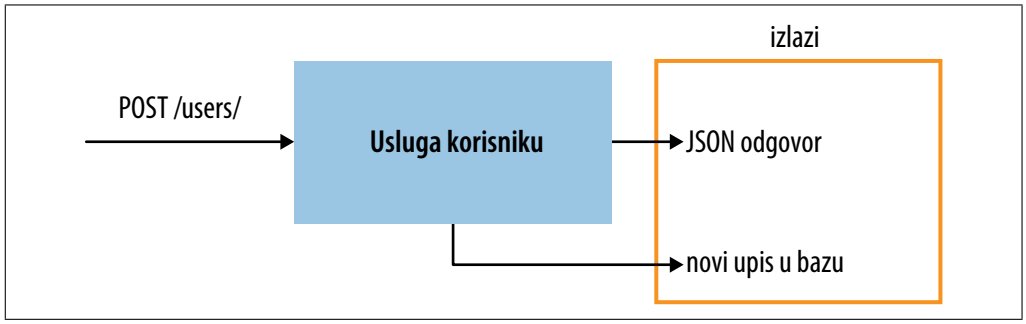


Slika 1-2. Jednostavan prepravljen sistem sa ulazima i izlazima

Sistem S može biti bilo šta; to može biti pojedinačna naredba `if`, funkcija od deset redova, popularna biblioteka otvorenog koda, višemilionska aplikacija ili bilo šta između. (Ulazi i izlazi mogu biti vrlo raznoliki.) Sistem može da radi sa unosima baze podataka, kolekcijama datoteka ili tokovima podataka. Izlazi nisu ograničeni na vraćene vrednosti, ali mogu takođe da uključuju brojna neželjena dejstva, kao što je ispis na konzoli ili zadavanje mrežnih zahteva. Na slici 1-3 možete videti kako se usluga RESTful odgovorna za rad na korisničkim entitetima može preslikati na našu definiciju sistema.

Kako nastavljamo da nadograđujemo našu definiciji refaktorisanja i započinjemo s istraživanjem različitih aspekata procesa, najbolji način da smo svi na istom koloseku je povezivanje svake ideje sa konkretnim primerom.

Korišćenje primera programiranja iz prakse je teško iz nekoliko razloga. S obzirom na širinu iskustava u našoj industriji, odabir samo jednog primera nad drugim odmah jednu grupu čitalaca izdvaja u odnosu na druge. Sa druge strane, oni koji su dobro upoznati sa primerom mogu biti frustrirani kada se pojedini koncepti pojednostave zbog kratkoće izlaganja ili kada se zanemaruju određene nijanse da bi se koncept prikazao čistije. U nadi da ćemo uspostaviti jednake uslove, kad god želimo da na visokom nivou ilustrujemo opšti problem, za primer ćemo koristiti posao poznat (nadamo se) većini od nas: postrojenje za hemijsko čišćenje.



Slika 1-3. Jednostavna aplikacija kao sistem

Simon's Dry Cleaners je lokalno preduzeće za hemijsko čišćenje sa lokacijom u prometnoj ulici u Springfieldu. Otvoreno je od ponedeljka do subote tokom redovnog radnog vremena. Mušterije ostavljaju i stvari za pranje i stvari samo za hemijsko čišćenje. U zavisnosti od količine, hitnosti i težine svake stvari, stvari se čiste i vraćaju kupcima u bilo koje vreme između dva i šest radnih dana kasnije.

Kako se ovo odnosi na našu definiciju sistema? Hemijsko čišćenje smešteno u okviru preduzeća je sam sistem. Prljavu odeću kupaca obrađuje kao ulaz, a očišćenu vraća vlasnicima kao izlaz. Sve složenosti postupka hemijskog čišćenja skrivene su od potrošača; sve što treba da uradimo je da predamo odeću i da se nadamo da će oni koji treba, uraditi svoj posao. Sam sistem je prilično složen; u zavisnosti od vrste ulaza (kožna jakna, gomila čarapa, svilena suknja), može odgovoriti izvođenjem jedne ili više operacija kako bi osigurala ispravan izlaz (čista odeća). Postoji dosta prilika da nešto pođe po zlu između predaje i preuzimanja: kaiš se može izgubiti, mrlja prevideti, košulja se slučajno vratila pogrešnoj mušteriji. Međutim, ako zaposleni proaktivno komuniciraju jedni s drugima, a mašine su u dobrom stanju i radni nalozi se popunjavaju kako treba, sistem će nastaviti da radi bez zastoja i biće lako odraditi narudžbine.

Recimo da Simon i dalje vodi svoje poslovanje koristeći papirne radne naloge. Svi kupci koji bi došli da ostave odeću napisali bi svoje ime i broj telefona na priloženom listiću, a službenica bi zabeležila njihovu porudžbinu. Ako bi kupci izgubili svoju priznanicu, Simon bi lako mogao da pronađe kopiju listajući nedavne porudžbine po abecedi prezimena. Nažalost, kada kupci zakasne po očišćeno i ako su još i izgubili priznanicu,

službenik mora da uzme listiće iz arhive. Iako se gotovo sve narudžbe uspešno preuzimaju, u ovom slučaju potrebno je mnogo više vremena da bi kupac preuzeo odeću i otišao svojim putem. Papirni nalozi su takođe nezgodni kada vlasnik izračunava zaradu na kraju svakog meseca; mora ručno da uskladi sve transakcije (i kreditne kartice i gotovinu) sa urađenim narudžbama. Nestrpljiv da modernizuje i refaktoriše proces, tim je odlučio da nadogradi sistem tako da izbace papirne naloge. Ta da, refaktorisanje završeno! Kupci nastavljaju da ostavljaju stvari za hemijsko čišćenje i preuzimaju ih nekoliko dana kasnije uz minimalne uočene promene, ali sada sve iza prijemnog odeljenja teče mnogo glađe.

Šta je refaktorisanje po meri?

Krajem 2013. godine, usred burnog pokretanja vladine veb lokacije za zdravlje Healthcare.gov, svi glavni američki mediji proglasili su je potpunim fijaskom. Veb lokaciju su mučili sigurnosni problemi, višečasovni prekidi rada i mnoštvo ozbiljnih grešaka. Pre pokretanja, ne samo da su se troškovi povećali na skoro dve milijarde dolara, baza koda je eksplodirala na preko pet miliona redova koda. Iako je veliki deo neuspeha Healthcare.gov nastao usled neuspelih razvojnih praksi zaglavljenih birokratskim politikama savezne vlade. Pošto je Obamina administracija najavila da planira da uloži velika sredstva u poboljšanje usluge, neporecive poteškoće povezane sa izmenom arhitekture i refaktorisanjem nadvile su se nad sistemom i postale su glavne vesti u medijima. U narednim mesecima, timovi koji su imali zadatak da prerade Healthcare.gov zaronili su u gotovo kompletnu reviziju baze koda, refaktorisanje po meri.

Refaktorisanje srazmerno potrebnoj meri je ono koje utiče na značajnu oblast vašeg sistema. Tipično (ali ne isključivo) uključuje veliku bazu koda (milion ili više redova koda) koju koriste aplikacije sa mnogo korisnika. Sve dok postoje stari sistemi, postojaće potreba za ovakvim refaktorisanjem, onim kod kojih programeri moraju kritički razmišljati o širokoj strukturi koda i o tome kako ga je moguće efikasno poboljšati. Po čemu se refaktorisanje multimilionskih baza koda razlikuje od refaktorisanja manjih, preciznije definisanih aplikacija? Iako bi nam moglo biti lako da vidimo konkretne, iterativne načine za poboljšanje malih, dobro definisanih sistema (mislimo o pojedinačnim funkcijama ili klasama), postaje gotovo nemoguće utvrditi uticaj koji bi promena mogla imati kada se ujednačeno primenjuje na široki, složeni sistem. Postoje mnogi alati za identifikovanje mirisa koda ili automatsko otkrivanje mogućih poboljšanja unutar delova koda, ali mi u velikoj meri nismo u stanju da automatizujemo ljudsko rezonovanje o tome kako da restrukturiramo velike aplikacije u bazama kodova koje rastu sve bržim tempom, posebno u kompanijama sa velikim rastom.

Neko može da tvrdi da možete napraviti izmerljivo poboljšanje ove vrste sistema kontinuiranom primenom malih, korak po korak transformacija. Ova metoda bi mogla početi da naginje vagu u pozitivnom smeru, ali napredak će verovatno znatno pasti kad

nestane većina voća na niskim granama i kada postane neprijatno uvoditi ove promene pažljivo (i postepeno).

Refaktorisanje po meri odnosi se na identifikovanje sistemskog problema u vašoj bazi koda, osmišljavanje boljeg rešenja i sprovođenje tog rešenja na strateški, disciplinovan način. Da biste identifikovali sistemske probleme i njihova odgovarajuća rešenja, biće vam potrebno čvrsto razumevanje jednog ili više većih delova aplikacije. Takođe će vam trebati velika izdržljivost da biste pravilno primenili rešenje na celo zahvaćeno područje.

Refaktorisanje srazmerno potrebnoj meri takođe ide paralelno sa refaktorisanjem sistema u punom pogonu. Mnogi od nas rade na aplikacijama sa čestim ciklusima primene. U Slacku našim korisnicima šaljemo novi kôd desetak puta dnevno. Moramo imati na umu kako se naši napori za refaktorisanje uklapaju u ove cikluse, kako bismo minimizovali rizik i ometanje naših korisnika. Razumevanje strateškog raspoređivanja na različitim tačkama tokom napora pri refaktorisanju često čini razliku između neprimetne popravke sistema i potpunog prekida usluge.

Kako bi mogle da izgledaju Simon's Dry Cleaners uzimajući u obzir refaktorisanje po meri? Recimo da je postavljanje sistema prodajnih mesta (point-of-sale) dramatično optimizovalo posao – zapravo toliko dobro da je Simon otvorio pet novih lokacija u susednim gradovima za samo dve godine! Sada kada posluje na više lokacija, povećavajući obim svog poslovanja, imaju drugačiji skup problema. Da bi troškovi bili niski, samo dve od šest lokacija imaju opremu za hemijsko čišćenje na licu mesta. Kada kupci ostave odeću za hemijsko čišćenje na nekoj od četiri lokacije na kojima nema opreme za hemijsko čišćenje, odeća se mora poslati u najbliži objekat putem kombija kompanije. Kombi se zaustavlja na sva četiri lokacije da pokupi odeću, odlažući je u velike korpe na dve lokacije koje imaju hemijsko čišćenje. Simonovi zaposleni vredno rade na sortiranju gomile odeće, čišćenju i vraćanju u odgovarajuću radnju. Međutim, većinu dana to je mučan proces. Obe lokacije za hemijsko čišćenje obrađuju odeću i iz svoje radnje i iz četiri udaljene. Nisu retki slučajevi da se odeća razdvoji ili zaplete kada ih vozači kombija spuste u odgovarajuće korpe. Hitnije porudžbine često se gube u gomili, a radnici moraju da kopaju po celoj pošiljci da prvo njih izdvoje.

Kako Simon može najefikasnije poboljšati svoje poslovanje? Da li bi trebalo da dodeli određenu lokaciju sa hemijskim čišćenjem svakoj lokaciji koja ga nema, tako da svaki pogon čišćenja obrađuje narudžbine sa najviše tri lokacije? Ako je odgovor da, da li bi trebalo razmisliti o preusmeravanju kombija na određeni način? Šta ako je učinilo i jedno i drugo? Da li bi bilo isplativo otvoriti još jedno postrojenje za hemijsko čišćenje ako bi to omogućilo poslovanju da skrati vreme obrade odeće? Kako da postavi korpe utovara tako da se manje odeće zaplete? Da li bi se vozači mogli naučiti da raspoređuju poslove po prioritetu pre nego što krenu u novu turu? Da li bi kompanija trebalo da ograniči primanja odeće od odmah posle vremena za ručak i nedugo nakon zatvaranja kako bi lokacije za hemijsko čišćenje imale više vremena za organizovanje preuzimanja? Postoji nekoliko opcija koje treba razmotriti, od kojih bi se mnoge mogle kombinovati i

izvršavati po brojnim nalozima ili istovremeno. Zamislite da ste suočeni sa svim ovim mogućnostima i da morate da odlučite koju polugu ćete prvo povući. To je pozitivno parališuće! Ispostavlja se da je isti osećaj refaktorisanje velikih aplikacija.

Zašto bi trebalo da vam je važno refaktorisanje?

Refaktorisanje u teoriji može zvučati primamljivo, ali kako znate da čitanje ove knjige neće biti gubljenje vremena? Naravno da se nadam da će svi čitaoci koristeći ovu knjigu doći do nekoliko novih alatki za svoj skup alata, ali ako postoji jedan razlog koji bih mogao da navedem da biste nastavili da čitate, to je ovaj:

Poverenje u sopstvenu sposobnost da refaktorišete omogućava vam da krenete u akciju i započnete izgradnju sistema pre, mnogo pre nego što ste razvili snažno razumevanje svih pokretnih delova, zamki i graničnih slučajeva. Ako znate da ćete uspeti da identifikujete načine za efikasno poboljšanje komponenata tokom čitavog razvojnog procesa, a to ćete i dalje biti u stanju kako sistem bude postajao sve složeniji, nećete trošiti toliko vremena na unapređenje arhitekture programa. Jednom kada usavršite veštine potrebne za manipulisanje kodom bez napora, potrošićete manje vremena brinući o svakoj odluci o dizajnu. Tokom programiranja, doći ćete u situacije da ćete pisati nešto jednostavno što dobro funkcioniše za trenutne okolnosti, umesto da idete korak unazad i planirate sledećih pola tuceta poteza. Prepoznavaćate da uvek postoji (ponekad sa začkoljicom) put do boljeg rešenja.

Programiranje nije igra šaha. Kada dobiju konfiguraciju ploče i optimalne protivnike, najbolji takmičari spretno odigraju na desetine kompletnih mečeva u roku od nekoliko minuta. Nažalost, u našem poslu nismo dobili potpuno nabrojan skup mogućih poteza i ne postoji unapred određeno stanje. Ne želim da impliciram da nema smisla sedeti i mozgati robusno rešenje problema, za dati razuman skup zahteva; međutim, želim da vas upozorim da ne trošite značajno vreme na peglanje poslednjih 10 do 20 procenata. Ako ste usavršili sposobnost refaktorisanja, moći ćete da razvijete svoje rešenje kako biste sasvim dobro izašli u susret konačnim specifikacijama.

Prednosti refaktorisanja

Refaktorisanje može imati neke opipljive koristi osim mogućnosti da se ranije započne sa samouverenim rešavanjem problema. Iako možda nije pravi alat za svaki problem, sigurno može imati trajni, pozitivni uticaj na vašu aplikaciju, inženjerski tim i širu organizaciju. Razmotrimo dve glavne prednosti: povećana produktivnost programera i veća lakoća u identifikovanju grešaka. Iako bi neki mogli tvrditi da od refaktorisanja ima mnogo više koristi od onih koje su ovde navedene, ja tvrdim da se sve one svode na dve ovde predstavljene.

Produktivnost programera

Jedan od primarnih ciljeva refaktorisanja je proizvodnja koda koji je lakše razumljiv. Pojednostavljivanje zapetljanog rešenja dok ga promišljate ne samo da vam pomaže da bolje shvatite šta kôd radi, već pomaže i svima koji dolaze posle vas da rade isti posao. Kôd koji lako možete shvatiti podiže apsolutno sve u vašem timu, bez obzira na njihovo zvanje ili nivo iskustva.

Ako ste inženjer u timu, uglavnom ste dobro upoznati sa nekim delovima baze koda, ali kako baza koda raste, sve više i više delova vam je nepoznato i sve je veća verovatnoća da će vaš kôd razviti zavisnost od tih delova. Zamislite da implementirate novu funkciju i provlačeći svoje rešenje kroz sistem, prelazite sa koda koji prilično dobro poznajete na nepoznatu teritoriju. Ako se vama nepoznato područje dobro održava i redovno refaktoriše kako bi se uzeli u obzir promenljivi zahtevi proizvoda i ispravile greške, moći ćete da suzite idealno mesto za vašu promenu *i* primenite intuitivno rešenje bez napora. Ako se umesto toga kôd vremenom pogoršavao skupljanjem loše ispravljenih grešaka i rastom u dužinu, potrošićete eksponencijalno više vremena provlačeći se kroz svaki red, pokušavajući prvo da razumete šta kôd radi i kako to radi pre nego što pređete na razmišljanje o prihvatljivom rešenju. (Nije retko uvući i nekog drugog u zečju rupu sa mučnim kodom, da bi odgovorio na vaša pitanja, bilo da je to još jedan inženjer koji radi uz vas ili onaj koji je blisko upoznat s kodom.)

Evolucija familijarnosti sa bazom koda

Za manje baze kodova sa samo nekoliko inženjera, nije retko da većini inženjera bude izuzetno ugodno u bilo kom delu baze kodova. Familijarnosti će se postepeno smanjivati s vremenom kako se dodaje i modifikuje sve više modula, a inženjeri počinju da se specijalizuju; na kraju, baza koda dostiže kritičnu masu gde je nemoguće da bilo koji pojedini inženjer (čak i prvo anagažovani!) bude upoznat sa celim kodom.

Preokrenimo scenario. Šta ako bi kolega iz drugog tima koji nije upoznat sa kodom vašeg tima morao da ga razume dok ga čita. Da li bi lako razumeo kako funkcioniše? Ili očekujete pitanja i njegov zbunjeni pogled ili zahtev za pregled koda?

Šta da ste vi novi inženjer u timu. Možda ste to bili nedavno ili ste možda nedavno uključili nekoga u svoj tim. Oni apsolutno nemaju u svojoj glavi model koda. Njihova sposobnost sticanja poverenja u bilo koji deo koda direktno je proporcionalna čitljivosti koda. Ne samo da će moći prirodno da izgrade tačan mentalni prikaz veza između različitih delova u vašoj bazi koda, oni će moći da obrazlože šta kôd radi, bez potrebe da upitukuju kolege u timu. (Vredi napomenuti da je znati kada i kako postavljati pitanja svojim kolegama neverovatno važna veština koju treba usavršiti. Naučiti proceniti koliko vam vremena treba da biste izgradili sopstveno razumevanje pre nego što zatražite pomoć je teško, ali je presudno za razvoj programera. Postavljanje pitanja nije loše, ali ako ste inžinjer u timu i osećate se bombardovano njima, možda je vreme da napišete neku dokumentaciju i prepravite neki kôd.)

Svi smo skloni kopiranju ustaljenih obrazaca kada razvijamo nešto novo. Ako su rešenja na koja se pozivamo jasna i sažeta, veća je verovatnoća da ćemo primeniti jasan i sažet kôd. Tačno je i obratno: ako su jedina rešenja koja imamo kao referencu zapetljana, proširićemo nered. Osiguranje da najbolji obrasci budu najzastupljeniji je posebno presudno za uspostavljanje pozitivne povratne sprege kod programera koji tek počinju. Ako je kôd sa kojim redovno komuniciraju lako razumljiv, slično će se fokusirati u sopstvenim rešenjima.

Identifikovanje grešaka

Traženje i rešavanje grešaka je neophodan (i zabavan!) deo naših poslova. Refaktorisanje može biti efikasno sredstvo za izvršavanje oba ova zadatka! Razbijanjem složenih naredbi na manje delove veličine jednog zalogaja i izdvajanjem logike u nove funkcije, možete stvoriti bolje razumevanje onoga šta kôd radi a time i izolovanju greške. Refaktorisanje dok aktivno pišete kôd takođe može olakšati uočavanje grešaka u ranom procesu razvoja, omogućavajući vam da ih potpuno izbegnete.

Razmotrite scenario u kome je vaš tim pre nekoliko sati primenio novi kôd u proizvodnji. Nekoliko promena je ugrađeno u pregršt datoteka od kojih se svi plaše da ih izmene: kôd je nemoguće pročitati i sadrži minsko polje grešaka koje čekaju da se dogode. Nažalost, vaši testovi nisu pokrili jedan od mnogih graničnih slučajeva, a neko iz korisničke službe vam javlja o dosadnoj grešci na koju korisnici počinju da nailaze. Vi i vaš tim odmah počinjete da kopate i brzo shvatate da je greška, kako se i očekivalo, u najstrašnijem delu koda. Srećom, vaš saradnici mogu dosledno reprodukovati problem i zajednički pišete test kako biste potvrdili ispravno ponašanje. Sada morate suziti grešku. Poduzimate metodičke korake za razbijanje zamršenog koda: pretvarate dugačke jednorede linije koda u sažete, višeredne izjave i migrirate sadržaj nekoliko uslovnih blokova koda u pojedinačne funkcije. Na kraju locirate grešku. Sada kada je kôd pojednostavljen, moći ćete da ga brzo popravite, pokrećete test da biste proverili da li radi i isporučite ispravku svojim kupcima. Pobeda!

Kupcu su ponekad greške samo manja smetnja, ali ponekad greške mogu sprečiti kupca da u potpunosti koristi vašu aplikaciju. Iako ometajuće greške uglavnom zahtevaju hitnu ispravku, neophodno je da vaš tim bude u stanju da brzo reši greške svih nivoa ozbiljnosti kako bi korisnici bili zadovoljni. Rad u dobro održavanoj bazi koda može dramatično smanjiti vreme koje programerima treba da se fokusiraju i isprave grešku, obradujući vas rekordnim rokom isporuke u proizvodnju.

Rizici refaktorisanja

Iako bi blagodati refaktorisanja mogle biti ubedljive, postoje neki ozbiljni rizici i zamke koje treba razmotriti pre nego što krenete na putovanje kako biste poboljšali svaki centimetar svoje baze koda. Možda počinjem da zvučim kao pokvarena ploča, ali svejedno ću ponoviti: refaktorisanje zahteva da budemo u stanju da obezbedimo da ponašanje

ostane identično u svakoj iteraciji. Možemo da budemo sigurniji da se ništa nije promenilo pisanjem paketa testova (jediničnih, integracionih, s kraj na kraj) i ne bismo trebali ozbiljno razmišljati o napornom napredovanju sa refaktorisanjem dok ne uspostavimo dovoljno pokrivenost testovima. Međutim, čak i uz temeljno testirranje, uvek postoji određena šansa da nešto promakne kroz pukotine. Takođe moramo imati na umu naš krajnji cilj: poboljšanje koda na način koji je jasan i vama i budućim programerima u čitanju koda.

Ozbiljne regresije

Refaktorisanje netestiranog koda je vrlo opasno i visoko obeshrabrujuće. Razvojni timovi opremljeni najtemeljitijim, sofisticiranim paketima za testiranje još uvek isporučuju i greške u proizvodima. Zašto? Svakom promenom, velikom ili malom, narušavamo ravnotežu sistema na merljiv način. Nastojimo da napravimo što manje poremećaja, ali kad god promenimo sisteme, postoji rizik da to može dovesti do nepredviđene regresije. Dok prepravljamo izuzetno zastrašujuće, zbunjujuće ćoškove naše baze koda, uvođenje ozbiljne regresije posebno je zabrinjavajuće. Ova područja baze koda često su u tekućem stanju, jer su imala dovoljno vremena da se pogoršaju. U kompanijama koje brzo rastu, ona su često i sastavni deo načina na koji vaša aplikacija radi i najmanje su testirana. Pokušaj raspetljavanja tih datoteka ili funkcija može izgledati kao da pokušavate da neozleđeno pređete preko minskog polja – moguće je, ali vrlo opasno.

Otkopavanje uspavanih grešaka

Baš kao što vam refaktorisanje može pomoći da identifikujete greške, tako može i nenamerno otkriti uspavane greške. Ovde, uspavane, neaktivne greške klasifikujem kao regresije koje su najčešće otkrivene kodom za restrukturiranje. Ponovo ćemo posetiti Simon's Dry Cleaners radi ilustracije. Posao je narastao i krenulo je naručivanje sredstava za čišćenje u većim količinama u istoj dinamci isporuke kako bi se dobila bolja ponuda. Na nesreću, nema mnogo prostora za skladištenje u zadnjem delu glavne radnje, pa Simon odlučuje da počne da slaže kutije bliže vratima. Posle nekoliko nedelja kiše, tim primećuje da su neke od kutija najbližih vratima mokre i da se raspadaju. Vlasnik primećuje da su zadnja vrata slabo zapečaćena i omogućavaju prodor vode u vlažnim danima. Simon nikada nije naišao na problem sa skladištenjem zaliha blizu vrata za utovar, jer to jednostavno tako nikada ranije nisu radili; korišćenje novog skladištenja otkrilo je kritičnu manu u infrastrukturi, koju možda nikada inače ne bi otkrili.

Pomeranje mete

Refaktorisanje može biti pomalo nalik na jedenje kolačića: prvih nekoliko zalogaja je ukusno, što olakšava da nastavite i desi se da pojedete tuce. Kada progutate poslednji zalogaj, javlja se malo žaljenja a možda i nalet mučnine. Iskusiti trenutna, izuzetno značajna poboljšanja čineći fokusirane, lokalizovane promene je neverovatno korisno! Lako se zaneti i dozvoliti da oblasti vaših promena pređe *razumne granice*. Šta mislim pod razumnim

granicama? U zavisnosti od baze koda, ovo se može odnositi na jedno funkcionalno područje ili mali, međusobno zavisni skup biblioteka. U idealnom slučaju, refaktorisani kôd ograničen je na skup promena koje drugi programer može udobno pregledati u okviru jednog skupa promena.

Kada se mapiraju veći napori za refaktorisanje, posebno onaj koji bi mogao potrajati nekoliko meseci ili više, apsolutno je neophodno držati se uskog opsega. Svi nailazimo na neočekivane hirove kada prepravljamo male oblasti (nekoliko redova koda, pojedinačne funkcije); iako možemo održivo povezati nekoliko poboljšanja kako bismo se efikasno izborili sa novim hirovima, ovaj pristup postaje opasan ako zahvatimo veliku oblast. Što je veća oblast planiranog refaktorisanja, to ćete naići na više problema koje verovatno niste predvideli. To vas ne čini lošim programerom, već vas jednostavno čini ljudima. Držeći se dobro definisanog plana, smanjujete šanse za ozbiljnu regresiju ili da naletite na neotkrivene greške i povećavate produktivnost. Održivi, metodički napori u refaktorisanju sami po sebi su teški; ali ako imamo nedefinisani cilj, refaktorisanje čini neostvarivim.

Nepotrebna složenost

Budite oprezni pri prekomernom dizajniranju na početku i budite otvoreni za izmene svog početnog plana. Primarni cilj treba da bude stvaranje koda prilagođenog čoveku, čak i po cenu vašeg originalnog dizajna. Ako je laserski fokus usredsređen na rešenje, a ne na proces, veća je šansa da će vaša aplikacija na kraju biti isforsirana i složenija nego što je bila u početku. Refaktorisanje na svim nivoima trebalo bi da bude iterativno. Preduzimajući male, namerne korake u jednom smeru i održavajući postojeće ponašanje na svakoj iteraciji, bićete u stanju da zadržite fokus na svoj krajni cilj. To je mnogo lakše učiniti kada se bavite samo količinom koda koji staje na vaš ekran, a ne tri desetine biblioteka istovremeno. Kada planiramo novi projekat, većina nas se uglavnom trudi da razvije detaljan dokument sa specifikacijama i plan izvršenja. Čak i uz velike napore na refaktorisanju, važno je dobro razumeti kako rezultujući kôd treba da izgleda po završetku.

Kada se refaktoriše

Bilo bi lako reći „kada je dobit veća od rizika“, ali to ne bi bio koristan odgovor. Da, u praksi je refaktorisanje vredan napor kada su koristi veće od rizika, ali kako pravilno dodeliti težinu svakom delu slagalice? Kako da znamo kada smo stigli do prekretnice i treba li uzeti u obzir refaktorisanje?

Prema mom iskustvu, prelomna tačka je više prelomni *opseg* i različit je za svakoga i za svaku aplikaciju. Određivanje gornjih i donjih granica za ovaj opseg je ono što refaktorisanje čini malo više subjektivnom naukom: ne postoji formula kojom bismo mogli da pružimo odlučan odgovor „da“ ili „ne“. Srećom, možemo se osloniti na neke empirijske dokaze iz iskustva drugih koji će nas voditi u donošenju sopstvenih odluka.

Mali opseg

Kada želite da refaktorišete mali, jasan deo dobro testiranog koda, trebalo bi da bude vrlo malo nečega da vas sputava. Ako niste sigurni da li je vaše prepravljeno rešenje objektivno poboljšanje prethodnog ili ako se plašite da će promena uticati na preveliku oblast, to je verovatno vredan poduhvat. Pažljivo pristupite izvršenju i uvedite promene! Kasnije u ovom poglavlju videćemo primer koji jasno spada u ovu kategoriju.

Kompleksnost koda aktivno ometa razvoj

Postoje trenuci kada moramo ući u delove naše baze koda kojih se plašimo. Svaki put kad čitamo kod, obrve nam se podignu, srce zakuca, neuroni počnu varničiti. Tada dolazi trenutak kada moramo da zagrizemo problem, zauzememo busiju, i izvršimo promenu zbog koje smo došli. Ali razvoj pod prinudom je siguran način da se nehotice izazove više problema. Kada ste toliko hiperkoncentrisani da uradite tačno ispravnu stvar, držeći u glavi mnoge dimenzije problema, rizikujete da izgubite iz vida svoj *stvarni* cilj. Kako možete adekvatno izvršiti taj cilj kada vam je um negde drugde?

Ako nas određeni deo koda nije još ugrizao, tada ćemo često rizikovati i to će se desiti. Ako nas je već ugrizo nas ili kolegu iz tima (ponekad i više puta), rizik od preuređivanja koda kako bi se sprečile buduće greške, mogao bi da nadmaši rizik ostavljanja koda u postojećem stanju. Ako niste sigurni u kom smeru se vaga naginje, porazgovarajte sa saradnicima i sakupite podatke o broju grešaka prijavljenih u poslednjih šest meseci koje možete povezati sa tim delom baze koda.

Promena u zahtevima proizvoda

Drastične promene u zahtevima proizvoda često se mogu preslikati na drastične promene u kodu. Koliko god se trudili da napišemo apstraktna, proširiva rešenja za svaki deo funkcionalnosti u našoj aplikaciji, ne možemo predvideti budućnost; i iako se naš kôd može lako prilagoditi malim promenama, retko je savršeno prilagodljiv većim. Ove promene daju nam retku poslovnu priliku da se vratimo praznom listu papira i preispitamo sopstveni dizajn.

Možda mislite da ovakve izmene nikako ne mogu sačuvati ponašanje. Sa istim ulazima, sada moramo pružiti različite izlaze! Kako to da je ovo pogodno za refaktorisanje? Ako vaš kôd u trenutnom stanju ne odgovara novim zahtevima, morate da smislite rešenje koje i dalje podržava postojeću funkcionalnost i da glatko podrži buduću. Možete prvo da uradite refaktorisanje koda, a zatim (i tek tada!) da ugradite novu funkcionalnost preko toga. Na ovaj način nastavljate da postavljate standard visokokvalitetnog koda, realizujući sve beneficije refaktorisanja, a istovremeno podržavajući poslovne ciljeve. I opet, to je uspeh, uspeh, uspeh!

Performanse

Poboljšanje performansi može biti težak zadatak; prvo morate duboko razumeti postojeće ponašanje, a zatim biti u stanju da prepoznate koje poluge treba da koristite da nagnete vagu u pozitivnom smeru. Početak od čiste pozicije (ili je napraviti kao prvi korak) to će vam najbolje omogućiti. Takođe je ključno pravilno izolovati poluge koje ste identifikovali tako da je s njima lakše manipulisati bez rizika od neželjenih efekata.

Ne veruju svi programeri da su poboljšanja performansi valjan razlog za refaktorisanje; neki tvrde da su performanse sistema sastavni deo njegovog ponašanja i da ga stoga promena performansi na neki način menja ponašanje. Ne slažem se stim. Ako i dalje definišemo refaktorisanje koristeći naš generički sistem kojem dajemo skup ulaza a sistem proizvodi očekivani skup izlaza, tada je poboljšanje brzine (ili opterećenja memorije) potrebno za generisanje ovih rezultata validno za refaktorisanje.

Refaktorisanje u ovu svrhu je jedinstveno na jedan važan način: kao ishod ne obezbeđuje jasniji kôd. Ponekad ćemo čitati bazu koda i naići na poduži blok komentara sa upozorenjem o kodu ispod. Prema mom iskustvu, većina ovih komentara blokira čitaoca upozorenjem na jednu (ili više) komplikacija: neobično ponašanje aplikacije, privremena zaobilazna rešenja i neobične zakrpe zbog performansi. Većina poboljšanja performansi ispred kojih su ovi komentari napisana su pametno i uz duboko razumevanje baze koda, da bi se minimizovala oblast zahvaćena promenom. Ova „poboljšanja" podložnija su degradaciji tokom kraćeg perioda i kao takva nisu dobri primeri održivosti koju bi refaktorisanje trebalo da podstakne. Vredna poboljšanja performansi, ona koja treba da spadaju pod refaktorisanje, duboka su i dalekosežna; oni su primeri efikasnog refaktorisanja po potrebnoj meri. Ove promene ćemo detaljnije obraditi u Delu II.

Korišćenje nove tehnologije

U svetu razvoja softvera redovno usvajamo nove tehnologije. Bez obzira da li ćemo ići u korak sa najnovijim trendovima u našoj industriji, moramo pojačati našu sposobnost da se prilagodimo većem broju korisnika ili napraviti naš proizvod na novi način, mi neprestano procenjujemo nove biblioteke otvorenog koda, protokole, programske jezike, dobavljače usluga, i mnogo drugog. Donošenje odluke da koristimo nešto novo nije nešto što radimo olako; ovo je delimično zbog troškova integracije unutar naših postojećih baza koda. Ako se odlučimo da *zamenimo* postojeće rešenje novim, moramo da napravimo plan odustajnja od starog tako što ćemo identifikovati sve pogođene lokacije i migrirati ih (ponekad jednu po jednu). Ako se odlučimo da *usvojimo* novu tehnologiju, moramo da za rano usvajanje identifikujemo visoko zavisne kandidate, sa planom proširenja na sve relevantne slučajeve upotrebe.

Neću nabrajati svaki način na koji korišćenje nove tehnologije može uticati na vaš sistem (ima ih mnogo), ali iz ova dva scenarija je jasno da svaki zahteva pažljiv uvid u vaš trenutni sistem. Srećom, uvid može otkriti glavne kandidate za refaktorisanje! Želim da naglasim

da je ovo pomalo kontroverzno mišljenje. Zbog rizika koji dolaze sa usvajanjem nove tehnologije, drugi programeri mogu da vas odvrate od bilo kakvih drugih promena. Međutim, čvrsto sam ubeđen da je najgori način da u svoj sistem unesete nešto novo i da ga stavite uz veliku zapetljanu zbrku. Da bi mu se pružila najbolju šansu da ispuni svoju svrhu, mislim da je najbolje odvojiti vreme da prvo raščisti oblasti sa kojim će nova tehnologija doći u kontakt.

Ovaj koncept možemo lako primeniti na Simonovo radnju za hemijsko čišćenje. Recimo da je nedavno pustio u red neke nove, najsavremenije, ekološke mašine za hemijsko čišćenje. Da bi odredili plan postavljanja mašina, vlasnici shvataju da njihov postojeći raspored ima ozbiljnih neefikasnosti. Zaposleni moraju da prođu čitav niz mašina kako bi pokupili odeću sa mesta udaljenih gotovo deset metara. Ako preraspodele mašine tako da zaposleni treba da idu samo par metara da bi stigli do potrebnog mesta, možda će uštedeti nekoliko minuta u svakom ciklusu. Donose odluku da instaliraju nove mašine u revidiranoj konfiguraciji. Time je Simon možda smanjio uticaj na životnu sredinu i povećao produktivnost svojih zaposlenih. Uspeh, uspeh!

Kada ne refaktorisati

Refaktorisanje može biti zapanjujuće korisno sredstvo za programere. Mnogi programeri veruju da je vreme posvećeno refaktoringu uvek dobro utrošeno vreme, ali stvar nije tako jednostavna. Postoji vreme i mesto za refaktorisanje, a najzreliji programeri razumeju važnost znanja kada treba refaktorisati, a kada ne.

Iz zabave ili iz dosade

Zatvorite oči na minut i zamislite kako sedite ispred svog računara. Gledate posebno zapetljanu funkciju. Preduga je; pokušava da uradi previše stvari. Njeno ime odavno je prestalo da je opisuje. Svrbi vas da to popravite. Voleli biste da je podelite na jasno definisane, sažete jedinice sa boljim imenima promenljivih. Bilo bi *zabavno*. Ali da li je to najvažnija stvar koju biste trenutno trebali da radite? Možda vas saradnik čeka da pregledate kôd već nekoliko dana ili ste odlagali pisanje nekih testova? Ako kopate po nekakvom starom neupotrebljivom kodu i menjate ga kako biste se zabavili, možda sebi (i svojim saradnicima) činite medveđu uslugu.

Šanse su da, ako refaktorišete iz zabave, niste fokusirani na uticaj koji će vaša promena imati na okolni kôd, celokupan sistem i vaše saradnike. Imamo različite motivacije kada prepravljamo iz zabave: veća je verovatnoća da ćemo koristiti mogućnosti jezika koji sežu dalje ili isprobati potpuno novi obrazac koji smo želeli da isprobamo. Postoji vreme i mesto za isprobavanje novih stvari i rast naših programerskih mišića, ali vreme refaktorisanja nije to. Refaktorisanje bi trebalo da bude namerni proces gde se fokus usredsređuje na sprovođenje (u idealnom slučaju) najmanje promene uz najveće pozitivno dejstvo.