Robert C. Martin Series

Čista arhitektura

Stručni vodič za strukturu i dizajn softvera

Robert C. Martin

prilozi: James Grenning i Simon Brown

kompjuter biblioteka

Čista arhitektura

Stručni vodič za strukturu i dizajn softvera

Robert C. Martin

kompjuter
biblioteka

Izdavač:

kompjuter biblioteka

Obalskih radnika 4a, Beograd

Tel: 011/2520272

e-mail: kombib@gmail.com

internet: www.kombib.rs

Urednik: Mihailo J. Šolajić

Za izdavača, direktor:
Mihailo J. Šolajić

Autor: Robert C. Martin

Prevod: Đorđe Badža

Lektura: Nemanja Lukić

Slog: Zvonko Aleksić

Znak Kompjuter biblioteke:
Miloš Milosavljević

Štampa: „Pekograf“, Zemun

Tiraž: 500

Godina izdanja: 2020.

Broj knjige: 534

Izdanje: Prvo

ISBN: 978-86-7310-557-4

Clean Architecture

Fifth Edition

Robert C. Martin

ISBN 978-0-13-449416-6

Kratak sadržaj

Sadržaj

DEO II

POGLAVLJE 3

POGLAVLJE 4

POGLAVLJE 5

POGLAVLJE 6

DEO IV

POGLAVLJE 12

POGLAVLJE 13

POGLAVLJE 14

DEO VII

POGLAVLJE 15

POGLAVLJE 16

POGLAVLJE 17

POGLAVLJE 18

POGLAVLJE 19

POGLAVLJE 20

POGLAVLJE 21

POGLAVLJE 22

POGLAVLJE 23

POGLAVLJE 24

POGLAVLJE 25

POGLAVLJE 26

POGLAVLJE 27

Predgovor

O čemu govorimo kada razgovaramo o arhitekturi?

Baš kao i kod svake metafore, opis softvera sa stanovišta arhitekture može sakriti onoliko koliko može i pokazati; može obećati više nego što može dati, ali i dati više nego što obećava.

Očigledna privlačnost arhitekture je njena struktura, a struktura je ono što dominira paradigmama i raspravama o razvoju softvera - komponente, klase, funkcije, moduli, slojevi i servisi, mikro ili makro. Ali makrostruktura mnogih softverskih sistema često prkosi svakom verovanju ili razumevanju - organizacija sovjetskih preduzeća, neverovatne Jenga kule koje sežu do oblaka, arheološki slojevi zatrpani pod velikim odronima blata. Struktura softvera nije uvek tako očigledna, kao što je to građevinska struktura.

Zgrade imaju očiglednu fizičku strukturu, bez obzira da li su izgrađene od kamena ili betona, bilo da su visoko zasvođene, ili da zauzimaju veliki prostor, bilo velike ili male, velelepne ili obične. Njihova struktura se ne razlikuje mnogo - to je u velikoj meri posledica zakona gravitacije i fizičkih osobina materijala od kojih su izgrađene. S druge strane - osim u smislu ozbiljnosti - softver ima malo vremena za gravitaciju. A od čega je napravljen? Za razliku od zgrada, koje mogu biti izgrađene od cigle, betona, drveta, čelika i stakla, softver je napravljen od softvera. Velike softverske konstrukcije su napravljene od manjih softverskih komponenti koje su, pak, napravljene od manjih softverskih komponenti i dalje, i tako dalje, sve do kraja.

Kada govorimo o softverskoj arhitekturi, možemo reći da je softver rekurzivne i fraktalne prirode, zapisan i skiciran u kodu. Ovde su važni svi detalji. Preplitanje nivoa detalja takođe doprinosi arhitekturi zgrade, ali je besmisleno razgovarati o softveru u fizičkim merama. Softver ima strukturu - mnoge strukture i mnoge vrste struktura - a njegova raznolikost zasenjuje raspon fizičkih struktura koje se mogu videti u zgradama. Možete, čak, prilično argumentovano da tvrdite da je arhitektura mnogo više naglašena u dizajnu softvera nego u dizajnu zgrade - u tom smislu, nije nerazumno smatrati da je arhitektura softvera više arhitektonska nego arhitekture zgrada!

Ali fizička mera je nešto što ljudi razumeju i traže u svetu. Iako su privlačni i vizuelno očigledni, okviri na PowerPoint dijagramu nisu arhitektura softverskog sistema. Nema sumnje da predstavljaju poseban pogled na arhitekturu, ali je pogrešno zameniti okvire velikom slikom ili konceptom arhitekture, znači promašiti i koncept i arhitekturu: arhitektura softvera ne liči ni na šta drugo. Konkretna vizuelizacija nije ništa drugo nego lični izbor. Ovaj izbor je zasnovan na sledećem skupu opcija: šta uključiti; šta isključiti; šta naglasiti oblikom ili bojom; šta ignorisati radi sličnosti ili propusta. Ne postoji ništa prirodno ili suštinsko što bi dalo prednost jednom pogledu u odnosu na drugi.

Iako, možda, nema smisla razgovarati o zakonima fizike i fizičkim merama u arhitekturi softvera, razmatramo određena fizička ograničenja. Brzina procesora i propusnost mreže mogu grubo presuditi performansu sistema. Memorija i prostor na disku mogu ograničiti ambicije bilo koje baze programskog koda. Softver se može uporediti sa materijom poput snova, ali i dalje radi u fizičkom svetu.

> *To je, gospo, ono ono što je čudovišno u ljubavi; volja je beskrajna a moć ostvarenja ograničena; želja je bezgranična, a delo je ropski omeđeno granicama*[1].
>
> William Shakespeare

1 William Shakespeare, Troilus and Cressida, prevod Živojin Simić i Sima Pandurović [prim.prev.].

Fizički svet je svet u kom živimo, u kom se nalaze naše kompanije i naše ekonomije. To nam daje još jedan, drugačiji pristup razumevanju arhitekture softvera, omogućavaju nam da govorimo i rasuđujemo, ne u smislu fizičkih zakona i koncepata.

Arhitektura predstavlja značajne dizajnerske odluke koje oblikuju sistem, pri čemu se važnost meri prema troškovima promene.

Grady Booch

Vreme, novac i rad - ovo je još jedan referentni okvir koji nam pomaže da razlikujemo veliko i malo, da razlikujemo arhitekturu od svega ostalog. Takođe, ova mera nam pomaže da možemo da utvrdimo da li je arhitektura dobra ili ne: Dobra arhitektura ne samo da zadovoljava potrebe svojih korisnika, programera i vlasnika u datom trenutku, već ih vremenom i ispunjava.

Ako mislite da je dobra arhitektura skupa, isprobajte lošu.

Brian Foote i Joseph Yoder

Tipične promene koje se dešavaju tokom razvoja sistema ne bi trebalo da budu skupe, teške za sprovođenje i da zahtevaju formiranje posebnih projekata; one bi trebalo da se uklapaju u raspored razvoja projekta i da budu unutar dnevnih ili nedeljnih zadataka.

Ta nas direktno vodi do, ne tako malog, problema vezanog za fiziku: putovanja kroz vreme. Kako da znamo koje će se tipične promene dogoditi kako bismo, na osnovu ovog znanja, mogli doneti te značajne odluke? Kako da smanjimo troškove rada i troškove razvoja u budućnosti bez kristalnih kugli i vremenskih mašina?

Arhitektura je skup ispravnih odluka koje želite da donesete odmah, na početku projekta, ali koje nisu, nužno, verovatnije od drugih.

Ralph Johnson

Razumevanje prošlosti je dovoljno teško; naše shvatanje sadašnjosti je, u najboljem slučaju, promenljivo; predviđanje budućnosti nije trivijalno.

Tu je raskrsnica i put se račva na mnogo strana.

Najmračnijim putem dolazi ideja da snažna i stabilna arhitektura zavisi od autoriteta i krutosti. Ako se pokaže da je promena skupa, promena se odbacuje - njeni se uzroci poništavaju svojevoljnom odlukom. Mandat arhitekte je totalan i totalitaran, a arhitektura postaje distopija za svoje programere i stalni izvor frustracije za sve.

Drugi put odaje snažan miris spekulativne zajednice. Pun je ručno kodiranih nagađanja, bezbrojnih parametara, grobnica mrtvih kodova i mnoštva slučajnih složenosti koje ga čekaju, što može poljuljati budžet dodeljen za održavanje.

Put koji nas najviše zanima je najčistiji. Uzima u obzir svojstvenu fleksibilnost softvera i nastoji da je sačuva kao osnovno svojstvo sistema. Uzima u obzir da radimo s nepotpunim znanjem, ali isto tako razume da, budući da smo ljudi, tome smo se dobro prilagodili. On više igra na naše snage, nego na naše slabosti. Stvaramo nešto i otkrivamo. Postavljamo pitanja i izvodimo eksperimente. Dobra arhitektura se zasniva na razumevanju cilja kao kontinuiranog procesa istraživanja, a ne na razumevanju samog cilja kao fiksnog artefakta.

Arhitektura je hipoteza koju bi trebalo dokazati implementacijom i merenjem.

Tom Gilb

Da biste išli ovim putem, trebalo bi da budete vredni i pažljivi, da budete u stanju da razmišljate i posmatrate, da steknete praktične veštine i savladate principe. U početku će vam se činiti da je to dug put, ali zaista sve zavisi od tempa vašeg hoda.

Jedini način da brzo idete je da idete dobro.

Robert C. Martin

Uživajte u putovanju.

Kevlin Henney

Maj 2017

Uvod

Naslov ove knjige je *Čista arhitektura*. Smeo naslov. Neki će ga smatrati arogantnim. Pa, zašto sam odlučio da napišem ovu knjigu i odabrao ovaj naslov?

Svoju prvu liniju koda sam napisao 1964. godine, sa 12 godina. Sada je na kalendaru 2016. godina, tako da pišem kod više od pola veka. Za to vreme sam naučio ponešto o strukturiraju softverskih sistema, a moje bi znanje, čini mi se, mnogi smatrali dragocenim.

Ovo znanje sam stekao gradeći mnoge sisteme, velike i male. Napravio sam male ugrađene sisteme i velike sisteme za beč obradu, sisteme koji rade u realnom vremenu i veb sisteme, konzole, GUI aplikacije, aplikacije za upravljanje procesima, igre, računovodstvene sisteme, telekomunikacione sisteme, alate za dizajn, aplikacije za obradu grafike i još mnogo toga.

Napisao sam jednonitne aplikacije, višenitne aplikacije, aplikacije koje obrađuju više teških procesa, aplikacije koje obrađuju mnogo laganih procesa, višeprocesorske aplikacije, aplikacije baze podataka, aplikacije za matematiku i računsku geometriju i mnoge, mnoge druge.

Napisao sam mnogo aplikacija. Izgradio sam mnogo sistema. I zahvaljujući stečenom iskustvu, naučio sam nešto neverovatno.

Arhitektonska pravila su ista!

To je zapanjujuće, jer su se svi sistemi koje sam izgradio radikalno razlikovali jedni od drugih. Zašto bi svi tako različiti sistemi trebalo da dele ista pravila arhitekture? Došao sam do zaključka da *su pravila za kreiranje arhitekture softvera nezavisna od bilo koje druge promenljive*.

Ovaj zaključak još više iznenađuje kada uzmete u obzir promenu koja se dogodila u hardveru tokom proteklih pola veka. Počeo sam da programiram na mašinama veličine kuhinjskih frižidera čiji su procesori radili na frekvenciji pola megaherca, 4K centralne memorije, 32K memorije na disku i terminalski interfejs od 10 znakova u sekundi. Ovaj predgovor sam napisao u autobusu tokom turneje po Južnoj Africi. Koristim MacBook koji ima procesor i7 sa četiri jezgra, svako na taktu 2,8 GHz, 16 GB RAM-a, terabajt SSD-a i 2880*1800 retina ekran koji je u stanju da prikaže video zapis visokog kvaliteta. Razlika u procesorskoj snazi je zapanjujuća. Svaka razumna analiza će pokazati da je ovaj MacBook barem 1022 puta moćniji od onih ranih računara koje sam počeo da koristim pre pola veka.

Dvadeset i dva reda veličine je vrlo velik broj. To je broj angstrema[2] od Zemlje do zvezde Alpha-Centuri. To je broj elektrona u kovanicama u džepu ili torbici. Pa ipak taj broj, takođe, opisuje koliko se puta (*barem*) povećala snaga računara, koju iskušavam u svom životnom veku.

Uz svu tu ogromnu promenu u računarskoj snazi, kakav je to uticaj imalo na softver koji pišem? Softver je definitivno postao veći. Nekad sam mislio da je 2000 linija veliki program. Na kraju krajeva, takav program bi zauzeo punu kutiju bušenih kartica tešku 4.5 kilograma. Međutim, sada se program smatra stvarno velikim samo ako količina koda premaši 100.000 linija.

2 **Angstrem** (simbol Å) je jedinica za dužinu. Imenovana je po švedskom fizičaru Andersu Jonasu Ångströmu, jednom od osnivača spektroskopije. 1 angstrem iznosi 10^{-10} metara (odnosno 0,1 nanometara). Angstrem **nije SI jedinica**, stoga se ne preporučuje njegova upotreba. U prošlosti se upotrebljavao za izražavanje veličine atoma, dužine hemijskih veza i talasnih dužina svetla i ultravioletnogog zračenja. [op.prev.]

Softver je, takođe, postao znatno produktivniji. Danas možemo brzo da izvršimo proračune o kojima šezdesetih godina prošlog veka nismo mogli ni da sanjamo. *The Forbin Project*, *The Moon Is a Harsh Mistress* i *2001: A Space Odyssey,* tako su pokušali da zamisle našu neposrednu budućnost, ali su ovi autori promašili cilj. Svi su zamišljali ogromne mašine koje su stekle moć osećanja. Ono što imamo, umesto toga, su neverovatno male mašine koje su još uvek ... samo mašine.

Postoji još jedna bitna sličnost između modernog softvera i softvera koji smo imali nekad: *Napravljen je od istih elemenata.* Napravljen je od if naredbi, instrukcija dodeljivanja i while petlji.

O da, možete tvrditi da imamo puno bolje programske jezike i superiornije paradigme. Na kraju krajeva, programiramo u Javi, C#-u ili Ruby-ju i koristimo objektno orijentisani dizajn. Tačno - a ipak se kod još uvek sastoji od skupa sekvenci operacija, uslovnih instrukcija i iteracija, baš kao u 1960-im i 1950-im godinama.

Kada pažljivo pogledate praksu programiranja računara, primetićete da se za 50 godina malo toga promenilo. Jezici su postali malo bolji. Alati su postali fantastično bolji. Ali, osnovni gradivni blokovi računarskog programa su ostali isti.

Ako uzmete programera iz 1966. godine, premestite ga[3] u 2016. godinu i stavite ga ispred mog MacBooka koji koristi IntelliJ, te mu pokažete programski jezik Java, možda bi mu trebalo 24 časa da se oporavi od šoka. A onda bi mogao da piše savremene programe. Jezik Java se ne razlikuje mnogo od jezika C, ili čak Fortrana.

Ako bih vas sad vratio nazad u 1966. godinu i pokazao vam kako pisati i uređivati PDP-8 kod bušenjem papirne kartice na terminalu, brzinom od 10 znakova u sekundi, možda bi vam trebalo ceo dan da se oporavite od razočarenja. Ali biste onda mogli da napišete i kod. Sam kod se ne bi mnogo promenio.

To je cela tajna: ova nepromenljivost programskih principa je razlog što su pravila arhitekture softvera toliko dosledna za sve tipove sistema. Pravila arhitekture softvera su pravila redosleda i montaže gradivnih blokova programa. A budući da su ti blokovi univerzalni i nisu se menjali, pravila za njihovo uređivanje su, takođe, univerzalna i nepromenjiva.

3 Upravo "ona", jer su tih godina programeri uglavnom bile žene.

Programeri početnici mogu pomisliti da su sve to gluposti. Mogli bi insistirati na tome da je danas sve bolje i drugačije, da su pravila prošlosti prevaziđena i nestala. Ako je to ono što oni misle, oni se, nažalost, varaju. Pravila se nisu promenila. Uprkos pojavi novih jezika, okvira i paradigmi, pravila ostaju ista kao i onda kada je Alan Turing napisao prvi mašinski kod 1946. godine.

Ali jedno se promenilo: Tada, u prošlosti, nismo znali koja su to pravila. Zbog toga smo ih kršili, iznova i iznova. Sada, sa pola veka iskustva iza sebe, mi znamo i razumemo ta pravila.

Upravo o ovim pravilima - koja su bezvremenska i nepromjenjiva - govori ova knjiga.

Zahvalnice

U nastavku su navedeni, bez nekog određenog redosleda, svi ljudi koji su bili ključni za stvaranje ove knjige:

Chris Guzikowski

Chris Zahn

Matt Heuser

Jeff Overbey

Micah Martin

Justin Martin

Carl Hickman

James Grenning

Simon Brown

Kevlin Henney

Jason Gorman

Doug Bradbury

Colin Jones

Grady Booch

Kent Beck

Martin Fowler

Alistair Cockburn

James O. Coplien

Tim Conrad

Richard Lloyd

Ken Finder

Kris Iyer (CK)

Mike Carew

Jerry Fitzpatrick

Jim Newkirk

Ed Thelen

Joe Mabel

Bill Degnan

I mnogi drugi, previše brojni da bi svi bili imenovani.

U poslednjoj reviziji ove knjige, dok sam čitao poglavlje o Arhitekturi koja vrišti, kao da sam video nasmejani lik Jim-a Weirich-a, a u ušima mi je odjekivao njegov zvonki smeh. Srećno Jim!

O autoru

Robert C. Martin, poznat kao "Ujka-Bob", profesionalno se bavi programiranjem od 1970. godine. Suosnivač je cleancoders.com-a, koji nudi online video trening za programere softvera. Osnivač je kompanije Uncle Bob Consulting LLC, koja velikim korporacijama širom sveta nudi softversko savetovanje, obuku i usluge razvoja veština. Služio je kao predradnik u 8th Light, Inc., firmi za savetovanje o softveru sa sedištem u Čikagu. Objavio je desetine članaka, u raznim stručnim časopisima i redovan je govornik na međunarodnim konferencijama i sajmovima. Tri godine je bio glavni urednik časopisa *C++ Report* i prvi predsednik Agile Alliance.

Martin je autor i urednik mnogih knjiga, uključujući *The Clean Coder*, *Clean Code*, *UML for Java Programmers*, *Agile Software Development*, *Extreme Programming in Practice*, *More C++ Gems*, *Pattern Languages of Program Design 3*, and *Designing Object Oriented C++ Applications Using the Booch Method*.

Postanite član Kompjuter biblioteke

Kupovinom jedne naše knjige stekli ste pravo da postanete član Kompjuter biblioteke. Kao član možete da kupujete knjige u pretplati sa 40% popusta i učestvujete u akcijama kada ostvarujete popuste na sva naša izdanja. Potrebno je samo da se prijavite preko formulara na našem sajtu. Link za prijavu: http://bit.ly/2TxekSa

Skenirajte QR kod
registrujte knjigu
i osvojite nagradu

Uvod

Nije vam potrebno mnogo veštine i znanja da napišete program koji radi. Deca to stalno rade. Mladići i devojke na fakultetima počinju milionske poslove sa nekoliko linija PHP ili Ruby koda. Mladi programeri, u kancelarijama širom sveta, prebiraju planine dokumentovanih zahteva, koje nalaze u džinovskim sistemima za praćenje grešaka, i popravljaju ih da bi njihovi sistemi "radili". Kod koji oni pišu možda nije najbolji, ali uspeva. Uspeva, jer naterati nešto da radi - jednom - jeste jednostavno.

Sasvim je druga stvar napisati program koji radi ispravno. Pisanje ispravnog programa jeste teško. Za to je potrebno znanje i veštine koje većina mladih programera tek treba da stekne. To zahteva razmišljanje i analizu, za šta mnogi programeri, jednostavno, nemaju vremena. To zahteva određeni stepen samodiscipline i organizacije o kojima većina programera ni ne sanja da im je potrebna. Za to bi trebalo da imate strast prema profesiji i želju da postanete profesionalac.

Međutim, kada napišete ispravan kod događa se ono magično: nije vam potrebna gomila programera da ga održite u radu. Nema potrebe za obimnom dokumentacijom zahteva, ni za ogromnim sistemima za praćenje grešaka. Ne trebaju vam ogromni prostori podeljeni na male, odvojene radne delove; ne treba vam rad programera 24/7.

Ispravan programski kod zahteva samo deo ljudskih resursa za kreiranje i održavanje. Promene su jednostavne i lako se izvršavaju. Greške su retke. Napor je sveden na minimum, a funkcionalnost i fleksibilnost su dovedeni do maksimuma.

Da, ova vizija zvuči, pomalo, utopijski, ali ja sam je video rođenim očima. Učestvovao sam u projektima čiji su dizajn i arhitektura olakšavali kreiranje i održavanje. Imam iskustva u radu na projektima za koje je bilo potrebno manje od predviđenog broja saradnika. Radio sam na sistemima u kojima su greške ekstremno retke. Video sam neverovatan efekat koji dobra arhitektura ima na sistem, projekat i razvojni tim. Bio sam u obećanoj zemlji.

Ali, ne verujte mi na reč. Osvrnite se na lično iskustvo. Da li ste nekad iskusili suprotno? Da li ste radili na sistemima toliko zamršenim i isprepletanim da bilo kakva promena, ma kako trivijalna, trajala nedeljama i bila opterećena velikim rizicima? Da li ste iskusili otpor lošeg koda i rđavog dizajna? Da li je dizajn sistema u kom ste radili imao negativan uticaj na moral tima, poverenje kupaca i strpljenje rukovodstva? Da li ste nekad bili u situaciji da vidite kako timovi, odeljenja i čitave kompanije postaju žrtve rđave strukture svog softvera? Da li ste bili u programerskom paklu?

Ja jesam - kao i mnogi drugi. Borba sa lošim softverskim dizajnom je mnogo češće iskustvo, nego uživanje u primeni onog koji je dobro osmišljen.

1 Šta su dizajn i arhitektura?

Tokom godina je bilo mnogo konfuzije oko koncepata "dizajna" i "arhitekture". Šta je dizajn? Šta je arhitektura? U čemu se razlikuju?

Jedan od ciljeva ove knjige je da eliminiše svu tu zabunu i jednom za svagda definiše: šta su dizajn i arhitektura. Za početak, ustvrdio bih da među njima nema razlike. *Ni najmanje*.

Reč "arhitektura" se često koristi u kontekstu rezonovanja na višem nivou, kao nešto potpuno odvojeno od detalja nižeg nivoa, dok reč "dizajn" obično podrazumeva strukture i odluke na nižem nivou. Međutim, takva podela je besmislena kada se pogleda šta pravi arhitekta radi.

Uzmimo za primer arhitektu koji je dizajnirao moju novu kuću. Da li ova kuća ima arhitekturu? Naravno da ima. A kako se ona izražava? Pa, to je oblik kuće, spoljašnji izgled, elevacije, kao i organizacija unutrašnjeg prostora i raspored prostorija. Međutim, dok sam gledao nacrte koje moj arhitekta kreirao, video sam na njima ogroman broj detalja. Video sam gde će biti sve utičnice, svi prekidači i svetla. Video sam koji će prekidač biti za koje svetlo. Video sam gde će biti grejna jedinica, kao i pozicije i dimenzije kotla za grejanje vode i pumpe za vodu. Video sam detaljan opis kako će biti izvedeni zidovi, krov i temelj.

Jednostavno rečeno, video sam sve sitne detalje koji podržavaju sve strukture visokog nivoa. Takođe, video sam da detalji niskog i strukture visokog nivoa, zajedno, čine ukupan dizajn kuće.

Isto se odnosi i na softversku arhitekturu. Detalji niskog i struktura visokog nivoa deo su iste celine. Oni čine beskrajno tkanje koje definiše oblik sistema. Jedno ne ide bez drugog; zaista, ne postoji jasna linija koja ih razdvaja. Jednostavno, postoji kontinuum odluka od najvišeg do najnižeg nivoa.

Cilj?

Šta je cilj tih odluka, cilj dobro dizajniranog softvera? Taj cilj nije ništa drugo nego moj utopijski opis:

> *Cilj softverske arhitekture je minimalizovanje ljudskih resursa potrebnih za izradu i održavanje željenog sistema.*

Mera kvaliteta dizajna je, jednostavno, mera napora koji je potrebno uložiti da se zadovolje potrebe kupaca. Ako je taj napor mali, i ostaje mali tokom veka trajanja sistema, dizajn je dobar. Ako se, svakom novom verzijom, napor povećava, sistem je loše dizajniran. Toliko je jednostavno.

Studija slučaja

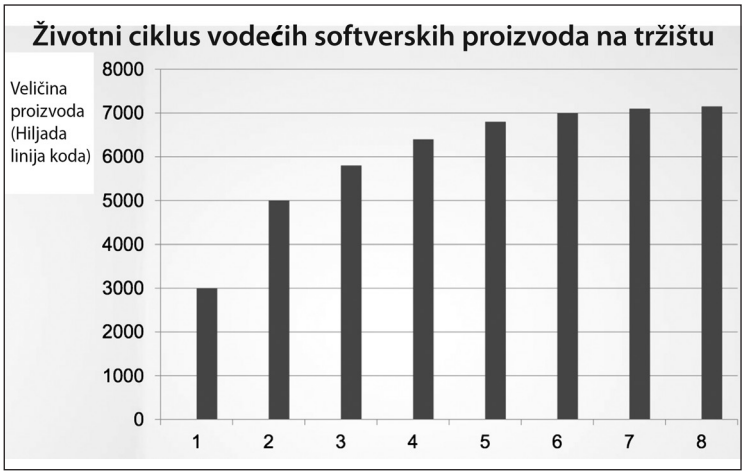
Kao primer, uzmite u obzir studiju slučaja, zasnovanu na stvarnim podacima koje je obezbedila stvarna kompanija, koja želi da ostane anonimna.

Prvo, razmotrite grafikon rasta broja inženjerskog i tehničkog osoblja. Verovatno ćete se složiti da je trend impresivan. Rast poput ovog prikazanog na slici 1.1 mora biti pokazatelj uspešnog razvoja preduzeća!

Slika 1.1 Rast inženjerskog osoblja

Reprodukovano iz prezentacije slajdova Jason-a Gorman-a, uz dozvolu

Pogledajmo sada produktivnost kompanije u istom vremenskom periodu, mereno linijama koda (slika 1.2).



Slika 1.2 Produktivnost u istom vremenskom periodu

Jasno je da je nešto krenulo po zlu. Iako svaku verziju podržava sve veći broj programera, čini se da se broj redova koda približava svojoj granici.

Sada, pogledajte zaista depresivan grafikon: grafikon na Slici 1.3 prikazuje kako vremenom raste cena po liniji koda.

Ovi trendovi su neodrživi. Nije važno koliko je kompanija trenutno profitabilna, porast opštih troškova će katastrofalno iscrpeti dobit iz poslovnog modela i odvesti kompaniju u stagnaciju, ako ne i u potpunu propast.

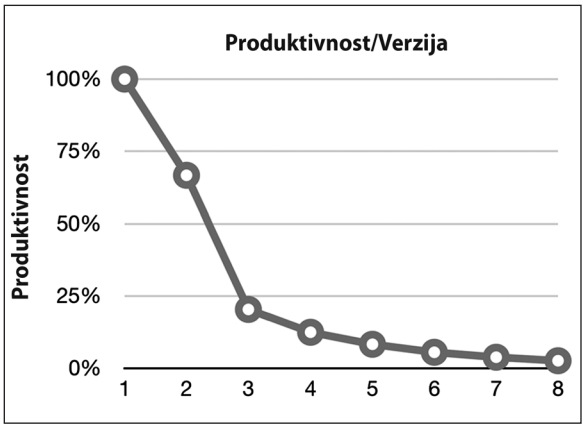
Šta je izazvalo ovu izvanrednu promenu produktivnosti? Zašto je linija koda 8. izdanja 40 puta skuplja od linije koda 1. izdanja?

Slika 1.3 Trošak po liniji koda tokom vremena

Uzrok nevolja

Uzrok nevolja vam je pred očima. Kada se sistemi grade na brzinu, kada je povećavanje broja programera jedini način da nastave izdavati nove verzije i kada se malo ili nimalo pažnje posvećuje čistoći koda ili strukturi dizajna, možete biti sigurni da će takav trend pre ili kasnije dostići ružan kraj.

Slika 1.4 prikazuje kako ova krivulja izgleda programerima. Počeli su sa gotovo 100% produktivnosti, ali je svakom novom verzijom njihova produktivnost opadala. Od četvrte verzije bilo je jasno da se njihova produktivnost približava donjoj granici - nuli.



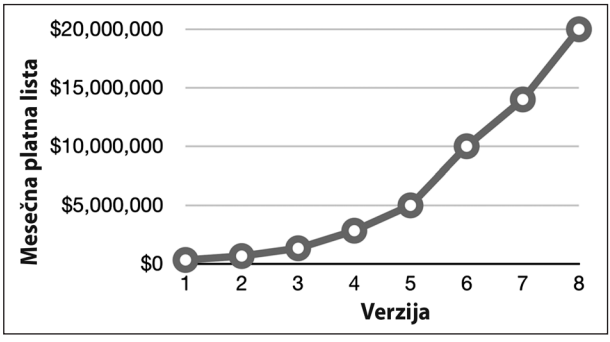
Slika 1.4 Produktivnost po verzijama

Iz perspektive programera ova situacija je izuzetno uznemirujuća, jer oni rade kao i do sada s *punom predanošću*. Niko ne zazire od posla.

Pa ipak, uprkos tom njihovom junaštvu, prekovremenom radu i posvećenosti, oni jednostavno ne mogu da postignu više. Njihovi napori nisu više usmereni na implementiranje novih funkcija, već na borbu protiv nereda. Njihov posao, takav kakav je, pretvorio se u premeštanje nereda s jednog mesta na drugo, pa na sledeće i na sledeće, toliko da mogu tek s vremena na vreme da dodaju po neku oskudnu karakteristiku.

Gledište rukovodstva

Ako mislite *da je to loše*, zamislite kako ova slika izgleda rukovodiocima! Slika 1.5 prikazuje grafikon promena mesečne zarade programera za isti period.



Slika 1.5 Mesečni obračun zarada prema verzijama

Kada je objavljena verzija 1, mesečna zarada programera je iznosila nekoliko stotina hiljada dolara. Do objavljivanja druge verzije, fond se povećao za još nekoliko stotina hiljada. Do izlaska osme verzije, mesečni obračun zarada programera je iznosio 20 miliona dolara, a trend rasta se nastavljao.

Samo ovaj grafikon je zastrašujući. Očigledno je da se dešava nešto strašno. Jedina nada je da rast prihoda nadmaši rast troškova, pa ih, samim tim, i opravda. Međutim, sa bilo koje tačke gledišta, ova kriva je zabrinjavajuća.

Sada uporedite krivu na slici 1.5 sa grafikonom rasta broja linija koda od verzije do verzije, na slici 1.2. Za početnih nekoliko stotina hiljada dolara mesečno implementirana je ogromna količina funkcionalnosti, a za 20 miliona skoro ništa nije dodato najnovijoj verziji! Svaki finansijski direktor, koji bi pogledao ova dva grafikona, znao bi da je za sprečavanje katastrofe neophodno hitno delovanje.

Ali šta se može učiniti? Šta nije u redu? Šta je izazvalo ovaj neverovatni pad produktivnosti? Šta menadžeri mogu učiniti, osim da lupaju šakom o sto i iskaljuju bes na programerima?

Šta je pošlo po zlu?

Pre gotovo 2600 godina, Ezop je napisao basnu Kornjača i zec. Pouka te basne se može izraziti na različite načine:

- „Strpljivi i uporni stižu do cilja."
- „Najbrži ne pobedi uvek u trci, a najjači ne pobedi uvek u bici"
- „Što više žurite, to više kasnite."

Sama priča ilustruje glupost oholosti: zec, siguran u svoju brzinu nije se ozbiljno shvatio trku, čak je prespavao kornjačin prolazak kroz cilj.

Savremeni programeri su, takođe, u sličnoj trci i pokazuju, slično, prekomerno samopouzdanje. Oh, ne spavaju oni - daleko od toga. Većina modernih programera, zaista, naporno radi. Međutim, deo njihovog mozga, zaista, spava - onaj deo koji zna da dobar, čist, dobro dizajniran kod igra *ključnu ulogu*.

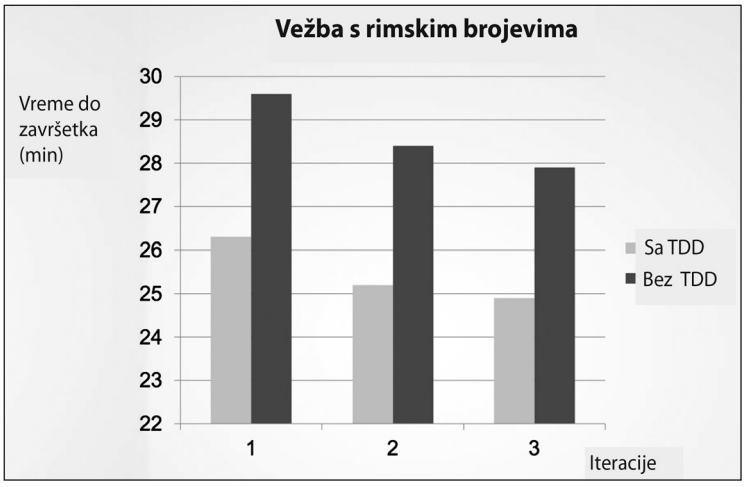
Ovi programeri veruju u dobro poznatu laž: „Kasnije možemo stvari dovesti u red, samo da prvo izađemo na tržište!" Naravno da red nikada ne bude uspostavljen, jer pritisci konkurencije na tržištu nikada ne popuštaju. Izlazak na tržište znači da sada imate konkurenciju za vratom i da se morate truditi da ostanete ispred njih, trčeći iz sve snage.

Tako da programeri nikad ne menjaju način rada. Ne mogu da se vrate da raspreme nered, jer moraju da kreiraju novu funkcionalnost, pa još jednu, i još jednu, i još jednu. Tako se nered gomila, a produktivnost teži donjoj granici, blizu nule.

Kao što je Zec bio previše siguran u svoju brzinu, tako su i mnogi programeri previše uvereni u da će ostati produktivni. Međutim, puzeći nered koda koji umanjuje njihovu produktivnost nikada ne spava i nikada ne popušta; ako mu se ukaže prilika, spustiće produktivnost na nulu u kratkom roku.

Najveća laž u koju veruju mnogi programeri je da će im neuredni kod pomoći da brzo izađu na tržište, ali u stvarnosti će ih dugoročno usporiti. Programeri koji veruju u ovu laž pokazuju aroganciju zeca, veruju da će u budućnosti moći da pređu sa stvaranja nereda na sređivanje stvari, ali čine jednostavnu grešku. Činjenica je da je *stvaranje nereda uvek sporije od održavanja reda*, bez obzira na izabrani vremenski okvir.

Razmotrite rezultate izvanrednog eksperimenta prikazanog na slici 1.6, koji je izveo Jason Gorman. Jason je ovaj test sprovodio u periodu od šest dana. Svakog bi dana napisao jednostavan program za pretvaranje celih brojeva u rimske brojeve. Znao bi da je posao završen kad bi njegov program prošao unapred definisan skup testova prihvatanja. Svakog dana mu je trebalo nešto manje od 30 minuta da reši problem. Prvog, drugog i trećeg dana Jason je koristio poznatu metodologiju zasnovanu na testiranju (TDD). Drugim danima je pisao kod ne ograničavajući se opsegom ove tehnike.



Slika 1.6 Vreme do završetka pomoću iteracija i korišćenja ili ne korišćenja TDD

Pre svega, uočite krivu učenja koja je očigledna na slici 1.6. Svaki put se manje vremena trošilo na rešavanje problema. Takođe, primetite da je vežba izvedena oko 10% brže u TDD danima, nego u danima koji nisu TDD i da je, čak, najgori rezultat dobijen pomoću TDD-a bio bolji od najboljeg rezultata bez TDD-a.

Neki ljudi mogu pogledati taj rezultat i pomisliti da je to izvanredan ishod, ali onima koji nisu zavedeni arogancijom zeca rezultat će biti očekivan, jer znaju ovu jednostavnu istinu razvoja softvera:

> *Jedini način da idete brzo je da idete kako treba.*

To je, takođe, odgovor na dilemu menadžmenta. Jedini način da se preokrene pad produktivnosti i rast troškova jeste da se programeri navedu da prestanu da razmišljaju kao arogantni zec i da preuzimaju odgovornost za nered koji prave.

Programeri mogu misliti da se problem može rešiti samo od početka i redizajniranjem sistema u celini, ali ovo je i dalje ono što Zec govori. Ista bahatost koja je ranije vodila do zabune sada im, ponovo, govori da mogu da izgrade bolji sistem, samo ako iznova uđu u trku. Međutim, stvarnost nije tako ružičasta:

> *Njihovo prekomerno samopouzdanje odvešće proces redizajniranja u isti nered u kom je i originalna verzija.*

Zaključak

U svakom slučaju, najbolja opcija je da razvojna organizacija prepozna i izbegne sopstveno prekomerno samopouzdanje i da počne ozbiljno da shvata kvalitet arhitekture svog softvera.

Da biste ozbiljno shvatili arhitekturu softvera, morate znati šta je to dobra arhitektura softvera. Da biste kreirali sistem čiji će dizajn i arhitektura smanjiti troškove rada i povećati produktivnost, morate znati koji arhitektonski elementi dovode do toga.

O tome govori ova knjiga. Objašnjava kako izgledaju čista arhitektura i dizajn, tako da programeri mogu da izgrade sisteme koji će imati dug i profitabilan radni vek.

2 Priča o dve vrednosti

Svaki softverski sistem obezbeđuje zainteresovnim stranama dve različite vrednosti: ponašanje i strukturu. Programeri softvera su odgovorni da održavaju ove dve vrednosti na visokom nivou. Oni se, na žalost, često fokusiraju na jedno, a zaboravljaju drugo. Još gore, oni se često fokusiraju na sekundarne delove ove dve vrednosti, a to na kraju obezvredi sistem.

Ponašanje

Prva vrednost softvera je njegovo ponašanje. Programeri su angažovani da učine da mašine rade tako da štede novac, ili da donose profit zainteresovanim stranama. Da bismo to uradili, pomažemo zainteresovanim stranama da razviju funkcionalne specifikacije, ili dokument sa zahtevima. Tada pišemo kod koji će učiniti da računari zainteresovane strane ispune date zahteve.

Kada računar narušava zahteve, programeri se maše programa za otklanjanje grešaka i rešavaju problem.

Mnogi programeri veruju da se njihov posao tu završava. Oni veruju da je njihov posao da učine da mašina implementira zahteve i da otklone sve uočene greške. Ljuto se varaju.

Arhitektura

Druga vrednost softvera leži u samom nazivu "softver" - složenici sastavljenoj od reči „soft“ i „ware“. Reč „ware“ znači „proizvod“; a reč „soft“ ... Upravo u njoj leži druga vrednost.

Ideja je da softver bude "mekan" i da olakša promenu ponašanja računara. Da smo želeli da ta promena bude teška, nazvali bismo ga hardver.

Da bi ispunio svoj cilj, softver mora biti prilagodljiv - odnosno, mora biti prilagodljiv. Kada se zainteresovane strane predomisle u vezi sa određenom osobinom, promena te osobine bi trebalo da bude jednostavan zadatak, lak za sprovođenje. Složenost bi u takvim slučajevima trebalo da bude u proporciji jedino sa razmerama promene, a ne sa njenim *oblikom*.

Upravo razlika između opsega i oblika često pokreće rast troškova razvoja softvera. Iz tog razloga trošak raste neproporcionalno veličini potrebnih promena. Zbog toga su troškovi razvoja u prvoj godini niži nego u drugoj, a troškovi u drugoj godini niži nego troškovi u trećoj godini.

Iz perspektive zainteresovane strane, oni jednostavno formiraju niz promena približnog obima. Sa stanovišta programera, zainteresovane strane formiraju tok fragmenata koje bi oni trebalo da uklope u sve složeniji mozaik. Svaki novi zahtev je složeniji od prethodnog, jer se forma sistema ne podudara sa formom zahteva.

Reč „forma“ ovde koristim na nekonvencionalan način, ali mislim da je metafora sasvim prikladna. Programeri softvera se često osećaju kao da su prisiljeni da guraju četvrtaste čepove u okrugle rupe.

Problem je, naravno, arhitektura sistema. Što više arhitektura preferira jednu formu nad drugom, to je verovatnije da će nove karakteristike biti teže uklopiti u tu strukturu. Stoga bi arhitekture trebalo da budu što je moguće više nezavisne od formi.

Veća vrednost

Funkcionalnost ili arhitektura? Šta je vrednije? Šta je važnije - pravilan rad sistema ili jednostavnost njegove promene?

Ako ovo pitanje postavite rukovodiocu preduzeća, on će sigurno odgovoriti da je pravilan rad važniji. Programeri se, često, slažu sa ovim stavom, *ali je to pogrešan stav*. Jednostavnim logičkim alatom za istraživanje ekstrema moguće je dokazati da je ovaj stav pogrešan.

- *Ako mi date program koji savršeno funkcioniše, ali ga je nemoguće promeniti, kada se promene zahtevi program neće raditi, a biće i nemoguće učiniti da radi. Stoga će program postati beskoristan.*
- *Ako mi date program koji ne radi, ali ga je lako promeniti, onda mogu učiniti da radi i da nastavi sa radom kako se zahtevi budu menjali. Stoga će program ostati kontinuirano koristan.*

Možda vam ovi argumenti ne deluju uverljivo. Na kraju krajeva, nema programa koji se ne može menjati. Međutim, postoje sistemi koje je *praktično* nemoguće promeniti, jer bi troškovi promene premašili korist. Mnogi sistemi dostižu ovo stanje u nekim svojim karakteristikama ili konfiguracijama.

Ako pitate poslovne menadžere da li bi želeli da mogu da vrše promene, oni će odgovoriti da bi to svakako želeli, ali će možda prekvalifikovati odgovor napomenom da im je trenutna funkcionalnost važnija od nekakve buduće fleksibilnosti. Nasuprot tome, ako poslovni menadžer zahteva promenu, a vi procenite da bi troškovi te promene bili nesrazmerno visoki, sigurno će vam zameriti što ste sistem doveli do tačke u kojoj troškovi promene prevazilaze isplativost.

Eisenhowerova matrica

Razmislite o matrici važnost naspram hitnosti, američkog predsednika D. D. Eisenhower-a (slika 2.1). O ovoj matrici Eisenhower je rekao:

Imam dve vrste problema, hitne i važne. Hitni nisu važni, a važni nikad nisu hitni[1].

Slika 2.1. Eisenhower-ova matrica

Mnogo je istine u ovoj staroj izreci. Hitno je retko kad zaista važno, a važno je retko kad hitno.

Prva vrednost softvera - ponašanje - hitno, ali ne uvek preterano važno.

Druga vrednost softvera - arhitektura - važna, ali nikad preterano hitna.

Naravno, postoje i zadaci koji su istovremeno i važni i hitni, kao i zadaci koji nisu ni važni ni hitni. Konačno, možemo poređati ova četiri para prema prioritetu:

1. Hitno i važno
2. Nije hitno, jeste važno
3. Hitno a nije važno
4. Nije hitno i nije važno

Imajte na umu da arhitektura koda - važno pitanje - zauzima prva dva mesta ove liste, dok ponašanje koda zauzima prvo i treće mesto.

1 Iz govora na Univerzitetu Northwestern 1954. godine.

Poslovni menadžeri i programeri često prave grešku podižući tačku 3 na nivo tačke 1. Drugim rečima, oni pogrešno odvajaju hitne i nevažne zadatke od zadataka koji su zaista hitni i važni. Ova pogrešna procena dovodi do zanemarivanja važnosti arhitekture sistema i prenaglašavanja nebitnog ponašanja.

Programeri softvera su suočeni sa problemom što poslovni menadžeri nisu sposobni da procnjuju značaj arhitekture. *Za to se angažuju programeri softvera*. Prema tome, odgovornost tima za razvoj softvera je da istakne važnost arhitekture u odnosu na hitnost karakteristika.

Borba za arhitekturu

Ispunjavanje ove odgovornosti podrazumeva ulazak u bitku - možda je „borba" bolja reč. Iskreno, to je način na koji funkcionišemo. Razvojni tim mora da se bori za ono što smatra najboljim za kompaniju, baš kao i menadžerski tim, marketinški tim, tim prodaje, ili operativni tim. *To je uvek borba*.

Efikasni razvojni timovi u ovu borbu ulaze dignuta čela. Oni otvoreno i kao jednaki ulaze u raspravu sa ostalim zainteresovanim stranama. Zapamtite, kao programer softvera i *vi ste zainteresovana strana*. Imate ulog u softveru koji bi trebalo da zaštitite. To je deo vaše uloge i vaš deo odgovornosti. To je i jedan od glavnih razloga za vaš angažman.

Ovaj izazov je dvostruko važniji ako ste arhitekta softvera. Arhitekte su, po prirodi svog posla, usredsređeni na strukturu sistema, a ne na njegove karakteristike i funkcije. Arhitekte kreiraju arhitekturu koja omogućava brži i lakši razvoj, modifikovanje i dopunjavanje ovih karakteristika i funkcija.

Zapamtite: ako je arhitektura na poslednjem mestu, razvoj sistema će biti skuplji, dok na kraju ne bude gotovo nemoguće izvršiti promene tog sistema ili njegovih delova. Ako se to dozvoli, to znači da se tim za razvoj softvera nije dovoljno borio za ono za šta je smatrao da je neophodno.