UVOD

„Softver preuzima svet.” Ova čuvena izjava Marka Andresena je jedan od mojih omiljenih citata, gotovo kao mim, koji otkriva trenutno stanje stvari. Nikada ranije u istoriji čovečanstva nije bilo toliko softvera, niti je toliko stvari bilo pod kontrolom softvera. Za nas koji živimo u gradovima, skoro svime što nas okružuje upravlja softver. Svake godine, sve više kontrole se prenosi na softverske artefakte. A uz eksplozivan nalet veštačke inteligencije, ovaj trend je sve naglašeniji: veštačke inteligencije sa kojima sarađujemo su takođe softveri.

Pripadam prvim generacijama programera, koji tada nisu toliko zavisili od softvera. Sa šesnaest godina sam počeo da pišem manje programe. Sa osamnaest, prešao sam na razvoj većih sistema i tada je začeta motivacija da napišem ovu knjigu, razlog zašto je tako dobrodošla i čak neophodna: softver, alat od kojeg sve više stvari zavisi, pišu ljudi. To je kod. Kvalitet tog koda direktno se odražava na kvalitet softvera, njegovu održivost, trajanje, cenu i performanse...

Zahvaljujući tim ranim sistemima, koje su programirali mali timovi od dva do tri člana, naučio sam, na teži način, zašto je čist kod koristan. Voleo bih da sam tada imao knjigu poput ove, jer bi mi uštedela mnogo vremena.

To ne znači da je značaj ove knjige ostao u tim praistorijskim informatičkim vremenima, niti da je namenjena programerima početnicima da uče osnove. Naprotiv.

U ovoj knjizi ćete, kao recepte u kuvaru, pronaći jednostavne načine da izbegnete brojne zamke i uobičajene greške u kodu. U mnogim slučajevima, sam recept nije najvažniji deo poglavlja, već je fokus na iznošenju i razmatranju određene teme, što daje osnovu za razmišljanje o tome kako rešiti dati problem i kako oceniti čistoću našeg rešenja. Kontijerov stil je izuzetno neposredan i direktan, jednako jasan kao što bismo želeli da naš kod bude. U svakom „receptu” postoje primeri koda kako bismo otklonili svaku sumnju u vezi sa situacijama u kojima ih treba pravilno primeniti.

Može se činiti da su čistoća i jasnoća koda problem i odgovornost programera. Ali u stvarnosti, problemi s kodom počinju mnogo pre, u fazi dizajna, a protežu se do alokacije resursa, politike razvoja, upravljanja projektima i timovima, faza održavanja i evolucije... Verujem da će većina stručnjaka iz softverske industrije imati koristi od ove knjige jer odlično ilustruje i objašnjava veliki broj uobičajenih problema u kodu, materijalu od kojeg se pravi softver... softver koji „guta” svet.

Primamljivo je mišljenje da je pisanje koda stvar prošlosti i da će generativna veštačka inteligencija i veliki jezički modeli preuzeti ulogu proizvodnje koda, bez ljudske intervencije. Na osnovu primera koje viđam svakodnevno, to još uvek nije moguće, zbog količine „halucinacija" (osnovne greške, problemi tumačenja, ranjivosti i poteškoće održavanja) koje se pojavljuju u kodu koji piše veštačka inteligencija. Međutim, jasno je da smo u prelaznoj fazi. Tokom ove faze, tehnološki kentauri će napredovati, s iskusnim programerima koji nadgledaju, ispravljaju i poboljšavaju kod koji proizvode automatizovani sistemi. Sve dok ljudsko oko mora da čita i održava kod, od suštinskog je značaja da to bude čist kod, kao što kaže ova knjiga.

— Karlos E. Fero

Diplomirani informatičar

Senior softverski inženjer kompanije Quorum Software

Buenos Ajres

20. jun, 2023.

PREDGOVOR

Programski kod se nalazi svuda, od razvoja veb stranica do pametnih ugovora, ugrađenih sistema, blokčejn tehnologije, softvera na svemirskom teleskopu Džejms Veb, hirurških robota i mnogih drugih oblasti. Softver praktično preuzima svet, a trenutno svedočimo usponu profesionalnih alata za generisanje veštačke inteligencije. To znači da je očuvanje čistog koda sada važnije nego ikad. Dok nastavljate da radite sa sve većim privatnim ili otvorenim kodnim bazama, čist kod je ključ održavanja svežine i spremnosti za dalji razvoj.

Za koga je ova knjiga

Ova knjiga vam pomaže da prepoznate uobičajene probleme u kodnoj bazi i ističe posledice tih problema, te vam na kraju pomaže da ih izbegnete uz jednostavne recepte koje je lako pratiti. To je vredan resurs koji može značajno pomoći programerima, recenzentima koda, arhitektama i studentima da unaprede svoje veštine i postojeće sisteme.

Kako je ova knjiga organizovana

Ova knjiga se sastoji od 25 poglavlja. Svako poglavlje počinje od nekih principa i osnovnih pojmova koji pokazuju prednosti čistog koda, njegove posledice i nedostatke kada se primenjuje nepravilno. Prvo poglavlje govori o osnovnom pravilu za čist kod: preslikavanje realnih entiteta 1:1 sa vašim dizajnom. Ovo pravilo služi kao osnova iz koje se mogu izvesti svi ostali principi.

Svako poglavlje sadrži nekoliko praktičnih recepata organizovanih po temama, uz alate i smernice za promene u vašem kodu. Svaki od ovih recepata će vam pomoći da napravite pozitivne promene i unapredite trenutnu situaciju. Pored saveta i primera, upoznaćete različite principe dizajna softvera, heuristike i pravila. U ovim savetima ćete pronaći uzorke koda na više programskih jezika, jer čist kod nije svojstven samo jednom od njih. Mnoge knjige o refaktorisanju fokusirane su na jedan programski jezik, a autori ažuriraju nove verzije koristeći najnovije, popularne programske jezike. Ova knjiga je neutralna prema programskom jeziku i većina saveta se primenjuje na mnoge programske jezike (osim ako nije drugačije naznačeno).

Kod treba čitati kao pseudokod, iako većina primera funkcioniše kako je prikazano. Kada sam suočen sa izborom između čitljivosti i performansi, uvek biram čitljivost. Definicije uobičajenih termina pružam kroz celu knjigu, ali takođe ih možete pronaći i u rečniku pojmova.

Šta vam je potrebno za korišćenje ove knjige

Da biste izvršavali uzorke koda, trebaće vam radno okruženje kao što su *O'Reilly sandboxes* ili *Replit*. Podstičem vas da prevedete uzorke koda na svoj omiljeni programski jezik. Danas to možete besplatno uraditi pomoću alatki za generisanje veštačke inteligencije. Za pisanje uzoraka koda u ovoj knjizi koristio sam alatke kao što su *GitHub Copilot*, *OpenAI Codex*, *Bard*, *ChatGPT* i mnoge druge. Uporeba ovih alatki mi je omogućila da u ovoj knjizi koristim više od 25 različitih programskih jezika, iako za mnoge od njih nisam ekspert.

Pristup digitalnom formatu ove knjige

Ova knjiga recepata za čist kod nudi besplatan pristup i mogućnost pretrage uvek dostupnom *online* izdanju na veb adresi *https://cleancodecookbook.com*.

Konvencije upotrebljene u ovoj knjizi

U ovoj knjizi su upotrebljene sledeće tipografske konvencije:

Kurziv

Označava nove termine, URL adrese, imejl adrese, nazive datoteka i ekstenzije datoteka.

`Konstantna širina`

Služi za programski kod, kao i unutar paragrafa da bi se ukazalo na programske elemente kao što su nazivi promenljivih ili funkcija, baze podataka, tipovi podataka, promenljive okruženja, iskaze i ključne reči.

`Konstantna širina podebljano`

Prikazuje komande ili druge tekstualne elemente koje bi korisnik trebalo doslovno da unosi.

`Kurziv konstantna širina`

Prikazuje tekst koji treba zameniti vrednostima koje korisnik unosi ili vrednostima koje su određene kontekstom.



Ovaj element označava savet ili sugestiju.



Ovaj element označava opštu napomenu.

 Ovaj element ukazuje na upozorenje ili oprez.

Upotreba primera koda

Dodatni materijali (primeri koda, vežbe, itd.) su dostupni za preuzimanje na veb adresi *https://github.com/mcsee/clean-code-cookbook*.

Ako imate tehničko pitanje ili problem pri upotrebi primera koda, molimo vas da pošaljete imejl na adresu *bookquestions@oreilly.com*.

Ova knjiga je tu da vam pomogne da obavite posao. Uopšteno, ako su primeri koda ponuđeni uz ovu knjigu, možete ih koristiti u svojim programima i dokumentaciji. Ne morate nas kontaktirati za dozvolu osim ako reprodukujete značajan deo koda. Na primer, pisanje programa koji koristi nekoliko delova koda iz ove knjige ne zahteva dozvolu. Prodaja ili distribucija primera iz *O'Reilly* knjiga zahteva dozvolu. Odgovaranje na pitanje citiranjem ove knjige i navođenje primera koda ne zahteva dozvolu. Uključivanje značajne količine primera koda iz ove knjige u dokumentaciju vašeg proizvoda zahteva dozvolu.

Cenimo, ali obično ne zahtevamo, citiranje knjige. Referenca obično sadrži naslov, autora, izdavača i ISBN. Na primer: „Recepti za čist kod autora Maksimilijana Kontijerija (O'Reilly). Autorska prava 2023 Maksimilijano Kontijeri."

Ako smatrate da vaša upotreba primera koda prelazi granice fer korišćenja ili dozvole navedene gore, slobodno nas kontaktirajte na *permissions@oreilly.com*.

POGLAVLJE 1

Čist kod

Kada je Martin Fauler definisao refaktorisanje u svojoj knjizi *Refaktorisanje: Poboljšanje dizajna postojećeg koda*, prikazao je snage, prednosti, kao i razloge iza refaktorisanja. Na sreću, nakon više od dve decenije, većina programera zna značenje refaktorisanja i izraz miris koda (code smells). Programeri se svakodnevno suočavaju s tehničkim dugom, a refaktorisanje je postalo osnovni deo razvoja softvera. U svojoj fundamentalnoj knjizi, Fauler je koristio refaktorisanje kako bi rešavao mirise koda. Ova knjiga prikazuje neke od tih metoda u obliku semantičkih recepata radi poboljšanja vaših rešenja.

1.1 Šta je miris koda?

Miris koda je simptom problema. Ljudi često misle da prisustvo mirisa koda dokazuje da čitav entitet treba rastaviti i ponovo izgraditi. Međutim, to nije duh originalne definicije. Mirisi koda su jednostavno pokazatelji mogućnosti za unapređenje. Miris koda vam ne govori nužno šta nije u redu; on vam sugeriše da posvetite posebnu pažnju.

Recepti ove knjige pružaju neka rešenja za te simptome. Kao i kod svakog kuvara, recepti su opcioni i mirisi koda su smernice i heuristika, a ne kruta pravila. Pre nego što primenite bilo koji recept na slepo, trebalo bi da razumete probleme i procenite cenu i prednosti sopstvenog dizajna i koda. Dobar dizajn uključuje balansiranje smernica sa praktičnim i kontekstualnim razmatranjima.

1.2 Šta je refaktorisanje?

Uzimajući u obzir knjigu Martina Faulera, on pruža dve dopunske definicije:

Refaktorisanje (imenica): promena unutrašnje strukture softvera da bi bio lakše razumljiv i ekonomičniji za modifikaciju, a da se pritom ne menja njegovo vidljivo ponašanje.

Refaktorisanje (glagol): prestruktuiranje softvera pomoću niza refaktorisanja bez menjanja njegovog vidljivog ponašanja.

Refaktorisanja je osmislio Vilijam Opdajk u svojoj doktorskoj disertaciji iz 1992. godine pod nazivom *Refaktorisanje objektno-orijentisanih radnih okvira*, a postala su popularna nakon Faulerove knjige. Ovi koncepti su se razvijali od Faulerove definicije. Većina savremenih integrisanih razvojnih okruženja podržava automatska refaktorisanja koja su bezbedna i omogućavaju strukturalne promene bez menjanja ponašanja sistema. Ova knjiga obuhvata mnoge primere sa automatskim, bezbednim refaktorisanjima, ali takođe i semantička refaktorisanja. Semantička refaktorisanja nisu bezbedna jer mogu promeniti deo ponašanja sistema. Treba pažljivo primenjivati primere sa semantičkim refaktorisanjima jer mogu dovesti do problema u softveru. Naznačiću da li određeni primer obuhvata semantičko refaktorisanje. Ukoliko imate kvalitetan test pokrivenosti koda koji proverava ponašanje aplikacije, možete biti sigurni da nećete narušiti ključne poslovne scenarije. Ne bi trebalo primenjivati primere refaktorisanja dok istovremeno ispravljate greške ili razvijate nove funkcionalnosti.

Većina savremenih organizacija poseduje obimne skupove testova u svojim procesima neprekidne integracije i isporuke. Pogledajte *Softversko inženjerstvo u Guglu* od Tita Vintersa i saradnika (O'Reilly 2020) da biste saznali da li posedujete ove skupove testova.

1.3 Šta je recept?

Pojam „recept" koristim s određenom slobodom. Recept je skup uputstava za stvaranje ili menjanje nečega. Recepti u ovoj knjizi najbolje funkcionišu ako razumete suštinu recepta, tako da ga možete primeniti sa vašim nijansama. Ostale knjige sa receptima iz ove serije su konkretnije, sa rešenjima korak po korak. Da biste iskoristili recepte iz ove knjige, moraćete da ih prevedete na vaš programski jezik i dizajnersko rešenje. Recept je sredstvo koje vas uči kako da razumete problem, prepoznate posledice i unapredite vaš kod.

1.4 Zašto čist kod?

Čist kod je lak za čitanje, razumevanje i održavanje. Dobro je strukturiran, sažet i koristi smislene nazive za promenljive, funkcije i klase. Takođe sledi najbolje prakse i dizajnerske obrasce i favorizuje čitljivost i ponašanje naspram performansi i detalja implementacije.

Čist kod je od ključnog značaja u svim sistemima koji se konstantno razvijaju i u kojima svakodnevno vršite izmene. Ostaje posebno relevantan u određenim okruženjima gde nije moguće implementirati ažuriranja onoliko brzo koliko bismo želeli. To uključuje ugrađene sisteme, svemirske sonde, pametne ugovore, mobilne aplikacije i mnoge druge primene.

Klasične knjige o refaktorisanju, veb sajtovi i integrisana razvojna okruženja fokusirani su na refaktorisanja koja ne menjaju ponašanje sistema. U ovoj knjizi ćete naći recepte za slične scenarije, poput sigurnih preimenovanja. Ali takođe ćete pronaći i nekoliko recepata vezanih za semantička refaktorisanja gde menjate način na koji rešavate neke probleme. Morate razumeti kod, probleme i recept da biste napravili odgovarajuće promene.

1.5 Čitljivost, performanse ili i jedno i drugo

Ova knjiga se bavi čistim kodom. Neki od njenih recepata nisu najefikasniji. Kada dođe do sukoba, biram čitljivost umesto performansi. Na primer, posvetio sam čitavo poglavlje (poglavlje 16) preuranjenoj optimizaciji kako bih se bavio problemima performansi bez dovoljno dokaza.

Za rešenja kritična po pitanju performansi najbolja strategija je pisati čist kod, pokriti ga testovima, a zatim poboljšati uska grla pomoću Paretovih pravila. Paretovo pravilo primenjeno na softver tvrdi da ćete, rešavajući 20% kritičnih uskih grla, poboljšati performanse softvera za 80%. Ako poboljšate 20% loših performansi, verovatno će se brzina sistema povećati za 80%.

Ova metoda vas odvraća od pravljenja preuranjenih optimizacionih promena bez scenarija zasnovanih na dokazima, jer će to rezultirati malim poboljšanjem i narušiti čist kod.

1.6 Tipovi softvera

Većina recepata u ovoj knjizi namenjena je serverskim sistemima sa složenim poslovnim pravilima. Simulator koji ćete početi da gradite počevši od poglavlja 2 je savršen za to. Pošto su recepti nezavisni od domena, većinu njih možete koristiti i za razvoj korisničkog interfejsa, baze podataka, ugrađene sisteme, blokčejn i mnoge druge scenarije. Takođe postoje i specifični recepti sa primerima koda za korisničko iskustvo, razvoj korisničkog interfejsa, pametne ugovore i druge specifične domene (pogledajte, na primer, recept 22.7, „Sakrivanje grešaka niskog nivoa od krajnjih korisnika").

1.7 Mašinski generisan kod

Da li nam je potreban čist kod sada kada postoji mnogo dostupnih alata za računarski generisan kod? Odgovor u 2023. godini je: da. Više nego ikada. Postoji mnogo komercijalnih asistenata za pisanje softverskog koda. Međutim, oni još uvek nemaju potpunu kontrolu; oni su kopiloti i pomoćnici, a ljudi su i dalje ti koji donose odluke o dizajnu.

U vreme pisanja ove knjige, većina komercijalnih alata i alata zasnovanih na veštačkoj inteligenciji pruža osnovna rešenja i standardne algoritme. Ali su neverovatno korisni kada ne možete da se setite kako da napravite malu funkciju i veoma su praktični za prevođenje između programskih jezika. Intenzivno sam ih koristio dok sam pisao ovu knjigu. Nisam savladao preko 25 programskih jezika koje sam koristio u receptima. Prevodio sam i testirao različite delove koda na različitim programskim jezicima pomoću mnogih alata-asistenata. Pozivam vas da takođe koristite sve dostupne alate kako biste preveli neke od recepata iz ove knjige na vaš omiljeni programski jezik. Alati su tu da ostanu, a budući programeri će biti tehnološki kentauri: pola ljudi, pola mašine.

1.8 Razmatranje o imenovanjima u knjizi

U knjizi koristim sledeće termine naizmenično:

- Metode/funkcije/procedure
- Atributi/ instance promenljivih/svojstva
- Protokol/ponašanje/interfejs
- Argumenti/saradnici/parametri
- Anonimne funkcije/zatvorenja/lambda funkcije

Razlike među njima su suptilne i ponekad zavise od programskog jezika. Dodajem napomenu kada je potrebno pojasniti upotrebu.

1.9 Obrasci dizajna

Ova knjiga pretpostavlja da čitalac ima osnovno razumevanje koncepata objektno-orijentisanog dizajna. Neki od primera u ovoj knjizi zasnivaju se na popularnim obrascima dizajna, uključujući one opisane u knjizi „velike četvorke" *Gotova rešenja – elementi objektno-orijentisanog softvera*. Ostali primeri predstavljaju manje poznate obrasce, kao što su *null objekat* i *metod objekat*. Pored toga, ova knjiga sadrži objašnjenja i uputstva o tome kako zameniti obrasce koji se sada smatraju anti-obrascima, kao što je obrazac *unikat* u receptu 17.2, „Zamena unikata".

1.10 Paradigme programskih jezika

Prema Dejvidu Farliju:

> Opsesija naše industrije programskim jezicima i alatima bila je štetna za našu profesiju. To ne znači da nema napretka u dizajnu jezika, ali se čini da većina napora u dizajniranju jezika usmerava na pogrešne stvari, poput sintaktičkih napredaka umesto strukturalnih napredaka.

Koncepti čistog koda predstavljeni u ovoj knjizi mogu se primeniti na razne programske paradigme. Mnoge od ovih ideja imaju korene u strukturiranom programiranju i funkcionalnom programiranju, dok druge potiču iz objektno-orijentisanog sveta. Ovi koncepti mogu vam pomoći da pišete elegantniji i efikasniji kod u bilo kojoj paradigmi.

Većinu recepata ću koristiti u objektno-orijentisanim jezicima i izgraditi simulator (nazvan *MAPPER*) pomoću objekata kao metafora za entitete stvarnog sveta. Često se u knjizi pozivam na MAPPER. Mnogi recepti će vas navesti da razmišljate o ponašajnom i deklarativnom kodu (pogledajte poglavlje 6, „Deklarativni kod") umesto o kodu implementacije.

1.11 Objekti naspram klasa

Većina recepata u ovoj knjizi govori o objektima, a ne o klasama (iako postoji čitavo poglavlje o klasifikaciji: pogledajte poglavlje 19, „Hijerarhije"). Na primer, recept 3.2 nosi naslov „Prepoznavanje suštine vaših objekata" umesto „Prepoznavanje suštine vaše klase". Ova knjiga govori o korišćenju objekata da bi se predstavili objekti iz stvarnog sveta.

Način na koji kreirate ove objekte je slučajan; možete koristiti klasifikaciju, prototipiranje, fabrike, kloniranje itd. Poglavlje 2 raspravlja o važnosti mapiranja vaših objekata i nužnosti da modelujete stvari koje možete videti u stvarnom svetu. Da biste kreirali objekte, mnogi jezici koriste *klase* koje su artefakti i nisu očigledni u stvarnom svetu. Potrebne su vam ako koristite jezik zasnovan na klasifikaciji. Ali one nisu glavni fokus recepata.

1.12 Prilagodljivost

Kvalitetan kod ne podrazumeva samo to da vaš program radi kako treba, već i da je jednostavan za ažuriranje i nadogradnju. Prema rečima Dejva Farlija u njegovoj knjizi *Savremeno softversko inženjerstvo*, trebalo bi da budemo majstori u učenju i prilagođavanju softvera budućim promenama. To predstavlja veliki izazov za IT sektor, a nadam se da će ova knjiga biti koristan alat u praćenju tih trendova.

POGLAVLJE 2

Uspostavljanje aksioma

2.0 Uvod

Evo uobičajene definicije softvera:

> Uputstva koja računar izvršava, za razliku od fizičkog uređaja na kojem se izvršavaju (tzv. „hardver").

Softver je definisan svojom suprotnošću; sve što nije hardver. Ovo nije baš dobra definicija onoga što softver zapravo jeste. Evo još jedne popularne definicije:

> Softver, uputstva koja govore računaru šta da radi. Softver obuhvata ceo skup programa, procedura i rutina povezanih sa radom računarskog sistema. Ovaj termin je skovan da bi se razlikovao od hardvera, tj. fizičkih komponenti računarskog sistema. Skup uputstava koji usmerava hardver računara da obavlja neki zadatak naziva se program ili softverski program.

Pre mnogo decenija, programeri su shvatili da je softver mnogo više od pukih uputstava. Uz ovu knjigu razmišljaćete o ponašanju sistema i shvatićete da je osnovna svrha softvera:

> Oponašati nešto što se događa u mogućoj stvarnosti.

Ova ideja vraća nas poreklu modernih programskih jezika, kao što je *Simula*.



Simula

Simula je bio prvi objektno-orijentisani programski jezik koji je uključivao klasifikaciju. Njegovo ime jasno ukazuje da je svrha izgradnje softvera bila stvaranje simulatora. To je i dalje slučaj sa većinom današnjih računarskih softverskih aplikacija.

U nauci, pravite simulatore da biste razumeli prošlost i predvideli budućnost. Još od vremena Platona, ljudi su pokušavali da izgrade dobre modele stvarnosti. Softver možete definisati kao izgradnju simulatora sa akronimom *MAPPER*:

Model: Apstraktan, Delimičan i Programibilan Objašnjavač Stvarnosti (Model: Abstract Partial and Programmable Explaining Reality).

Ovaj akronim će se često pojavljivati u knjizi. Pogledajmo detaljnije elemente koji čine MAPPER.

2.1 Zašto je to model?

Model je rezultat posmatranja određenog aspekta stvarnosti kroz specifičnu prizmu i perspektivu, primenjujući određenu paradigmu. To nije konačna, nepromenjiva istina, već najtačnije razumevanje koje trenutno imate na osnovu vašeg trenutnog znanja. Cilj softverskog modela, kao i bilo kojeg drugog modela, je da predvidi ponašanje u stvarnom svetu.



Model

Model objašnjava predmet koji opisuje pomoću intuitivnih koncepata ili metafora. Krajnji cilj modela je razumevanje kako nešto funkcioniše. Prema rečima Petera Naura, „Programirati znači graditi teorije i modele."

2.2 Zašto je apstraktan?

Model nastaje iz skupa delova. Ne možete ga u potpunosti razumeti posmatranjem izolovanih komponenti. Model se bazira na ugovorima i ponašanju, a oni ne detaljišu nužno na temu kako bi trebalo nešto postići.

2.3 Zašto je programabilan?

Svoj model bi trebalo da pokrećete u simulatoru koji reprodukuje željene uslove, što može biti Tjuringov model (kao što su moderni komercijalni računari), kvantni računar (budući računari) ili bilo koji drugi tip simulatora koji može da prati evoluciju modela. Možete da programirate model da reaguje na vaše radnje na određene načine, a da zatim posmatrate kako se sam razvija.



Tjuringov model

Računar zasnovan na *Tjuringovom modelu* je teorijski uređaj koji je sposoban da izvrši bilo koji izračunljiv zadatak za koji se mogu napisati skup instrukcija ili algoritam. Tjuringova mašina se smatra teorijskim temeljem modernog računarstva i služi kao model za dizajn i analizu stvarnih računara i programskih jezika.

2.4 Zašto je delimičan?

U cilju modeliranja problema koji nas zanima, uzimamo u obzir samo delimičan aspekt stvarnosti. Često je u naučnim modelima potrebno pojednostaviti određene aspekte koji nisu od suštinskog značaja, da bismo izolovali problem. Prilikom sprovođenja naučnih eksperimenata, neophodno je izolovati određene promenljive i fiksirati ostale da bismo testirali hipoteze.

U slučaju simulacije, ne možemo modelirati celu stvarnost, već samo relevantan deo nje. Nije potrebno modelirati celokupan predmet posmatranja (tj. stvarni svet), već samo interesantno ponašanje. Mnogi recepti u ovoj knjizi se bave problemom prekomernog dizajniranja modela uključivanjem nepotrebnih detalja.

2.5 Zašto je objašnjavajući?

Model mora biti dovoljno deklarativan da omogući posmatranje svoje evolucije i da vam pomogne da razumete i predvidite ponašanje u stvarnosti koju modelirate. Trebalo bi da bude sposoban da objasni šta radi i kako se ponaša. Mnogi moderni algoritmi mašinskog učenja ne pružaju informacije o tome kako dolaze do svojih izlaznih vrednosti (ponekad čak imaju halucinacije), ali modeli bi trebalo da budu u stanju da objasne šta su uradili, čak i ako ne otkrivaju specifične korake koje su preduzeli kako bi to postigli.



Da pojasnimo

Aristotel je rekao da „objasniti znači pronaći uzroke". Prema njemu, svaki fenomen ili događaj ima uzrok ili niz uzroka koji ga proizvode ili određuju. Cilj nauke je da identifikuje i razume uzroke prirodnih fenomena i, na osnovu toga, predvidi ponašanje u budućnosti.

Za Aristotela, „objašnjavanje" je podrazumevalo identifikaciju i razumevanje svih tih uzroka i njihove međusobne interakcije kod određenog fenomena. „Predviđanje", s druge strane, odnosi se na sposobnost korišćenja ovog znanja o uzrocima za predviđanje ponašanja fenomena u budućnosti.

2.6 Zašto je to povezano sa stvarnošću?

Model mora da simulira uslove koji se javljaju u posmatranom okruženju. Krajnji cilj je prognoziranje stvarnog sveta, kao u bilo kojoj simulaciji. U ovoj knjizi često ćete čitati o stvarnosti, stvarnom svetu i entitetima iz stvarnog sveta. Stvarni svet će vam biti krajnji izvor istine.

2.7 Zaključivanje pravila

Sada kada imate polaznu tačku za razumevanje softvera, možete početi sa dobrom praksom modeliranja i dizajniranja. Principi simulatora MAPPER će se pojavljivati u mnogim receptima.

U narednim poglavljima, nastavićete da otkrivate principe, heuristike, recepte i pravila za izradu odličnih softverskih modela, počevši od jednostavne aksiome koja je ovde uvedena: Model: Apstraktan, Delimičan i Programibilan Objašnjavač Stvarnosti. Radna definicija softvera za ovu knjigu je „simulator koji poštuje MAPPER akronim."



Aksioma

Aksioma je iskaz ili tvrđenje koji se smatra istinitim bez dokaza. Omogućava vam da izgradite logički okvir za rasuđivanje i zaključivanje, uspostavljanjem osnovnih pojmova i odnosa koji se mogu koristiti za izvođenje daljih istina.

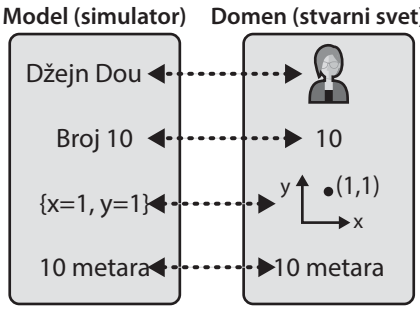
2.8 Jedini pravi princip dizajniranja softvera

Ako izgradite čitavu paradigmu dizajniranja softvera na osnovu jednog minimalnog pravila, možete zadržati stvari jednostavnim i kreirati odlične modele. Minimalistički i aksiomatski pristup je način da iz jedne definicije izvedete niz pravila:

> Ponašanje svakog elementa je deo arhitekture u meri u kojoj to ponašanje može pomoći u rasuđivanju o sistemu. Ponašanje elemenata obuhvata način na koji komuniciraju međusobno i sa okruženjem. Ovo je deo naše definicije arhitekture i uticaće na svojstva koja sistem ispoljava, kao što je njegova radna efikasnost.
>
> —Bas i saradnici, *Softverska arhitektura u praksi*, 4. izdanje

Jedan od najmanje cenjenih kvaliteta softvera je predvidljivost. Knjige često uče da softver treba da bude brz, pouzdan, robustan, vidljiv, bezbedan, itd. Predvidljivost je retko kada među pet najvažnijih prioriteta dizajna. Kao misaoni eksperiment, pokušajte da zamislite dizajniranje objektno-orijentisanog softvera praćenjem samo jednog pravila (kako je ilustrovano na slici 2-1): „Svaki domenski objekt mora biti predstavljen jednim objektom u izračunljivom modelu i obrnuto." Zatim pokušajte da izvedete sva pravila dizajna, heuristike

i recepte iz tog jednog preduslova da bi vaš softver bio predvidljiv, praćenjem recepata iz ove knjige.



Slika 2-1. *Odnos između objekata modela i entiteta iz stvarnog sveta je 1 prema 1*

Problem

Čitajući ove recepte za čist kod, shvatićete da većina implementacija jezika koje se koriste u industriji ignoriše pravilo jedne aksiome, što izaziva ogromne probleme. Većina savremenih programskih jezika je dizajnirana da rešava poteškoće u implementaciji koje su se javile u izgradnji softvera pre tri ili četiri decenije, kada su resursi bili oskudni i izračunavanja je trebalo da optimizuje programer. Sada su ovi problemi prisutni u vrlo malo domena. Recepti u ovoj knjizi će vam pomoći da prepoznate, razumete i rešite ove probleme.

Modeli za pomoć

Pri izgradnji bilo kakvih modela, važno je simulirati uslove koji se javljaju u stvarnom svetu. Možete pratiti svaki element od interesa u simulaciji i podsticati ga da biste videli da li se menja na isti način kao u stvarnom svetu. Meteorolozi koriste matematičke modele da predviđaju i prognoziraju vreme, a mnoge naučne discipline se oslanjaju na simulacije. Fizika traži objedinjene modele da bi razumela i predviđala pravila stvarnog sveta. Uz razvoj mašinskog učenja, počinjete da koristite modele koji nisu potpuno transparentni ili lako razumljivi, da biste simulirali i analizirali ponašanje u stvarnim uslovima.

Značaj bijekcije

U matematici, *bijekcija* je 1-1 funkcija, što znači da se svaki element iz domena preslikava u jedinstven element u kodomenu, i svaki element u kodomenu se može pratiti nazad do jedinstvenog elementa u domenu. Drugim rečima, bijekcija je funkcija koja uspostavlja *jedan-na-jedan* korespondenciju između elemenata dva skupa.

Izomorfizam, s druge strane, je jači tip korespondencije između dve matematičke strukture koja čuva strukturu objekata koji su u relaciji. Konkretno, izomorfizam je bijektivna funkcija koja čuva operacije struktura. Bijekcija je jedan-na-jedan korespondencija između dva skupa.

U domenu softvera, uvek ćete imati jedan i samo jedan objekat koji predstavlja entitet iz stvarnog sveta. Pogledajmo šta se događa ako ne postupate u skladu sa principima bijekcije.

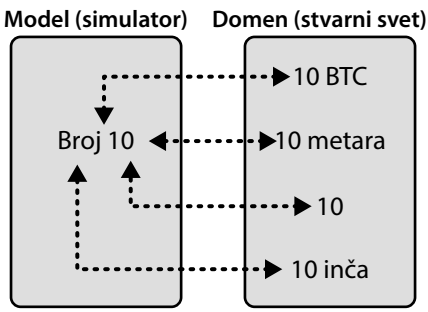
Česti slučajevi koji krše bijekciju

Ovo su četiri uobičajena slučaja koja krše princip bijekcije.

Slučaj 1

Jedan objekat u vašem izračunljivom modelu predstavlja više od jednog entiteta iz stvarnog sveta. Na primer, mnogi programski jezici modeluju algebarske mere koristeći samo skalarnu veličinu. Evo šta se dešava u ovom scenariju, kako je ilustrovano na slici 2-2.

- Možete predstaviti *10 metara* i *10 inča* (dva potpuno različita entiteta u stvarnom svetu) jednim objektom (*brojem 10*).
- Mogli biste ih sabrati, što bi dovelo do toga da model prikaže da je *broj 10* (koji predstavlja *10 metara*) plus *broj 10* (koji predstavlja *10 inča*) jednak *broju 20* (koji predstavlja ko zna šta).



Slika 2-2. *Broj 10 predstavlja više od jednog entiteta iz stvarnog sveta*

Bijekcija je narušena i to generiše probleme koji se ne uočavaju uvek na vreme. Pošto je reč o semantičkom problemu, posledice greške često uočavamo tek nakon nekog vremena, kao u čuvenom slučaju sa Marsovim klimatskim orbiterom.



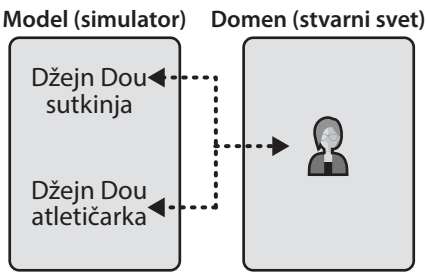
Marsov klimatski orbiter

Marsov klimatski orbiter bio je robotski svemirski brod koji je NASA lansirala 1998. godine s ciljem proučavanja klimatskih i atmosferskih uslova na Marsu. Misija nije uspela zbog problema sa sistemom za vođenje i navigaciju svemirskog broda. Motori svemirskog broda programirani su da koriste metričke jedinice sile, dok je tim za kontrolu sa Zemlje koristio engleske jedinice sile. Ova greška je dovela do toga da svemirski brod dođe preblizu površini planete i bude uništen pri ulasku u atmosferu. Problem sa Marsovim klimatskim orbiterom bio je neuspeh da se pravilno koordiniraju i konvertuju jedinice mere, što je dovelo do katastrofalne greške u putanji svemirskog broda. Sonda je eksplodirala mešajući različite jedinice mere. Bio je to veliki neuspeh za svemirsku agenciju NASA i koštao je agenciju 125 miliona dolara. Neuspeh je doveo i do niza promena u agenciji NASA, uključujući i otvaranje novog odeljenja za bezbednost i sigurnost misija (videti recept 17.1, „Eksplicitno iznošenje skrivenih pretpostavki").

Slučaj 2

Izračunljivi model predstavlja isti entitet iz stvarnog sveta pomoću dva objekta. Pretpostavimo da u stvarnom svetu postoji atletičarka Džejn Dou koja se takmiči u jednoj disciplini, ali je takođe i sutkinja u drugoj sportskoj disciplini. Jedna osoba u stvarnom svetu trebalo bi da bude predstavljena jednim objektom u izračunljivom modelu. Treba modelirati samo minimum ponašanja da bi se ispunila parcijalna simulacija.

Ako imate dva različita objekta (takmičarku i sutkinju) koji predstavljaju Džejn Dou, pre ili kasnije ćete imati nesaglasnosti. Ako jednoj od te dve uloge dodelite neku odgovornost i ne vidite kako se to odražava na drugu, imaćete nesaglasnosti (kako je ilustrovano na slici 2-3).



Slika 2-3. *Džejn Dou je predstavljena u modelu sa dva različita entiteta*

Slučaj 3

Bitkoin novčanik može biti predstavljen kao anemičan objekat sa nekim svojstvima kao što su adresa, stanje na računu i slično (videti recept 3.1, „Pretvaranje anemičnih objekata u bogate objekte"), ili kao bogat objekat (sa odgovornostima poput primanja transakcija, upisivanja u blokčejnu, prikazivanja stanja na računu, itd.) pošto su u vezi sa istim konceptom.

Morate prestati da posmatrate entitete samo kao strukture podataka sa atributima; razmislite o njima kao o objektima i shvatite da su to isti objekti koji ispunjavaju različite uloge u zavisnosti od konteksta. Poglavlje 3, „Anemični modeli", sadrži nekoliko recepata koji će vam pomoći da konkretizujete svoje objekte i pretvorite ih u entitete sa ponašanjem.



Konkretizacija objekta

Konkretizacija objekta je proces u kojem se apstraktnim idejama ili konceptima daje konkretan oblik. Osim što se materijalizuju kao objekti, ti koncepti dobijaju i funkcionalnost, prelazeći iz anemičnog ili čisto podatkovnog stanja u entitete sa ponašanjem. Ovako konkretizovani, objekti omogućavaju sistematično i strukturirano rukovanje i interakciju sa konceptima koje predstavljaju.

Slučaj 4

U većini modernih objektno-orijentisanih programskih jezika, *datum* se može kreirati unosom *dana*, *meseca* i *godine*. Ako unesete datum 31. novembar 2023, mnogi popularni programski jezici će vam ljubazno vratiti ispravan objekat (verovatno 1. decembar 2023).

Iako se ovo često prikazuje kao prednost, zapravo može sakriti određene greške koje se mogu pojaviti prilikom učitavanja podataka. Greška, u ovom slučaju nevažeći datum, biće otkrivena tek kasnije, kada se pokrene automatizovani proces obrade podataka ovih nevažećih datuma (često poznat kao „noćna grupna obrada"), daleko od glavnog uzroka problema, čime se krši princip „otkrij grešku što pre"(videti poglavlje 13, „Otkrij grešku što pre").



Princip brzog otkaza

Princip brzog otkaza ili „fail fast" naglašava da bi izvršenje programa trebalo prekinuti što je pre moguće ako se pojavi greška, umesto da je ignorišete i odlažete neuspeh.

Uticaj na jezik prilikom izrade modela

Ova knjiga se bavi poboljšanjem zaprljanog, nejasno definisanog, zagonetnog i unapred optimizovanog koda. Kao što sugeriše Sapir-Vorfova hipoteza, naše razumevanje sveta definiše jezik koji koristimo i potrebne su nam dobre metafore za objekte i ponašanja.



Sapir-Vorfova hipoteza

Sapir-Vorfova hipoteza, poznata i kao teorija lingvističke relativnosti, sugeriše da struktura i rečnik jezika osobe mogu uticati na njenu percepciju sveta. Jezik koji govorite ne samo da odražava i predstavlja stvarnost, već takođe igra ulogu u njenom oblikovanju i konstrukciji. To znači da je način na koji mislite i doživljavate svet delimično određen jezikom koji koristite da ga opišete.

Ako objekte posmatrate kao „čuvare podataka", vaši modeli će izgubiti *bijektivno* svojstvo i narušiti MAPPER. Mnogi recepti u vezi sa ovom temom nalaze se u poglavlju 3, „Anemični modeli". Ako objekte tretirate kao čuvare podataka, vaš računarski model (softver koji pravite) neće moći tačno da predviđa i simulira stvarni svet. Vaši korisnici će primetiti da im softver više ne pomaže u njihovom poslu. Ovo je čest izvor softverskih defekata (koje često i pogrešno nazivamo bagovima).



Bag

Termin *bag* je česta industrijska zabluda. U ovoj knjizi, umesto toga, govorim o defektima. Originalni bagovi su bili povezani sa insektima koji su ulazili u tople strujne krugove i uzrokovali nered u softverskom izlazu. To više nije slučaj. Preporučuje se upotreba termina defekt jer se odnosi na nešto što je uvedeno, a ne na spoljnog uljeza.