POGLAVLJE 2

Kako se kôd degradira

Uspešno istrčati maraton je impresivan podvig. Iako lično nikada nisam prihvatio izazov, dosta mojih prijatelja je to učinilo. Ono što će vas možda iznenaditi jeste da velika većina ovih prijatelja nisu bili strastveni trkači pre nego što su odlučili da se prijave za svoj prvi polumaraton ili puni maraton. Pridržavajući se redovnog, održivog rasporeda treninga, uspeli su da postignu potrebnu izdržljivost u roku od samo nekoliko meseci.

Većina mojih prijatelja već je bila u dobroj fizičkoj formi, ali ako vam je cilj da istrčite maraton, a većina vaših trenutnih fizičkih aktivnosti uključuje ustajanje sa kreveta i dohvatanje kesice čipsa iz ostave, neće vam biti lako. Ne samo da ćete prvo morati da poradite na kardiovaskularnoj i fizičkoj izdržljivosti redovno aktivne osobe, već ćete morati da usvojite nove navike oko redovnih vežbi i konzumiranja zdrave hrane (čak i kada je sve što želite da se smestite u udobnu stolicu sa velikim, slasnim komadom pice).

Mala kolebanja u treniniranju mogu dovesti do ozbiljnih zastoja. Ako niste dovoljno spavali ili vas je vreli dan uhvatio nespremnog, brže ćete se umoriti, ugrožavajući vašu sposobnost da pretrčite planiranu udaljenost. Čak i u punoj formi za maraton, morate biti spremni za neočekivano na dan trke. Možda kiša; pertle su vam se pokidale; možete biti zaglavljeni u gomili trkača. Naučili ste da savladate promenljive koje možete kontrolisati, ali morate biti spremni i da razmišljate dok ste u trci.

Biti programer je pomalo kao biti maratonac. I jedan i drugi se neprekidno trude. Oboje se nadograđuju na prethodni napredak, korak po korak, kilometar po kilometar. Ako se iskreno potrudite da održite zdrave navike, to čini razliku između mogućnosti vraćanja u maratonsku formu ili u visok tempo razvoja, od nekoliko nedelja naspram više meseci. Održavanje visokog nivoa budnosti i u vašem unutrašnjem i u spoljnom okruženju i blagovremeno prilagođavanje je ključno za uspešno završavanje trke. Isto se može primeniti i na razvoj: visok nivo budnosti nad stanjem baze koda i bilo kakvim spoljnim uticajima ključno je za minimizovanje zastoja i za osiguravanje nesmetanog puta do cilja.

U ovom poglavlju ćemo razmotriti zašto je razumevanje degradacije koda ključno za uspešan napor refaktorisanja. Razmotrićemo kôd koji stagnira i koji je u aktivnom razvoju i opisaćemo načine na koje svako od ovih stanja može doživeti degradaciju koda, uz nekoliko primera izvučenih iz nedavne i rane istorije računarskih nauka. Na kraju ćemo reći nešto o načinima na koje možemo rano otkriti degradaciju i kako bismo je uopšte mogli sprečiti.

Zašto je razumevanje degradacije koda važno

Kôd se degradirao kada je uočeno da se njegova korisnost smanjila. To znači da se kôd, iako je bio zadovoljavajući, ili se više ne ponaša onako kako bismo želeli ili ga nije lako pročitati ili koristiti iz perspektive razvoja. Upravo iz ovih precizno navedenih razloga degradirani kôd je odličan kandidat za refaktorisanje. S tim u vezi, čvrsto verujem da ne možete krenuti u poboljšanje nečega dok čvrsto ne shvatite njegovu istoriju.

Kôd nije stvoren u vakuumu. Ono što bismo danas mogli smatrati lošim kodom je verovatno bilo dobro kada je prvobitno napisan. Posvetivši vreme da razumemo okolnosti pod kojima je kôd prvobitno napisan i kako je vremenom mogao preći iz dobrog u loše, možemo izgraditi bolje razumevanje o srži problema, steći osećaj za zamke u koje se može upasti, a i preuzeti pravi korak da ga vratimo iz lošeg u dobro stanje.

Uopšteno govoreći, postoje dva načina na koja se kôd može degradirati. Ili su se promenili zahtevi za tim šta kôd treba da radi ili je vaša organizacija pravila kompromise u nastojaju da postigne što više u kratkom periodu. Nazvaćemo prvo „promena zahteva", a drugo „tehnološki dug".

Važno je da ne pretpostavite da su sve degradacije koda na koje nalećete posledica tehnološkog duga, zbog čega ćemo prvo pogledati razne načine na koje promena zahteva može vremenom da kvari kôd. Svi znamo onaj trenutak kada ćemo naići na neki posebno strašan kôd i pomisliti: „Ko je ovo napisao? Kako smo mogli dopustiti da se ovo dogodi? Zašto ovo niko nije popravio? " Ako odmah počnemo da ga prepravljamo, rizikujemo da napravimo rešenje koje prenaglašava ono što smatramo najneugodnijim u vezi sa kodom, umesto da se bavimo njegovim istinskim, suštinskim problemom. Važno je da izgradimo empatiju prema kodu tražeći od sebe da identifikujemo šta se promenilo od kada je napisan. Ako se potrudimo da potražimo šta je kodu bilo početno dobro, shvatićemo zamke koje je prvobitno rešenje izbeglo, pametne načine na koje se nosio sa nizom ograničenja i uraditi refaktorisanje koje će ispoštovati sve te uvide.

Nažalost, postoje trenuci kada jednostavno moramo dati sve od sebe, s obzirom na vrlo ograničene resurse. Kada nemamo dovoljno vremena ili novca za stvaranje boljeg rešenja, počinjemo da pravimo kompromise i sečemo a tehnološki dug raste. Iako bi početni uticaj tog duga mogao biti minimalan, njegova dodatna težina na našim bazama koda može vremenom značajno porasti. Lako je odbaciti tehnički dug odbacivši loš kôd, ali pozivam vas da prepravite kôd. Ponekad je najbrže rešenje ono koje vaš proizvod ili funkciju najbrže plasira na tržište; ako je prelazak vašeg proizvoda u ruke korisnika presudan za opstanak vaše kompanije, onda bi se tehnološki dug mogao isplatiti.

Dok prolazite kroz načine na koje se kôd može degradirati, podstičem vas da pokušate da pronađete slučajeve svakog od njih u kodu sa kojim vi najčešće radite. Možda nećete moći da pronađete primer za sve slučajeve, ali proces pretraživanja simptoma degradacije koda mogao bi vas odvesti do toga da razvijete novu perspektivu za delove aplikacije sa kojima ste najviše frustrirani.

Jednom kada ste odredili kôd koji želite da refaktorišete, steći ćete dragoceni uvid u to kako i zašto se prvobitno rešenje originalnih autora degradiralo i šta bi vam autori rekli. Ako bi autori rekli nešto poput „nismo znali da ...“ ili „u to vreme smo mislili ...“, verovatno imate slučaj degradacije koda zbog promene zahteva. S druge strane, ako autori kažu nešto poput: „oh, tačno, taj kôd nikada nije bio dobar“ ili „samo smo pokušavali ispoštovati rok“, znate da se verovatno bavite standardnim slučajem tehnološkog duga.

Promena zahteva

Kad god napišemo novi deo koda, idealno je da provedemo neko vreme izričito definišući njegovu svrhu i praveći detaljnu dokumentaciju koja demonstrira namenu upotrebe. Iako bismo se potrudili da predvidimo sve buduće zahteve i pokušamo da osmislimo pametan sistem koji će moći da se nosi sa tim novim zahtevima, malo je verovatno da ćemo moći da predvidimo sve što će doći. Prirodno je da će se vremenom okruženja oko naših aplikacija nepredvidivo promeniti. Ove promene mogu uticati i na kôd koji je u aktivnoj izradi i na kôd koji je ostao netaknut u različitom stepenu. U ovom delu razmotrićemo nekoliko načina na koje zahtevi koji se postavljaju pred naš kôd mogu premašiti njegove mogućnosti, koristeći primere iz baze koda u aktivnom i neaktivnom razvoju.

Proširivost

Jedan od zahteva koji često pokušavamo da procenimo je smer i stepen do kojeg naš proizvod treba da se proširi. Ova lista zahteva može biti prilično dugačka i može sadržati širok spektar parametara. Uzmimo, na primer, zahtev za jednostavnim interfejsom za programiranje aplikacije (API) za postizanje novog korisnikovog unosa u sistem. Mogli bismo postaviti neke smernice oko očekivanog kašnjenja zahteva, broja upita u bazu podataka po zahtevima, ukupnog broja dozvoljenih novih korisničkih zahteva u sekundi itd.

Pre puštanja u pogon novog proizvoda, jedna od naših prvih pretpostavki bavi se brojem korisnika koje očekujemo da će ga koristiti. Pravimo rešenje za koje mislimo da će glatko obraditi taj broj (plus minus margina greške) i isporučujemo ga! Iako je naš proizvod uspešan, možemo završiti sa eksponencijalno više korisnika nego što smo u početku predvideli, i iako je to iz poslovne perspektive sigurno neverovatna situacija, naša originalna primena verovatno neće moći da podnese ovo novo, nepredviđeno opterećenje. Sam kôd se možda nije promenio, ali je efektivno nazadovao zbog drastične izmene zahteva za proširenje.

Pristupačnost

Svaka aplikacija treba da se trudi da bude što dostupnija od prvog dana. Trebali bismo da koristimo šeme boja koje razlikuju i daltonisti, da dodamo alternativne tekstove za slike i ikone i da obezbedimo da se svakom interaktivnom elementu može pristupiti

preko tastature. Nažalost, timovi koji žure sa isporukom novog proizvoda ili funkcije često zamute pristupačnost zbog tesnog datuma isporuke. Iako vam brza isporuka novih funkcija može pomoći da zadržite trenutne korisnike i privučete nove, ako ove funkcije nisu dostupne određenim korisnicima, rizikujete da vas napuste. Čim vaš proizvod nekome postane nepristupačan, njegova opažajna korisnost se znatno smanjuje.

Iako su najbolje službene prakse za veb pristupačnost razvijene od strane Web Accessibility Initiative (WAI) 1999. godine pretrpele nekoliko važnih revizija, standardizovane su. U svakoj iteraciji, programeri aktivnih veb lokacija i aplikacija moraju ponovo da posete dugo nedirnuti kôd i da primene sve neophodne promene kako bi bili u skladu sa najnovijim standardima. Neprimenjivanje standarda pristupačnosti može smanjiti kvalitet vaše aplikacije.

Kompatibilnost uređaja

Svake godine hardverske kompanije puštaju u prodaju nove verzije svojih uređaja; ponekad će čak napraviti korak dalje i predstaviti potpuno novu klasu uređaja. Među pametnim telefonima, pametnim satovima, pametnim automobilima i pametnim televizorima, neprestano moramo da ih sustižemo, pokušavajući da prepakujemo svoje aplikacije kako bi neometano radile na najnovijem hardveru. Korisnici su evoulirali da očekuju da njihove omiljene aplikacije rade na raznim platformama. Ako ste programer za popularnu mobilnu igru a velika hardverska kompanija izbaci novi uređaj sa većom rezolucijom ekrana, rizikujete da izgubite značajan deo svoje korisničke baze ukoliko ne isporučite novu verziju igre napravljenu za veći ekran.

Promene u okruženju

Kada se promene dogode u okruženju programa, sve vrste neočekivanog ponašanja mogu početi da se manifestuju. Pre doba modernih računara za igre napunjenih moćnim hardverom za obradu grafike (GPU) i desetinama gigabajta memorije (RAM), imali smo skromne male igraće konzole smeštene u igraonicama, a kasnije i u našim sobama. Programeri igara osmislili su pametne načine kako da koriste ograničeni hardver koji im je tada bio dostupan da naprave klasike poput *Space Invaders* i *Super Mario Bros*. U to vreme bila je uobičajena praksa da se takt centralne procesorske jedinice (CPU) koristi kao tajmer u igri. Pružao je stabilnu, pouzdanu meru vremena. Iako ovo nije predstavljalo problem za igre na konzolama, gde ni kertridži često nisu bili kompatibilni sa novijim moćnijim iteracijama konzole, to je postalo prilično ozbiljna stavka za igre koje se pokreću na ličnim računarima. Kako se brzina takta na novijim računarima povećavala, povećavala se i brzina igranja. Zamislite da morate slagati Tetris ili izbegavati tok Goombasa dvostruko većom brzinom od normalne; u određenom trenutku igra postaje potpuno neupotrebljiva. U oba ova primera zahtev je bio da se kôd izvodi na određenom fizičkom hardveru; nažalost, hardver se od tada dramatično promenio i kao rezultat toga, kôd se degradirao.

Ove vrste promena u okruženju i danas je ozbiljna briga. U januaru 2018. istraživači bezbednosti iz Google Project Zero i Cyberus Technologi, u saradnji sa timom Graz University of Technology, identifikovali su dve ozbiljne bezbednosne ranjivosti koje pogađaju sve Intel x86 mikroprocesore, IBM POWER procesore i neke Advanced RISC Machine (ARM) mikroprocesore. Prvi, Meltdown, omogućio je zlonamernim procesima da pročitaju celu memoriju na mašini, čak i bez ovlašćenja za to. Drugi, Spectre, omogućio je napadačima da iskoriste predviđanje grane (eng. *branch prediction*) (osobina koja poboljšava performansu pogođenih procesora) kako bi otkrili privatne podatke o drugim procesima koji se izvode na mašini. Više o ovim ranjivostima i njihovom unutrašnjem delovanju možete pročitati na službenoj veb stranici meltdownattack.com.

U vreme otkrivanja ranjivosti, to je uticalo na sve uređaje koji nisu imali najnovijiu verzija iOS-a, Linuxa, macOS-a i Windovsa. Pogođeni su brojni serveri i usluge u oblaku, kao i većina pametnih uređaja i ugrađenih uređaja. Za nekoliko dana softverska zakrpe postale su dostupne za obe ranjivosti, ali cena je bila smanjenje performansi od 5 do 30 procenata, u zavisnosti od radnog opterećenja. Intel je kasnije izvestio da radi na pronalaženju načina da pomogne u zaštiti od Meltowna i Spectre u sledećoj seriji procesora. Čak i stvari za koje verujemo da su najstabilnije (operativni sistemi, firmver) podložne su promenama u svom okruženju; i kada su ovi osnovni sistemi koji čine jezgro pogođeni, bezbroj aplikacija koje pokrećemo na njima je takođe pogođeno.

Spoljne zavisnosti

Svaki komad softvera ima spoljne zavisnosti; da navedemo samo nekoliko primera, to mogu biti skup biblioteka, programski jezik, intepreter ili operativni sistem. Stepen u kome su ove zavisnosti povezane sa softverom može se razlikovati. Ova povezanost nije ništa novo; mnogi važni programi iz ranih dana istraživanja veštačke inteligencije razvijeni su u Lispu i istraživačkim programskim jezicima sličnim Lispu, jer su se aktivno razvijali šezdesetih i ranih sedamdesetih. SHRDLU, rani računarski program koji razume prirodni jezik, napisan je u Micro Planneru na PDP-6, koristeći nestandardne makronaredbe i softverske biblioteke koje danas više ne postoje, i nalazi se u stanju nepopravljivog propadanja.

Danas se trudimo da ažuriramo naše spoljne zavisnosti kako bismo bili u toku sa najnovijim funkcijama i bezbednosnim zakrpama. Međutim, ponekad nam nije u prioritetu ili smo izgubili trag o ažuriranjima, posebno kada je reč o kodu koji ne održavamo aktivno. Iako dopuštanje da zavisnosti zaostaju za nekoliko verzija možda ne predstavlja neposredni problem, ipak postoji rizik. Postajemo podložniji bezbednosntim ranjivostima. Takođe sebe uvodimo u potencijalno tešku nadogradnju.

Recimo da pokrećemo program koji se oslanja na verziju 1.8 biblioteke otvorenog koda pod nazivom Super Timezone Library. Samo nekoliko nedelja posle objavljivanja verzije 4.0, programeri Super Timezone Library najavljuju da više neće aktivno podržavati verzije ispod 3.0. Sada moramo da se nadogradimo na verziju 3.0 najmanje, da bismo

nastavili da instaliramo sigurnosne zakrpe. Nažalost, verzija 2.5 je uvela neke promene koje nisu unazad kompatiblne a verzija 2.8 je odbacila neke funkcionalnosti koja se uveliko koristi u našoj aplikaciji. Ono što je moglo biti malo, redovno ulaganje u održavanje biblioteke u toku poslednjih nekoliko godina, sada se pretvorilo u mnogo složeniju, hitniju investiciju.

Neiskorišćen kod

Promene u okruženju mogu dovesti do neiskorišćenog koda. Uzmimo na primer javni API. Vaš tim odlučuje da obustavi rad API-ja i upozorova nezavisne programere na predstojeću promenu. Nažalost, pošto ste saopštili nameravanu promenu, uklonili dokumentaciju sa veb lokacije i potvrdili da se nijedan spoljni sistem i dalje ne oslanja na njega, vaš tim zaboravlja da ukloni kôd. Nekoliko meseci kasnije, novi inženjeri počinju da primenjuje novu funkciju, nailaze na povučenu ali i dalje aktivnu krajnju tačku API-ja i pretpostavljaju, sasvim prirodno, da je i dalje u funkciji. Odlučuju da ga upotrebe za svoj slučaj. Nažalost, brzo otkriju da kôd ne radi baš ono što su očekivali, jednostavno zato što je API ostao u prošlosti i što se nije prilagodio ostatku baze koda i brojnim iteracijama zbog promena zahteva.

Neiskorišćen kôd takođe može biti problematičan sa stanovišta produktivnosti programera. Svaki put kada se susretnemo sa kodom za koji verujemo da je neiskorišćen, moramo vrlo pažljivo utvrditi da li ga možemo bezbedno ukloniti. Ako nismo opremljeni pouzdanim alatima koji će nam pomoći da pravilno istaknemo obim mrtvog koda, možda ćemo imati poteškoća da odredimo njegove tačne granice. Ako nismo sigurni da li možemo da ga izbrišemo, obično idemo dalje i nadamo se da će neko drugi to kasnije moći da shvati. Ko zna koliko će inženjera naići na isti komad koda i postaviti sebi isto pitanje pre nego što konačno bude uklonjen!

Konačno, neiskorišćen kôd, ako mu se dozvoli da se nagomila, može predstavljati smetnju performansama. Ako, na primer, vaš tim radi na delu veb stranice namenjene klijentu, veličina JavaScript datoteka koje vaš veb čitač traži direktno povećavaju vreme učitavanja stranice. Obično je što je datoteka veća, odgovor je sporiji. Nekritički pretrpan kôd aplikacije može biti prilično loš za iskustvo korisnika.

Isključen kôd

U slučaju isključenog koda, jasno je da se kôd ne koristi. Uvek preporučujem programerima koji dolaze u iskušenje da isključe kôd, umesto toga, da ga jednostavno obrišu ako se razvoj koda prati kroz kontrolne verzije. Ako vam jednog dana ponovo zatreba, možete ga lako povratiti vraćanjem iz istorije promena.

Promene u zahtevima proizvoda

U većini slučajeva je lakše napisati rešenje za današnje ili sutrašnje zahteve proizvoda, rešavajući probleme i ograničenja koja razumemo i koje možemo lako predvideti, nego napisati rešenje za sledeću godinu, pokušavajući rešiti nepoznate buduće zamke. Trudimo se da budemo pragmatični, odmeravajući trenutne brige u odnosu na buduće i pokušavamo da utvrdimo koliko vremena bi trebalo uložiti u rešavanje obe. A ponekad jednostavno nemamo dobru intuiciju o budućnosti.

Bulovi argumenti funkcija su sjajan primer poteškoće u predviđanju budućih zahteva u vezi proizvoda. Većinom se logički argumenti uvode u postojeće funkcije radi izmene njihovog ponašanja. (Videli smo jedan u „Naš prvi primer refaktorisanja“ na stranici 16, gde se pomoću Bulove vrednosti odlučivalo da li želimo da znamo da li je svaka ocena ili prosek tih ocena u zadatom opsegu.) Dodavanje Bulove vrednosti je često najmanja, najjednostavnija promena koju možete da napravite kada pronađete funkciju koja radi gotovo tačno ono što vi želite, sa vrlo malim izuzetkom. Nažalost, ova vrsta promena može uzrokovati sve vrste problema. Neke od njih možemo videti u primeru 2-1, gde imamo malu funkciju odgovornu za slanje na server slike koja ima naziv datoteke i Bulov indikator koji označava da li je datoteka tipa PNG.

Primer 2-1. Funkcija sa Bulovim argumentom

```
function uploadImage(filename, isPNG) {
  // detalji implementacije
  if (isPNG) {
    // odgovarajuća PNG logika
  }
  // uradi ostalo
}
```

Šta ako, za nekoliko meseci, odlučimo da podržimo novi format slike? Možda bismo odlučili da dodamo još jedan logički argument za označavanje `isGIF`, kao što je prikazano u primeru 2-2.

Primer 2-2. Funkcija sa dva Bulova argumenta

```
function uploadImage(filename, isPNG, isGIF) { ❶
  // detalji implementacije
  if (isPNG) {
    // odgovarajuća PNG logika
  } else if (isGIF) { ❷
    // odgovarajuća GIF logika
  }
  // uradi ostalo
}
```

❶ Uveden je novi Bulov argument koji označava da li je slika GIF.

❷ Slika ne može biti i PNG i GIF, pa smo dodali `else if`.

Da bismo pozvali ovu funkciju i pravilno otpremili GIF, trebali bismo da postavimo drugi logički argument na `true`. Čitaoci koji naiđu na kôd koji poziva `uploadImage` verovatno bi bili zbunjeni i morali bi da se pozovu na definiciju funkcije da bi razumeli koju ulogu igraju dva logička argumenta.

U jeziku sa imenovanim argumentima, manje bismo bili zabrinuti potrebom da se referenciramo na definiciju funkcije da bismo znali ulogu i redosled argumenata. Bez obzira na izbor jezika, ostaje da `uploadImage(filename=filename, isPNG=true, isGIF=true)` može izgledati besmisleno, ipak to je savršeno važeći poziv funkcije (i vrlo je verovatno da će u budućnosti uzrokovati greške). Primer 2-3 prikazuje primer gde čitaocu može biti teško da prepozna šta `uploadImage` radi s obzirom na kontekst.

Primer 2-3. Funkcija slanja na server GIF-a

```
function changeProfilePicture(filename) {
  // detalji implementacije
  if (isAnimated) {
    uploadImage(filename, false, true); ❶
  } else {
    uploadImage(filename, true, false); ❷
  }
  // uradi ostalo
}
```

❶ Ovde prenosimo GIF na server.

❷ U suprotnom, prenosimo PNG.

Ne samo da je programerima teško da razumeju kako `uploadImage` radi prilikom čitanja funkcija poput `changeProfilePicture`, nego je neodrživ obrazac koji treba i dalje održavati ako se u budućnosti uvede više formata slika. Programer koji je dodao prvi logički argument kao podršku `isPNG` uglavnom se bavio tadašnjim problemima, a ne sutrašnjim. Bolji pristup bi bio podeliti logiku na različite funkcije: `uploadJPG`, `uploadPNG` i `uploadGIF`, kao što je prikazano u primeru 2-4.

Primer 2-4. Izdvojene funkcije za slanje različitih vrsta datoteka

```
function uploadImagePreprocessing(filename) {
  // detalji implementacije
}

function uploadImagePostprocessing(filename) {
  // uradi neke druge stvari
}

function uploadJPG(filename) {
  uploadImagePreprocessing();
  // uradi JPG
```

```
  uploadImagePostprocessing();
}

function uploadPNG(filename) {
  uploadImagePreprocessing();
  // uradi PNG
  uploadImagePostprocessing();
}

function uploadGIF(filename) {
  uploadImagePreprocessing();
  // uradi GIF
  uploadImagePostprocessing();
}
```

Sad se možda pitate zašto je dodavanje logičkog argumenta isPNG ozbiljan problem ako ga kasnije možemo samo refaktorizovati. Da bismo pravilno zamenili sve pojave `uploadImage`, trebalo bi da revidiramo svako pozivno mesto pojedinačno i zamenimo ga ili sa `uploadJPG` ili sa `uploadPNG`, u zavisnosti od toga da li je logički argument postavljen na `true`. Budući da su ove promene ručne, ali svakodnevne, verovatnoća da ćemo pogrešno zameniti prilično je velika i mogla bi dovesti do ozbiljnog nazadovanja. U zavisnosti od toga koliko bi problem mogao biti raširen i koliko bi mogao biti usko povezan sa nekom drugom ključnom poslovnom logikom, refaktorisanje onog što izgleda kao jednostavan logički argument mogao bi biti zastrašujući zadatak.

Tehnološki dug

Najčešći krivci za tehnološki dug su ograničeno vreme, ograničeni broj inženjera i ograničeni novac. S obzirom na to da su sve tehnološke kompanije suočene sa ograničenim resursima po jednoj ili više osnova, svaka od njih ima tehnološki dug. Mali, šestomesečni startapi; džinovski, višedecenijski konglomerati; i svaka od njih ima popriličan deo zapuštenog koda. U ovom odeljku ćemo detaljnije razmotriti uticaje koji mogu dovesti do akumulacije tehnološkog duga. Iako je lako upirati prstom u originalne autore koda i optuživati ih da donose odluke koje se danas čine neoptimalnim, važno je imati na umu da su radili pod ozbiljnim pritiscima i ograničenjima. Moramo priznati da je ponekad gotovo nemoguće napisati dobar kôd pod velikim pritiskom.

Zaobilaženje tehnoloških izbora

Kada primenjujemo nešto novo, moramo doneti neke kritične odluke o tome koje tehnologije želimo da koristimo. Moramo da izaberemo jezik, program za rad sa zavisnostima (eng. *dependency manager*), bazu podataka itd. Postoji prilično dugačak spisak odluka koje treba doneti pre nego što aplikacija postane dostupna svim korisnicima. Mnoge od ovih odluka donose se s obzirom na iskustvo inženjera; ako je ovim inženjerima ugodnije da koriste jednu tehnologiju u odnosu na drugu, lakše će pokrenuti projekat i brže će učiniti da radi nego da su odlučili da usvoje novi paket.

Jednom kada je projekat pokrenut i počne ozbiljnije da radi, ove rane tehnološke odluke stavljaju se na test. Ako se problem sa izborom tehnologije pojavi dovoljno rano u životnom veku aplikacije, možda će biti lako i jeftino pronaći odgovarajuću alternativu i okrenuti se ka njoj, ali često ograničenja učinjenih izbora postaju očigledna tek kada aplikacije pređe ranu tačku u svom životnom veku.

Jedna od takvih odluka mogla bi biti razvoj aplikacije korišćenjem programskog jezika sa dinamičkim tipovima podataka umesto programskog jezika sa statičkim tipovima podataka.

Pristalice programskih jezika sa dinamičkim tipovima tvrde da oni olakšavaju čitanje i razumevanje koda; manje indirektnosti oko strogo definisanih struktura i deklaracija tipa omogućavaju čitaocu da bolje i lakše razume svrhu koda. Mnogi takođe tvrde da je brži razvojni ciklus koji pružaju zbog nepostojanja vremena kompajliranja.

Iako postoji mnogo dobrih strana u korišćenju dinamički tipiziranih programskih jezika, njima postaje teško upravljati kada aplikacije pređu kritičnu masu. Budući da se tipovi podataka verifikuju samo tokom izvođenja, odgovornost programera je da osigura ispravnost tipova pisanjem celokupnog skupa jediničnih testova koji izvršavaju sve putanje u programu i utvrđuju očekivano ponašanje. Novi programeri koji žele da se upoznaju sa načinom na koji različite strukture međusobno komuniciraju mogu imati poteškoća ako imena promenljivih odmah ne označavaju kog su tipa. Ne retko je da na kraju bude potrebno da se programira defanzivno, kao što je prikazano u primeru 2-5, gde tvrdimo da vrednost prosleđena u funkciju ima određena svojstva i da nije nenamerno `null`.

Primer 2-5. Odbrambeno programiranje na delu

```
function addUserToGroup(group, user) {

  if (!user) {
    throw 'korisnik ne može biti null';
  }

  // obezveđuje potrebna polja
  if (!user.name) {
    throw 'ime je potrebno';
  }

  if (!user.email) {
    throw 'email je potreban';
  }

  if (!user.dateCreated) {
    throw 'datum stvaranja je potrean';
  }
```

```
  // ne dozvoljava prazne znakovne nizove ili druge nedozvoljene vrednosti
  if (user.name === "") {
    throw 'ime ne može da bude prazno';
  }

  if (user.email === "") {
    throw 'email ne možda da bude prazno';
  }
  if (user.dateCreated === 0) {
    throw 'datum stvaranja ne može biti 0';
  }
  group.push(user);
  return group;
}
```

Verovatno je da autor koda gornjeg primera redovno nailazi na probleme sa korisnicima koji se provuku kroz zadate uslove, jednostavno zbog dinamičke prirode JavaScripta. Autor jednostavno želi da bude siguran da u grupu dodaje samo važeće korisnike, i to je potpuno razumljivo. Nažalost, sada se `addUserToGroup` prvenstveno bavi obezbeđivanjem da je validno ono što je korisnik uneo, umesto dodavanjem korisnika u grupu. Kako se donosi više odluka o tome šta predstavlja važećeg korisnika, svaka od ovih ad hok validacija rasutih u bazi koda mora biti ažurirana. Takođe je sve veća šansa da unesemo grešku jednostavnim zaboravljanjem da ažuriramo jednu takvu lokaciju. Na kraju, svuda imamo duge, zamršene, funkcije sklone greškama.

Možemo uvesti novu funkciju koja ublažava propadanje koda. Recimo da napišemo jednostavnu pomoćnu funkciju koja obuhvata svu logiku za proveru valjanosti objekta `user`; nazvaćemo je `validateUser`. Primer 2-6 prikazuje njenu primenu.

Primer 2-6. Jednostavna pomoćna funkcija za enkapsulaciju logike validacije korisnika

```
function validateUser(user) {
  if (!user) {
    throw 'korisnik ne može biti null';
  }

  // obezveđuje potrebna polja
  if (!user.name) {
    throw 'ime je potrebno';
  }

  if (!user.email) {
    throw 'email je potreban';
  }

  if (!user.dateCreated) {
    throw 'datum stvaranja je potrean';
  }
```

```
  // ne dozvoljava prazne znakovne nizove ili druge nedozvoljene vrednosti
  if (user.name === "") {
    throw 'ime ne može da bude prazno';
  }

  if (user.email === "") {
    throw 'email ne možda da bude prazno';
  }
  if (user.dateCreated === 0) {
    throw 'datum stvaranja ne može biti 0';
  }

  return;
}
```

Zatim možemo ažurirati `addUserToGroup` tako da koristi našu novu pomoćnu funkciju, drastično pojednostavljujući logiku, kao što je prikazano u primeru 2-7.

Primer 2-7. Pojednostavljena funkcija `addUserToGroup` *bez ugrađene logike provere*

```
function addUserToGroup(group, user) {
  validateUser(user);
  group.push(user);
  return group;
}
```

Nažalost, iako nam je mnogo lakše da pozovemo `validateUser`, zamena svih lokacija na kojima smo prethodno nabrojali svaku proveru biće lak zadatak. Prvo, moramo da identifikujemo svako od tih mesta. Ako imamo posla sa velikom bazom koda, to bi mogao biti zastrašujući zadatak. Drugo, tokom revizije svake od ovih lokacija verovatno ćemo na kraju naći nekoliko slučajeva gde smo zaboravili jednu ili dve. U nekim slučajevima je ovo greška i provere možemo bezbedno zameniti pozivom na `validateUser`; u drugim slučajevima moglo je biti namerno i ne možemo slepo zameniti postojeći kôd novim pomoćnom funkcijom rizikujući da uvedemo regresiju. Zbog toga, da bismo olakšali teret našeg odbrambenog programiranja, moramo da planiramo i izvršimo značajno refaktorisanje.

Trajni nedostatak organizacije

Održavanje organizovane baze koda pomalo liči na održavanje urednog doma. Deluje kao da uvek imamo da uradimo nešto važnije nego da odložimo odeću nagomilanu preko komode ili da razvrstamo hrpu pošte koja se nakupila na stočiću. Ali što više ostavljamo, više vremena ćemo provesti pročešljavajući sve to kada konačno odlučimo da se latimo raščišćavanja. Možete čak da dozvolite da se nered nakupi do te mere da je počeo da se preliva na druge površine. Moji roditelji su imali jasan cilj kada su me podsticali da svakodnevno održavam stvari urednim i da ih pomalo čistim; znali su da je uvek mnogo lakše brinuti se o malom neredu nego o velikom.

Mnogi od nas upadaju u iste obrasce kada je u pitanju održavanje organizovanim naših baza kodova. Uzmimo, na primer, bazu koda sa relativno ravnom strukturom datoteka. Većina koda je organizovana u dvadesetak datoteka, sa jednim direktorijumom za testove. Aplikacija raste stalnim tempom, dodajući nekoliko novih datoteka svakog meseca. Budući da je lakše održavati status kvo, umesto da proaktivno počnu da organizuju povezane datoteke u posebne direktorijume, inženjeri su umesto toga naučili da se kreću po sve širem kodu. Novi inženjeri koji su ušli u rastući haos podižu zastavu upozorenja i podstiču tim da započne sa podelom koda, ali zabrinutost stiže do gluvih ušiju; menadžeri ih podstiču da se usredsrede na rokove koji nadolaze, a stariji inženjeri sležu ramenima i uveravaju ih da će brzo shvatiti kako da budu produktivni u rasulu. Na kraju, baza koda dostiže kritičnu masu u kojoj je stalni nedostatak organizacije dramatično usporio produktivnost tima. Tek tada tim odvoja vreme za izradu plana uređivanja baze koda, kada je broj promenljivih koje treba uzeti u obzir daleko veći nego što bi bio, da su uložili zajednički napor u rešavanju problema mesecima (ili čak godinama) ranije.

Previše babica

Loše organizovan kôd može dovesti do još brže degradacije u kombinaciji sa brzim zapošljavanjem. Kompanije koje se brzo razvijaju možda će imati na desetine novih inženjera svakog meseca. Ovi inženjeri su nestrpljivi da zarone i započnu angažovanje oko koda, ali bez dobro definisane strukture i stila rizikuju da prodube postojeće problematične obrasce duboko ukorenjene u bazu koda.

Sa previše inženjera koji rade u istoj bazi koda, vi definišete ergonomiju koja nije nužno na osnovu onoga što je najbolje za dugoročno zdravlje baze koda, već oko onoga što najbolje funkcioniše znajući da ćete morati raditi sa drugim saradnicima. To može dovesti do dugotrajnog, odbrambenog koda ili suboptimalno postavljenog koda kako bi se izbegli potencijalni sukobi pri spajanju koda.

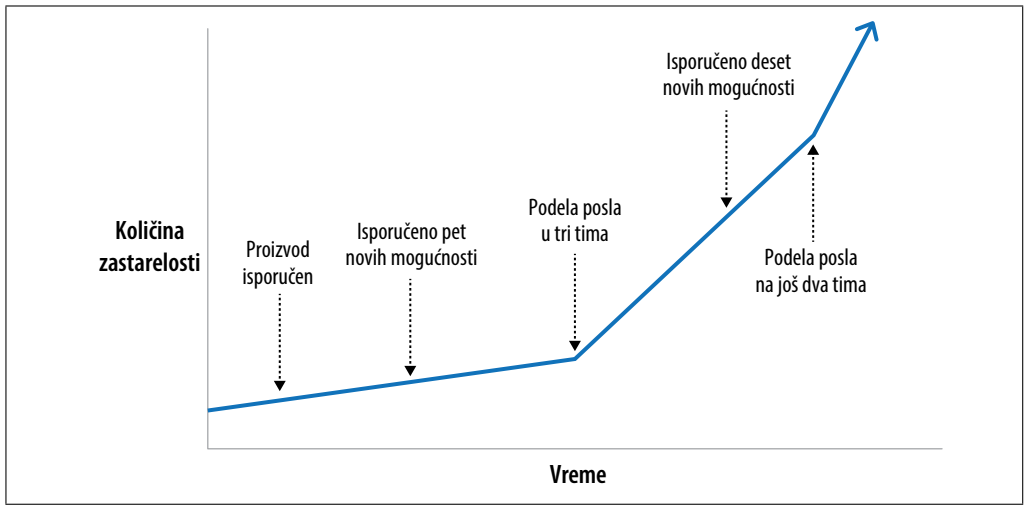
Prebrzo kretanje

Brze iteracije i razvoj proizvoda mogu za čas pogoršati kvalitet softvera ako se ne kontroliše. Kada stvaramo nove mogućnosti proizvoda pod agresivnim rokovima, skloni smo izostavljanju: izostavićemo neke slučajeve u testiranju, promenjivama ćemo dati generička imena ili dodati nekoliko nardbi `if` za koje smo mogli da napravimo nove funkcije. Ako ne zabeležimo pravilno ono što smo preskočili i ako ne odvojimo potrebno vreme da to ispravimo odmah pošto što smo ispunili traženi rok, zapušten kod se gomila. Ubrzo ćete završiti širom vaše baze koda sa izuzetno dugim funkcijama, prepunim granama sa logikom i sa malo ili nimalo jediničnih testova. Kada radite u složenijim aplikacijama, gde više timova izvršava ponavljanje različitih mogućnosti programa, posledice prebrzog kretanja počinju da se gomilaju. Ukoliko svaki tim ne može efikasno

da odkomunicira promene proizvoda sa svakim drugim timom, loša gomila koda raste. Možete videti primer tog efekta gomilanja prikazan na slici 2-1.

Mnogi od nas koji rade na modernim aplikacijama praktikuju kontinuiranu integraciju i isporuku; naše promene ulivamo natrag u glavnu granu što je češće moguće, gde se proveravaju pokretanjem automatskih testova na novoj verziji aplikacije.

Obezbeđujemo da kupci ne dobiju polovične mogućnosti aplikacije ni delimično ispravljene greške stavljanjem ovih promena iza indikatora, zastavica mogućnosti aplikacije (poznatih kao preklopnik mogućnosti, eng. *features toggles*). Iako nam to pruža dobru dozu fleksibilnosti tokom aktivnog razvoja, njih je lako zaboraviti nakon što smo promenu uveli svim korisnicima.



Slika 2-1. Grafikon nakupljanja zastarlosti tokom vremena

Svaka kompanija u kojoj sam radio imala je na desetine (ako ne i stotine) indikatora mogućnosti u programu koji su se pozivali i nakon što su bili aktivirani u celosti. Iako se može činiti benignim ostaviti nekoliko ovih indikatora u pogonu, postoje određeni rizici.

Prvo, to dovodi do dodatnog kognitivnog opterećenja onih programera koji čitaju kôd; ako programer ne odvoji vreme da verifikuje status mogućnosti, tj. funkcije, mogao bi da dođe u zabludu misleći da je još uvek u aktivnom razvoju i da napravi važnu promenu u neaktivnoj putanji koda. Drugo, može biti frustrirajuće trošiti vreme na utvrđivanje da li je mogućnost i dalje aktivna u proizvodnji, samo da bi se otkrilo da je itekako živa. U težim slučajevima kada postoje stotine zastavica koje u osnovi ne služe ničemu, to može imati veoma ozbiljan uticaj na performanse aplikacije. Kumulativno vreme provedeno za proveru svakog uslova vezanog za mogućnosti, tj. funkciju ili putanje

može biti značajno. Mogli bismo da vidimo određeno poboljšanje performansi raščišćavanjem zastarelih zastavica.

Primena našeg znanja

Propadanje koda je neizbežno. Koliko god se trudili da ga izbegnemo, doći će do promena u zahtevima kojima će se naše aplikacije morati prilagoditi. Možemo da pokušamo da minimiziramo razvoj pod pritiskom, ali ponekad moramo da nešto izostavimo da bismo se brzo isporučili i da u svom poslu budemo konkurentni. Ako je degradacija koda neizbežna, onda je i refaktorisanje po potrebnoj meri isto tako neizbežno. Uvek će postojati potreba da rešimo škakljive sistemske probleme u našim bazama koda. Ako mislimo da smo došli do tačke da mislimo da je degradacija previše opterećujuća i sprečava naš inženjerski tim da se razvija što bolje, onda moramo da se zamislimo i da shvatimo i zašto i kako smo došli u takvu situaciju.

Kada naučimo da vidimo dalje od neposrednih problema koda i umesto toga pokušamo da razumemo okolnosti pod kojima je izvorno napisan, počinjemo da uviđamo da kôd u suštini nije loš. Gradimo empatiju i koristimo ovu novootkrivenu perspektivu da bismo identifikovali istinske temeljne probleme koda i izradili plan za njegovo poboljšanje na najbolji mogući način. Zamislite ovaj proces kao samo jednu veliku vežbu u arheologiji koda!

Sada kada smo naučili kako kôd propada, moramo da naučimo kako da ga pravilno kvantifikujem kako bi ga drugi razumeli. Moramo iskoristiti našu slutnju da je degradacija došla do kritične tačke, svoje znanje o tome zašto i kako je došlo do toga, da bismo pronašli najbolji način da se problem pretoči u skup metrika koje možemo koristiti da ubedimo druge da je u pitanju ozbiljan problem. Sledeće poglavlje govori o brojnim tehnikama koje možete koristiti za merenje problema u bazi koda i uspostavljanje solidne osnove za vaše napore refaktorisanja.