Edicija „Temeljno i intuitivno” (Grokking)

Otkrivanje jednostavnosti

Funkcionalno programiranje za kroćenje složenog softvera

Eric Normand

kompjuter biblioteka

Edicija „Temeljno i intuitivno” (Grokking)

Otkrivanje jednostavnosti

Eric Normand

kompjuter biblioteka

MANNING

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: Eric Normand

Prevod: Biljana Tešić

Lektura: Miloš Jevtović

Slog: Zvonko Aleksić

Znak Kompjuter biblioteke:
Miloš Milosavljević

Štampa: „Pekograf“, Zemun

Tiraž: 500

Godina izdanja: 2021.

Broj knjige: 543

Izdanje: Prvo

ISBN: 978-86-7310-566-6

grokking
Simplicity

ISBN 9781617296208

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.42.046

НОРМАНД, Ерик
Otkrivanje jednostavnosti / Eric Normand; [prevod Biljana Tešić]. - 1. izd. - Beograd: Kompjuter Biblioteka, 2021 (Zemun : Pekograf). - XXVI, 564 str.: ilustr.; 24 cm. - (Kompjuter biblioteka ; br. knj. 543) (Edicija „Temeljno i intuitivno“ : (Grokking))

Prevod dela: grokking Simplicity. - Tiraž 500. - Napomene uz tekst. - Registar.

ISBN 978-86-7310-566-6

а) Функционално програмирање

COBISS.SR-ID 45018121

Kratak sadržaj

Sadržaj

POGLAVLJE 9

POGLAVLJE 10

POGLAVLJE 15

POGLAVLJE 16

Predgovor

Guy Steele

Već više od 52 godine pišem programe. I dalje mi je uzbudljivo, jer uvek postoje novi problemi za rešavanje i nove „stvari" koje treba naučiti. Moj stil programiranja se znatno promenio tokom decenija dok sam učio nove algoritme, nove programske jezike i nove tehnike za organizovanje svog koda.

Kada sam naučio da programiram, 60-ih godina prošlog veka, dobro prihvaćena metodologija je bila da se nacrta dijagram toka programa pre pisanja stvarnog koda. Svako računanje bilo je predstavljeno pravougaonim okvirom, svaka odluka rombom, a svaka ulazno/izlazna operacija nekim drugim oblikom. Okviri su bili povezani strelicama koje su predstavljale tok kontrole od jednog do drugog okvira. Tada je pisanje programa bilo samo pisanje koda za sadržaj svakog okvira po nekom redosledu i kad god bi strelica ukazivala bilo gde, osim na sledeći okvir koji ste nameravali da kodirate, napisali biste iskaz `goto` koji bi ukazao na neophodan prenos kontrole. Problem je bio u tome što su dijagrami toka bili dvodimenzionalni, ali kod je bio jednodimenzionalan, pa čak i ako je struktura dijagrama toka izgledala lepo i uredno na papiru, mogla je da bude teško razumljiva kada se ispiše kao kod. Ako biste nacrtali strelice na svom kodu od svakog iskaza `goto` do odredišta, rezultat je često podsećao na gomilu špageta i u to vreme smo zaista govorili o poteškoćama razumevanja i održavanja „špageti koda".

Veliki uticaj na moj stil programiranja imao je najpre pokret „strukturiranog programiranja" ranih 70-ih godina prošlog veka. Ako se osvrnem unazad, vidim *dve* velike ideje koje su proizašle iz te rasprave u zajednici. Obe tehnike služe za **organizovanje kontrolnog toka**. Prva ideja koja je postala poznata bila je da se veći deo kontrolnog toka može izraziti pomoću nekoliko jednostavnih obrazaca: sekvencijalnog izvršavanja, višestrukih odluka, kao što su iskazi if-then else i switch, i ponavljajućeg izvršavanja, kao što su petlje while i for? To je ponekad bilo pojednostavljeno u slogan „Nema iskaza `goto`!", ali važni su bili obrasci i ako ste dosledno koristili obrasce, otkrili biste da retko treba da koristite stvarni iskaz `goto`. Druga ideja, manje poznata, ali ne manje važna, bila je da se sekvencijalni iskazi mogu grupisati u blokove koji bi trebalo da budu pravilno ugnežđeni i da prenos kontrole koji nije lokalni može „skočiti" na kraj bloka ili izvan njega (setite se iskaza `break` i `continue`), ali ne bi trebalo da „uskače" u blok spolja.

Kada sam naučio ideje strukturiranog programiranja, nisam imao pristup strukturiranom programskom jeziku. Međutim, primetio sam da malo pažljivije pišem svoj Fortran kod, organizujući ga u skladu sa principima strukturiranog programiranja. Čak sam pisao kod asemblerskog

jezika niskog nivoa kao da sam kompajler koji prevodi iz strukturiranog programskog jezika u mašinske instrukcije. Otkrio sam da mi je ova disciplina olakšala pisanje i održavanje programa. Doduše, još uvek sam pisao iskaze `goto` ili instrukcije grananja, ali skoro uvek u skladu sa jednim od standardnih obrazaca, a to je znatno olakšalo da vidim šta se dešava u kodu.

U loša stara vremena, kada sam pisao Fortran kod, sve neophodne promenljive u programu bile su deklarisane unapred, sve zajedno na jednom mestu, praćene izvršivim kodom. Jezik COBOL rigidno je formalizovao ovu specifičnu organizaciju koda; promenljive su deklarisane u „podeli podataka" programa koja je počinjala stvarnim rečima „DATA DIVISION". Zatim je usledio kod koji je uvek počinjao stvarnim rečima „PROCEDURE DIVISION". Na svaku promenljivu se može pozivati iz bilo koje tačke koda. Zbog toga je bilo teško za neku određenu promenljivu utvrditi kako tačno može da se izmeni i kako da joj se pristupi.

Moj stil programiranja promenilo je i „objektno-orijentisano programiranje", koje za mene obuhvata kombinaciju i vrhunac ranih ideja o objektima, klasama, „skrivanju informacija" i „apstraktnim tipovima podataka". Ponavljam: ako se osvrnem unazad, vidim *dve* velike ideje koje su proizašle iz ove velike sinteze, a obe su povezane sa **pristupom organizovanju podataka**. Prva ideja je da promenljive treba na neki način da bude „kapsulirane" ili „sadržane" da bi se lakše videlo da li ih mogu čitati ili pisati samo određeni delovi koda. Ovo može biti jednostavno poput deklarisanja lokalnih promenljivih unutar bloka, a ne na vrhu programa, ili razrađeno kao deklarisanje promenljivih unutar klase (ili modula) tako da im mogu pristupiti samo metodi te klase (ili procedure unutar modula). Klase ili moduli se mogu koristiti da bi se garantovalo da skupovi promenljivih poštuju određena svojstva konzistentnosti, jer metodi ili procedure mogu se kodirati kako bi se osiguralo da se, ako se ažurira jedna promenljiva, srodne promenljive takođe ažuriraju na odgovarajući način. Druga ideja je nasleđivanje, što znači da se može definisati složeniji objekat proširivanjem jednostavnijih objekata dodavanjem novih promenljivih i metoda i redefinisanjem postojećih metoda. Ova druga ideja je omogućena zahvaljujući prvoj.

U vreme kada sam saznao za objekte i apstraktne tipove podataka, pisao sam mnogo Lisp koda; iako sam Lisp nije čisto objektno-orijentisani jezik, prilično ga je lako koristiti za implementaciju struktura podataka i za pristup tim strukturama podataka samo pomoću odobrenih metoda (koji su implementirani kao Lisp funkcije). Ako bih organizovao podatke, imao bih mnogo koristi od objektno-orijentisanog programiranja, mada sam kodirao na jeziku koji nije primenjivao tu disciplinu.

Na moj stil programiranja u velikoj meri je uticalo i funkcionalno programiranje, koje je ponekad previše pojednostavljeno u slogan „Bez neželjenih efekata!" Međutim, to nije realno. Kada pravilno razumete funkcionalno programiranje, možete da primenite tehnike za **organizovanje sporednih efekata**, tako da se oni ne javljaju *svuda* - **i to je tema ove knjige**.

Ponavljam: u stvari, postoje dve velike ideje koje funkcionišu zajedno. Prva velika ideja je razlikovanje *izračunavanja* koja nemaju efekta na spoljni svet i daju isti rezultat čak i kada se izvode više puta od *akcija* koje mogu proizvesti različite rezultate uvek kada se izvrše i mogu imati neke *sporedne efekte* u spoljnom svetu, poput prikazivanja teksta na ekranu ili lansiranja rakete. Program je lakše razumeti ako je organizovan u standardne obrasce koji olakšavaju otkrivanje koji delovi mogu imati neželjene efekte, a koji su „puki proračuni". Standardni obrasci mogu se podeliti u dve potkategorije: one koje se obično koriste u programima sa jednom programskom

niti (sekvencijalno izvršavanje) i one koje se obično koriste u programima sa više niti (paralelno izvršavanje).

Druga velika ideja je skup tehnika za obradu kolekcija podataka (nizova, lista, baza podataka) odjednom, a ne stavku po stavku. Ove tehnike su najefikasnije kada se objekti mogu obraditi nezavisno, bez sporednih efekata i njihovog uticaja, tako da druga ideja bolje funkcioniše, zahvaljujući prvoj.

Pomogao sam 1995. godine da se napiše prva kompletna specifikacija za programski jezik Java; sledeće godine sam pomogao da se napiše prvi standard za jezik JavaScript (standard ECMAScript). Oba ova jezika bila su jasno objektno-orijentisana; zaista, u Javi ne postoji globalna promenljiva - svaka promenljiva mora biti deklarisana u nekoj klasi ili metodu. I oba ta jezika nemaju iskaz `goto`; dizajneri jezika zaključili su da je pokret strukturiranog programiranja uspešan i da više nije potreban iskaz `goto`. Danas se milioni programera odlično slažu bez tog iskaza i globalnih promenljivih.

Međutim, šta je sa funkcionalnim programiranjem? Postoje neki čisto funkcionalni jezici, kao što je Haskell, koji se često koriste. Haskell možete koristiti za prikaz teksta na ekranu ili za lansiranje rakete, ali upotreba neželjenih efekata u Haskellu podleže veoma strogoj disciplini. Jedna od posledica je da usred Haskellovog programa ne možete samo ubaciti iskaz print tamo gde želite da biste videli šta se događa.

Sa druge strane, Java, JavaScript, C ++, Python i mnogi drugi programski jezici nisu *čisto* funkcionalni jezici, već su u njima primenjene ideje iz funkcionalnog programiranja koje ih čine mnogo lakšim za upotrebu. I ovo je poenta: kada shvatite ključne principe organizovanja sporednih efekata, ove jednostavne ideje mogu se koristiti u *bilo kojem* programskom jeziku. U ovoj knjizi naučićete kako se to radi. Možda ona izgleda obimna, ali lako se čita, ispunjena je praktičnim primerima i bočnim trakama u kojima se objašnjavaju tehnički pojmovi. Privuklo me je funkcionalno programiranje. Zaista sam uživao u njemu i nekoliko novih ideja želim da primenim u svom kodu. Nadam se da ćete i vi uživati!

Jessica Kerr

Jessitron, LLC

Kada sam naučila programiranje, zavolela sam ga zbog njegove predvidljivosti. Svaki od mojih programa bio je jednostavan: bio je mali, funkcionisao je na jednoj mašini i bio je lak za upotrebu jednoj osobi (meni). Voleti razvoj softvera je nešto drugačije. Softver nije mali. Nije ga napisala jedna osoba. Funkcioniše na mnogim mašinama, u mnogim procesima. Privlači različite ljude, uključujući i one koje nije briga kako softver funkcioniše.

Korisni softver ne mora da bude jednostavan.

Šta programer treba da uradi?

U toku prethodnih 60 godina tehnike funkcionalnog programiranja su „rasle" u glavama informatičara. Istraživači vole da daju pozitivne izjave o onome što se nikada ne može dogoditi.

U prethodne dve decenije programeri su usvojili ove tehnike u poslovnom softveru. To je dobar tajming, jer se podudara sa dominacijom veb aplikacija: svaka aplikacija je distribuirani sistem, preuzima se na nepoznate računare i na nju klikću nepoznati ljudi. Funkcionalno programiranje je dobro prilagođeno. Čitave kategorije grešaka koje je teško pronaći nikada se ne mogu dogoditi.

Međutim, funkcionalno programiranje se ne prenosi mirno sa akademske zajednice na poslovni softver. Ne koristimo Haskell. Ne počinjemo „od nule". Zavisimo od vremena izvršavanja i biblioteka koji su van naše kontrole. Naš softver komunicira sa mnogim drugim sistemima; nije dovoljno da samo generiše odgovor. Dug je put od „FP zemlje" do starog poslovnog softvera.

Eric je za nas krenuo na ovo „putovanje". Uputio se u funkcionalno programiranje, pronašao njegovu najkorisniju suštinu i doneo je do nas.

Nestali su precizno zadavanje tipa promenljive, „čisti" jezici i teorija kategorija. Umesto toga, primećujemo kod koji komunicira sa svetom, odlučujemo da ne menjamo podatke i razdvajamo kod radi veće jasnoće. Umesto apstrakcija visokog uma, ovde se koriste stepeni i nivoi apstrakcije. Umesto da izbegavate stanje, evo načina za bezbedno zadržavanje stanja.

Eric nudi nove načine za posmatranje našeg postojećeg koda – nove dijagrame, nove „mirise", novu heuristiku. Da, ovo je proizašlo iz „putovanja" u funkcionalno programiranje. Pa, ipak, kada Eric svoje načine razmišljanja učini eksplicitnim da bismo ih mogli primeniti, on stvara nešto novo, nešto što će svima nama pomoći u našim kreacijama.

Moji jednostavni programi nisu bili korisni svetu. Naš korisni softver nikada neće biti jednostavan. Međutim, možemo učiniti naš kod jednostavnijim nego što je bio. A mi možemo upravljati onim ključnim bitovima koji komuniciraju sa svetom. Eric nam otklanja ove kontradikcije.

Ova knjiga će vam poboljšati programiranje i, još više, razvoj softvera.

Uvod

Funkcionalno programiranje (FP) sam upoznao u obliku Common Lispa još 2000. godine, kada sam pohađao čas veštačke inteligencije na svom univerzitetu. U početku se Lisp činio oskudnim i stranim u odnosu na objektno-orijentisane jezike na koje sam navikao. Međutim, do kraja prvog semestra kreirao sam mnoge zadatke u Lispu i počeo je da mi se dopada. „Okusio" sam FP, čak i ako sam tek počinjao da ga razumem.

Tokom godina moja upotreba funkcionalnog programiranja se produbila. Napisao sam svoj Lisp. Čitao sam knjige o Lispu. Počeo sam da radim zadatke iz drugih razreda u njemu. A odatle sam upoznao Haskell i na kraju Clojure 2008. godine. U Clojureu sam pronašao svoju „muzu". Izrađen je na pedesetogodišnjoj tradiciji Lispa, ali na modernoj i praktičnoj platformi. A zajednica je „smućkala" ideje o računanju, prirodi podataka i praktičnom inženjeringu velikih softverskih sistema. Te ideje su bile „leglo" filozofije, računarstva i inženjerstva. I ja sam shvatio funkcionalno progamiranje. Blogovao sam o Clojureu i na kraju osnovao kompaniju koja podučava ljude o Clojureu.

U međuvremenu, svest o Haskellu je takođe bila u porastu. Nekoliko godina sam profesionalno radio na Haskell jeziku. Haskell je imao mnogo štošta zajedničkog sa Clojureom, ali bilo je i poprilično razlika. Kako bismo mogli definisati *funkcionalno programiranje* tako da uključuje i Clojure i Haskell? To pitanje je dovelo do „semena" ove knjige.

To prvo „seme" bila je ideja o akcijama, proračunima i podacima kao primarnoj distinkciji paradigme funkcionalnog programiranja. Ako pitate bilo kojeg funkcionalnog programera, složiće se da je ovo razlikovanje od suštinske važnosti za praksu FP-a, mada se mali broj programera slaže da je to odlučujući aspekt paradigme. To je izgledalo kao kognitivna disonanca. Ljudi obično podučavaju druge onako kako su sami naučeni. Uvideo sam priliku u toj kognitivnoj disonanci da pomognem ljudima da nauče FP na nov način.

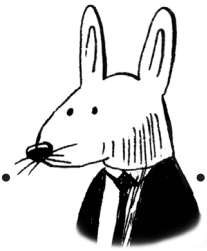
Radio sam na mnogim nacrtima knjige. Jedan je bio veoma teoretski. Drugi je pokazivao impresivne karakteristike FP-a. Treći je bio izuzetno didaktičan. Četvrti je bio potpuno narativan. Međutim, na kraju, uz mnogo podučavanja od mog urednika, stigao sam do trenutne inkarnacije koja funkcionalno programiranje tretira kao skup veština. U osnovi, to je bilo biranje veština koje su uobičajene u FP krugovima, ali su van njih retke. Kada sam se odlučio za takav pristup,

planiranje knjige je obuhvatalo samo pronalaženje, organizovanje i prioritizovanje tih veština. Posle toga pisanje se odvijalo veoma brzo.

Naravno, u ovoj knjizi ne možemo da predstaimo svaku veštinu. Funkcionalno programiranje staro je najmanje 60 godina. Postoji mnogo tehnika koje zaslužuju da budu naučene, a za koje jednostavno nisam imao mesta u knjizi. Siguran sam da bi se čitaoci usprotivili nekim mojim propustima, ali uveren sam da su veštine koje sam izabrao važni delovi repertoara profesionalnog funkcionalnog programera. Osim toga, te veštine otvaraju „vrata" za više veština. Nadam se da će drugi autori uzeti kao polaznu osnovu veštine iz ove knjige za diskusiju i dodatno podučavanje.

Moj glavni cilj je da pomoću ove knjige bar započnem proces legitimizovanja funkcionalnog programiranja kao praktične opcije za profesionalne programere. Kada programeri žele da nauče objektno-orijentisano programiranje, mogu da pronađu mnogo knjiga o toj temi napisanih samo za početnike profesionalce. Te knjige imenuju obrasce i principe i tehnike na kojima učenik može da „gradi" svoje veštine. Za funkcionalno programiranje nema iste pedagoške literature. Knjige koje postoje o funkcionalnom programiranju su, uglavnom, akademske, a, po mom mišljenju, one koje pokušavaju da privuku softversku industriju ne uspevaju da obuhvate najtemeljnije koncepte. Međutim, znanje i iskustvo postoje u glavama hiljada funkcionalnih programera. Nadam se da će ova knjiga doprineti da FP literatura svuda „procveta" iz teško stečenih veština funkcionalnih programera.

Zahvalnice



Pre svega se zahvaljujem Richu Hickeyju i celoj Clojure zajednici što su bili izvor filozofskih, naučnih i inženjerskih ideja povezanih sa programiranjem. Bili su mi velika inspiracija. Siguran sam da ćete prepoznati da mnoge ideje u ovoj knjizi potiču direktno iz Clojure majndseta.

Zahvaljujem se svojoj porodici, posebno Virginiji Medinillai, Oliviji Normand i Isabelli Normand, što su me podržavale tokom pisanja ohrabrenjem, strpljenjem i ljubavlju. Hvala i Liz Williams koja me je savetovala tokom pisanja ove knjige.

Hvala Guyu Steeleu i Jessi Kerr na pažnji koji su posvetili ovoj knjizi. I hvala im, naravno, što su u predgovoru podelili svoja lična iskustva.

Na kraju, zahvaljujem se ljudima iz Manninga. Bert Bates je zaslužila moju zahvalnost zbog bezbroj sati vijugave rasprave koja je nekako dovela do završetka knjige. Hvala joj na kontinuiranom podučavanju kako da budem bolji učitelj. I hvala na strpljenju i podršci dok sam shvatio šta knjiga treba da bude. Hvala Jenny Stout što je pratila ceo projekat. Hvala Jennifer Houle na prelepom dizajnu knjige. I hvala svima ostalima u Manningu koji su učestvovali u ovom mom projektu. Znam da je ovo bila teška knjiga na mnogo načina.

Recenzenti Michael Aydinbas, James J. Byleckie, Javier Collado, Theo Despoudis, Fernando García, Clive Harber, Fred Heath, Colin Joyce, Oliver Korten, Joel Luukka, Filip Mechant, Bryan Miller, Orlando Méndez, Naga Pavan Kumar T., Rob Pacheco, Dan Posey, Anshuman Purohit, Conor Redmond, Edward Ribeiro, David Rinck, Armin Seidling, Kaj Ström, Kent Spillner, Serge Simon, Richard Tuttle, YvanPhelizot i Greg Wright su mi mnogo pomogli da se ova knjiga poboljša – hvala im!

O knjizi

Ko treba da pročita ovu knjigu

Knjiga „Otkrivanje jednostavnosti“ je dizajnirana za programere koji imaju 2–5 godina iskustva. Očekujem da znate bar jedan programski jezik. Biće vam od pomoći ako ste izradili sistem dovoljno veliki da osetite „bol“ koju smo svi osetili kada se sistemi skaliraju. Primeri koda napisani su u stilu JavaScripta, koji naglašava čitljivost. Ako znate da čitate C, C #, C ++ ili Java jezik, ne bi trebalo da imate mnogo problema.

Kako je organizovana ova knjiga: mapa „puta“

Ova knjiga je organizovana u dva dela i 19 poglavlja. U svakom delu je predstavljena osnovna veština, a zatim se istražuju ostale veštine koje ona otvara. Svaki deo se završava delom o dizajnu i arhitekturi u kontekstu funkcionalnog programiranja. U Delu 1, koji započinje u Poglavlju 3, predstavljena je razlika između akcija, izračunavanja i podataka. U Delu 2, koji započinje u Poglavlju 10, predstavljena je ideja o vrednostima prve klase. Postoji „put“ mogućnosti visokog nivoa koje ove veštine otvaraju u Poglavlju 2.

- U Poglavlju 1 predstavljene su knjiga i glavna ideja funkcionalnog programiranja.
- Poglavlje 2 sadrži „put“ mogućnosti visokog nivoa onoga što je moguće korišćenjem veština iz ove knjige.

Deo 1: Akcije, izračunavanja i podaci

- U Poglavlju 3 započinje Deo 1 predstavljanjem praktičnih veština za razlikovanje akcija, izračunavanja i podataka.
- U Poglavlju 4 naučićete kako se refaktoriše kod u izračunavanja.

- U Poglavlju 5 saznaćete kako se mogu poboljšati akcije ako se ne mogu refaktorisati u izračunavanja.
- U Poglavlju 6 upoznaćete važnu disciplinu nepromenljivosti koja se zove kopiranje pri upisu (*copy-on-write*).
- U Poglavlju 7 ćemo predstaviti besplatnu disciplinu nepromenljivosti, koja se naziva *defanzivno kopiranje*.
- U Poglavlju 8 predstavljen je način za organizaciju koda prema slojevima značenja.
- Poglavlje 9 pomaže da analiziramo slojeve prema održavanju, ispitivanju i ponovnoj upotrebi.

Deo 2: Apstrakcije prve klase

- Poglavlje 10 započinje Deo 2 predstavljanjem ideje o vrednostima prve klase.
- Poglavlje 11 vas uči kako da bilo kojoj funkciji dodelite „supermoći".
- Poglavlje 12 pokazuje kako možete da kreirate i koristite alatke koje ponavljaju radnje preko niza.
- Poglavlje 13 pomaže da kreirate složene proračune pomoću alatki iz Poglavlja 12.
- U Poglavlju 14 predstavljene su funkcionalne alatke za bavljenje ugnežđenim podacima dok se dotičemo rekurzije.
- U Poglavlju 15 predstavljen je pojam vremenskih dijagrama kao način za analizu izvršavanja koda.
- U Poglavlju 16 pokazano je kako se mogu bezbedno, bez grešaka, deliti resursi između tokova događaja.
- U Poglavlju 17 saznaćete kako možete da manipulišete redosledom i ponavljanjem akcija da biste izbegli greške.
- U Poglavlju 18 se završava Deo 2 diskusijom o dva arhitektonska obrasca za izradu servisa u funkcionalnom programiranju.
- Poglavlje 19 sadrži retrospektivu i smernice za dalje učenje.

Knjigu treba da čitate od početka redom. Svako poglavlje se nadovezuje na sledeće. Obavezno radite vežbe. Vežbe „Razmislite o tome" ne sadrže odgovore. One treba da vam pomognu da formulišete sopstveno mišljenje o pitanjima koji se odnose na konkretnu temu. *Vi sami treba* da date odgovore. Ove vežbe su namenjene da vam obezbede praksu i poboljšaju veštine koje ste naučili u realnim scenarijama. Slobodno napravite pauzu u bilo kojem trenutku čitanja knjige. Niko nije savladao FP samo čitajući knjige. Ako ste naučili nešto važno, odložite knjigu i pustite „da se sve malo slegne" u vašoj glavi. Knjiga će biti tu kada budete spremni da nastavite čitanje.

O kodu

Kod postoji u celoj knjizi. Napisan je na JavaScriptu u stilu koji naglašava jasnoću, umesto trenutne najbolje prakse. Ja ne koristim JavaScript da bih zagovarao da radite funkcionalno programiranje na njemu. U stvari, on nije sjajan u FP-u. Međutim, pošto nema mnogo FP karakteristika, ovo ga čini odličnim jezikom za učenje FP-a. Moramo sami da izradimo mnogo funkcionalnih konstrukcija da bismo ih dobro razumeli. Takođe je dobro kada te funkcionalne konstrukcije obezbeđuju programski jezici, kao što su Haskell ili Clojure.

Delovi teksta koji su kod su jasni. Upućivanja na promenljive i druge delove sintakse koji se u tekstu pojavljuju koriste `ovakav font fiksne širine` da bi bili izdvojeni od normalnog teksta. Postoje i listinzi kodova koji koriste taj isti font. Ponekad je kod istaknut da bi se označilo da je nešto promenjeno u odnosu na prethodne korake. A promenljive i nazivi funkcija najvišeg nivoa podebljani su da bi bili naglašeni. Osim toga, podvlačenje koristim da bih skrenuo pažnju na relevantne delove koda.

Izvorni kod za primere u ovoj knjizi može se preuzeti sa veb sajta izdavača na adresi https://www.manning.com/books/grokking-simplicity.

liveBook forum za diskusiju

Ako kupite knjigu „Otkrivanje jednostavnosti“, dobijate besplatan pristup privatnom veb forumu, koji vodi „Manning Publications“, na kojem možete da komentarišete knjigu, da postavite tehnička pitanja i da dobijete pomoć od autora i drugih korisnika. Da biste pristupili forumu, posetite sajt https://livebook.manning.com/#!/book/grokking-simplicity/discussion. Takođe možete saznati više o „Manningovim“ forumima i pravilima ponašanja na adresi https://livebook.manning.com/#!/discussion.

„Manning“ nastoji da čitaocima obezbedi mesto na kojem se može odvijati smislen dijalog između pojedinih čitalaca i između čitalaca i autora. To nije posvećenost bilo kojem konkretnom učešću autora čiji doprinos forumu ostaje dobrovoljan (i neplaćen). Predlažemo da autoru postavite što više izazovnih pitanja da mu ne bi bilo dosadno! Forum i arhive prethodnih diskusija biće dostupni sa veb sajta izdavača sve dok je knjiga u prodaji.

Ostali izvori na Internetu

Previše je izvora na Internetu i van njega povezanih sa funkcionalnim programiranjem da bismo mogli sve da ih navedemo. I ni jedan nije dovoljno kanonski da bi zaslužio posebno pominjanje. Želeo bih da vas ohrabrim da potražite lokalne grupe za funkcionalno programiranje. Bolje se uči u društvu.

Ako želite dodatni materijal koji se odnosi posebno na ovu knjigu, potražite linkove do izvora i drugih materijala na sajtu https://grokkingsimplicity.com.

O knjizi

ERIC NORMAND je iskusni funkcionalni programer, trener, govornik i pisac o svemu što se odnosi na funkcionalno programiranje. Dolazi iz Nju Orleansa. Programiranje je započeo na Lisp programskom jeziku 2000. godine. Kreira materijale za Clojure obuku na sajtu PurelyFunctional.tv. Takođe se bavi savetovanjem kompanija da koriste funkcionalno programiranje za svoje poslovne ciljeve. Govornik je na međunarodnim programskim konferencijama. Njegovi tekstovi, govori, obuka i saveti mogu se naći na sajtu LispCast.com.

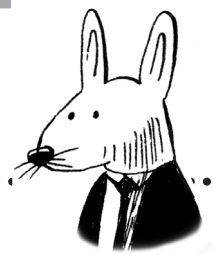
Postanite član Kompjuter biblioteke

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

Dobrodošli u otkrivanje jednostavnosti!

1



U ovom poglavlju:

- naučićete definiciju funkcionalnog mišljenja
- shvatićete po čemu se ova knjiga razlikuje od ostalih knjiga o funkcionalnom programiranju
- otkrićete primarnu razliku koju funkcionalni programeri prave kada gledaju kod
- procenićete da li je ova knjiga za vas

U ovom poglavlju ćemo definisati funkcionalno razmišljanje i razmotriti kako njegova glavna razlika pomaže programerima da izrade bolji softver. Takođe ćemo prikazati pregled budućeg putovanja pomoću dva glavna uvida koja funkcionalni programeri doživljavaju.

Šta je funkcionalno programiranje?

Programeri me stalno pitaju šta je funkcionalno programiranje (FP) i za šta je ono dobro. Za šta je FP dobro teško je objasniti, jer je to opšta paradigma. Dobro je za sve. Videćemo gde zaista „blista" na samo nekoliko stranica.

Takođe je teško definisati funkcionalno programiranje. Ono predstvlja ogromno polje. Koristi se i u softverskoj industriji i u akademskoj zajednici. Međutim, većina onoga što je napisano o funkcionalnom programiranju je iz akademske zajednice.

U knjizi „Otkrivanje jednostavnosti" pristupljeno je drugačije nego u tipičnim knjigama o funkcionalnom programiranju. Ova knjiga je posvećena industrijskoj upotrebi funkcionalnog programiranja - sve u njoj mora biti praktično za rad softverskih inženjera.

Možda ćete naići na definicije iz drugih izvora i važno je da razumete kako su one povezane sa onim što obrađujemo u ovoj knjizi. Evo tipične definicije parafrazirane sa Wikipedije:

funkcionalno programiranje (FP), *imenica.*

1. programska paradigma koju karakterišu upotreba matematičkih funkcija i izbegavanje sporednih efekata
2. stil programiranja koji koristi samo čiste funkcije bez sporednih efekata

Sada ćemo izdvojiti podvučene pojmove.

Sporedni efekti su sve ono što funkcija radi osim vraćanja vrednosti, poput slanja e-pošte ili promene globalnog stanja. To može biti problematično, jer se sporedni efekti javljaju uvek kada se pozove funkcija. Ako vam je potrebna povratna vrednost, a ne sporedni efekti, prouzrokovaćete da se neke radnje događaju nenamerno. Funkcionalni programeri obično veoma često žele da izbegnu nepotrebne sporedne efekte.

Ovo je razlog zašto mi uopšte pokrećemo softver!

Uobičajeni sporedni efekti uključuju:

- slanje e-pošte
- čitanje datoteke
- trepćuće svetlo
- podnošenje veb zahteva
- pritiskanje kočnica u automobilu

Čiste funkcije zavise samo od argumenata i nemaju nikakve sporedne efekte. Imajući u vidu iste argumente, one će uvek proizvesti istu povratnu vrednost. Ove *matematičke funkcije* možemo uzeti u obzir da bismo ih razlikovali od jezičke funkcije koja se naziva /functions/. Funkcionalni programeri stavljaju naglasak na upotrebu čistih funkcija, jer ih je lakše razumeti i kontrolisati.

Definicija podrazumeva da funkcionalni programeri u potpunosti izbegavaju sporedne efekte i koriste *samo* čiste funkcije. Međutim, to nije tačno. Funkcionalni programeri će koristiti sporedne efekte i nečiste funkcije.

Problemi definicije za praktičnu upotrebu

Definicija bi mogla biti dobra za akademsku zajednicu, ali ima niz problema za radnog softverskog inženjera. Sada ćemo ponovo pogledati definiciju:

funkcionalno programiranje (FP), *imenica*.

1. programska paradigma koju karakterišu upotreba matematičkih funkcija i izbegavanje sporednih efekata
2. stil programiranja koji koristi samo čiste funkcije bez sporednih efekata

Postoje tri glavna problema ove definicije za naše svrhe.

Problem 1: FP-u su potrebni sporedni efekti

Definicija „kaže" da FP izbegava sporedne efekte, ali sporedni efekti su upravo razlog zašto pokrećemo naš softver. Kakva je korist od softvera za e-poštu koji ne šalje e-poštu? Definicija podrazumeva da sporedne efekte potpuno izbegavamo, dok ih u stvarnosti koristimo kada moramo.

Problem 2: FP je dobar u sporednim efektima

Funkcionalni programeri znaju da su sporedni efekti neophodni, ali su problematični, tako da imamo mnogo alatki za upotrebu sporednih efekata. Definicija podrazumeva da koristimo *samo* čiste funkcije. Naprotiv, mi često koristimo nečiste funkcije - imamo mnoštvo funkcionalnih tehnika koje ih čine lakšim za upotrebu.

Problem 3: FP je praktičan

Definicija čini FP uglavnom matematičkim i nepraktičnim za softver iz stvarnog sveta. Postoji mnogo važnih softverskih sistema napisanih pomoću funkcionalnog programiranja. Definicija je posebno zbunjujuća za ljude koji poznaju FP pomoću definicije. Pogledajte primer dobronamernog menadžera koji čita definiciju na Wikipediji.

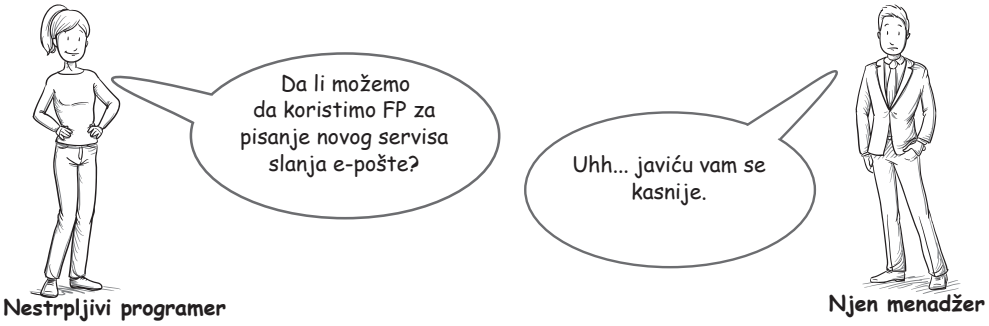
Vreme je za rečnik

Sporedni efekti su svako ponašanje funkcije, osim povratne vrednosti.

Čiste funkcije zavise samo od argumenata i nemaju nikakve sporedne efekte.

Definicija FP-a zbunjuje menadžere

Zamislite scenario u kojem Jenna, nestrpljivi programer, želi da koristi FP za slanje e-pošte. Ona zna da će FP pomoći arhitekti sistema da poboljša pouzdanost. Njen menadžer ne zna šta je FP, pa mora da potraži definiciju na Wikipediji.



Menadžer traži „funkcionalno programiranje“ na Wikipediji:

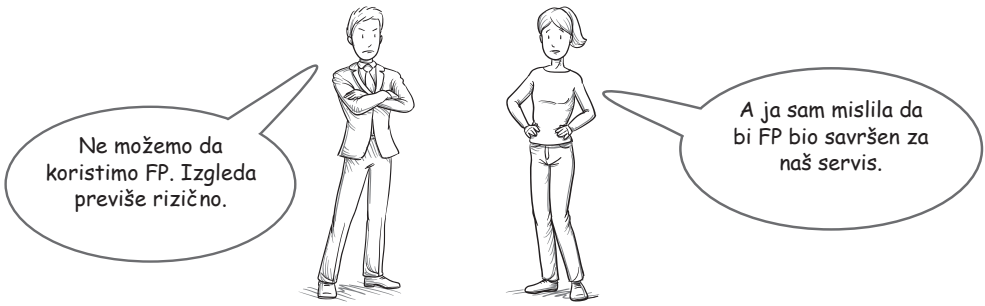
. . . izbegavanje sporednih efekata . . .

Hmmm, šta je to „sporedni efekat“?

slanje e-pošte

. . .

Kasnije tog dana . . .



Funkcionalno programiranje tretiramo kao skup veština i koncepata

Nećemo koristiti tipičnu definiciju u ovoj knjizi. FP ima mnoga značenja za mnoge ljude i ogromno je polje proučavanja i prakse.

Razgovarao sam sa mnogim funkcionalnim programerima šta im je od FP-a najkorisnije. „Otkrivanje jednostavnosti“ je destilacija veština, misaonih procesa i perspektiva funkcionalnih programera. Samo su najpraktičnije i najmoćnije ideje predstavljene u ovoj knjizi.

U knjizi „Otkrivanje jednostavnosti“ nećete pronaći najnovija istraživanja ili najezoteričnije ideje. Naučićete samo veštine i koncepte koje danas možete primeniti. Tokom svog istraživanja otkrio sam da se najvažniji koncepti iz FP-a primenjuju čak i na objektno-orijentisani i proceduralni kod i na svim programskim jezicima. Prava „lepota“ FP-a je u tome što se bavi korisnim univerzalnim tehnikama kodiranja.

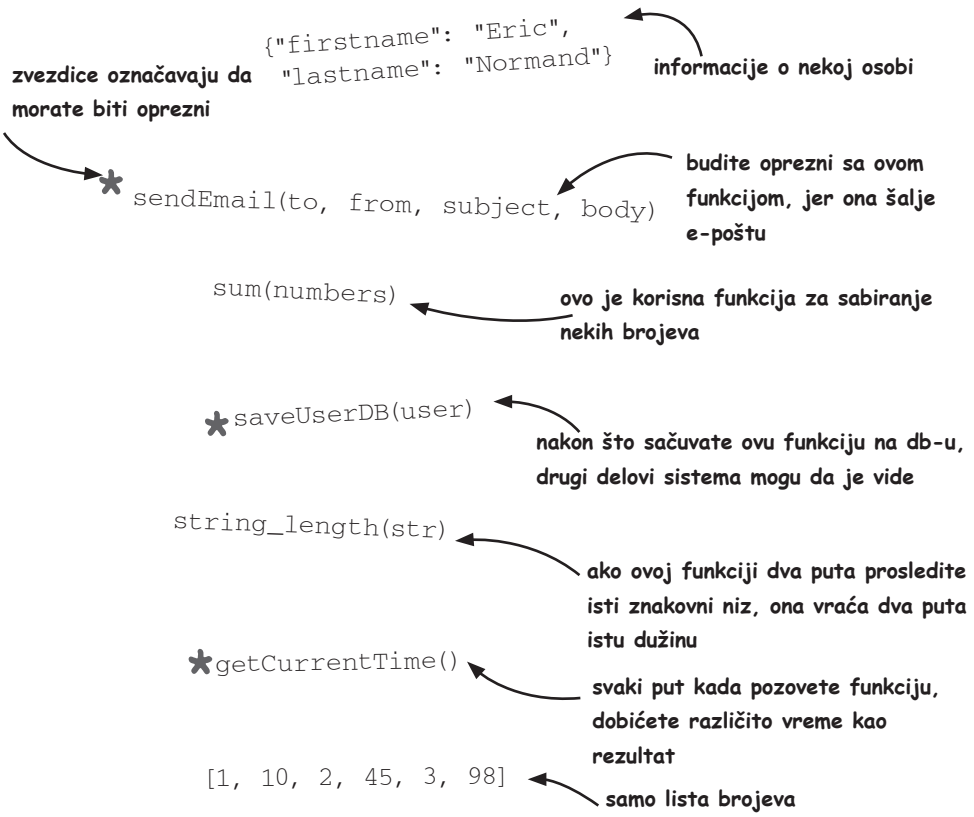
Sada ćemo da razmotrimo veštinu za koju bi svaki funkcionalni programer rekao da je važna: razlikovanje akcija, izračunavanja i podataka.

Razlikovanje akcija, izračunavanja i podataka

Kada funkcionalni programeri pogledaju kod, oni odmah počinju da ga klasifikuju u tri kategorije:

- akcije
- izračunavanja
- podaci

Evo nekoliko isečaka koda iz postojeće baze podataka. Morate biti oprezniji sa isečcima koji su označeni zvezdicom.

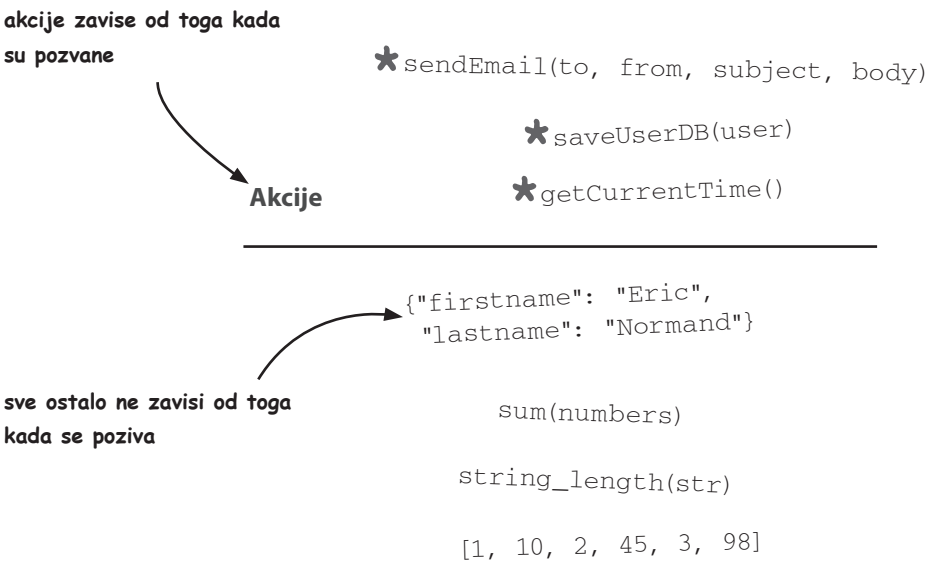


Treba da budete oprezni sa funkcijama koje su označene zvezdicom, jer one zavise od toga kada se pokreću ili koliko puta se pokreću. Na primer, ne želite dva puta da šaljete važnu e-poruku, niti želite da ona ne bude poslata.

Isečci koda označeni zvezdicom su *akcije*. Odvojimo ih od ostalih isečaka koda.

Funkcionalni programeri razlikuju kod koji je važan kada ga pozovete

Nacrtajte liniju i premestite sve funkcije koje zavise od poziva na jednu stranu linije:



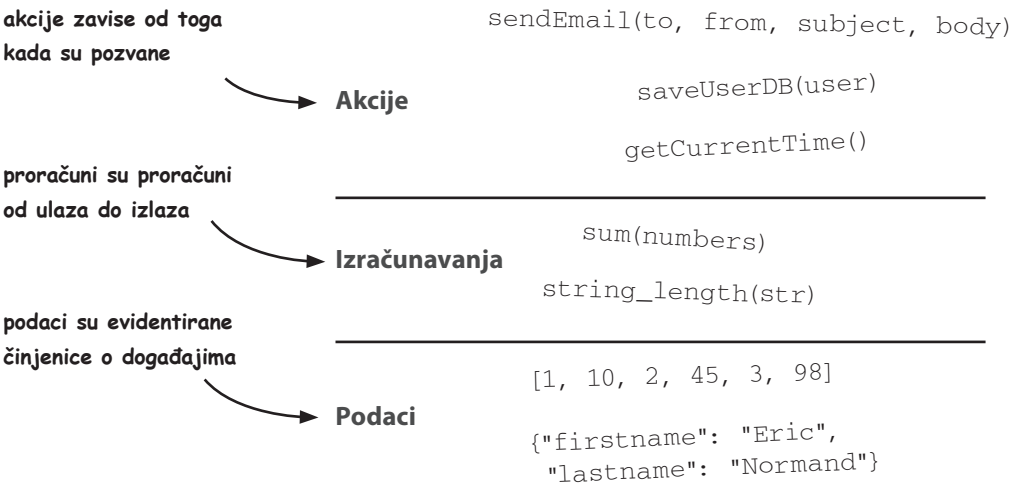
Ovo je veoma važna razlika. Akcije (sve iznad linije) zavise od toga kada su pozvane ili koliko puta su pozvane. Moramo biti posebno oprezni sa njima.

Međutim, isečke koda ispod linije je mnogo lakše upotrebiti. Nije važno kada pozivate funkciju sum. Daće vam tačan odgovor u bilo kojem trenutku. I nije važno koliko puta je pozivate. To neće uticati na ostatak programa ili na svet izvan softvera.

Možemo razlikovati još nešto: neki deo koda je izvršiv, a neki je inertan. Sada ćete nacrtati još jednu liniju na sledećoj stranici.

Funkcionalni programeri razlikuju umetnute podatke od koda koji funkcioniše

Možemo povući još jednu liniju između *izračunavanja* i *podataka*. Ni izračunavanja, ni podaci ne zavise od toga kada ili koliko puta se koriste. Razlika je u tome što se izračunavanja mogu izvršiti, dok podaci ne mogu. Podaci su inertni i transparentni. Izračunavanja nisu transparentna, što znači da, zapravo, ne znate šta će proračun raditi dok ga ne pokrenete.



Razlikovanje akcija, izračunavanja i podataka je osnovno za FP. Bilo koji funkcionalni programer složio bi se da je razlikovanje važna veština. Većina drugih koncepata i veština u FP-u „izgrađena“ je na toj veštini.

Važno je naglasiti da se funkcionalni programeri ne protive upotrebi koda u bilo kojoj od tri kategorije, jer su sve one važne. Međutim, oni prepoznaju kompromise i pokušavaju da koriste najbolje alatke za svoj posao. Generalno, oni više vole podatke od proračuna, a proračune od akcija. Podatke je najlakše koristiti.

Vredi ponovo naglasiti: **funkcionalni programeri vide ove kategorije kada gledaju bilo koji kod.** To je primarna razlika perspektive FP-a. Postoji niz veština i koncepata koji se odnose na ovu razliku. Na ove veštine ćemo se fokusirati tokom ostatka Dela 1.

Pogledajte šta ova razlika može „reći“ o jednostavnom servisu upravljanja zadacima.

Funkcionalni programeri više vole
podatke od proračuna i proračune od akcija.

Funkcionalni programeri vide akcije, izračunavanja i podatke

Razlikovanje akcija, izračunavanja i podataka je osnovno za FP. Ne biste mogli da vršite FP bez te veštine. Ovo je možda očigledno, ali samo da bi svima bilo jasno, sada ćemo razmotriti jednostavan scenario koji ilustruje tri kategorije.

Zamislimo servis u „oblaku" za upravljanje projektima. Kako klijenti označavaju svoje zadatke kao dovršene, tako centralni server šalje obaveštenja e-poštom.

Gde su akcije, izračunavanja i podaci? Drugim rečima, kako funkcionalni programer vidi šta se događa?

Korak 1: Korisnik označava zadatak kao završen.

Ovaj korak pokreće UI događaj, koji je akcija, jer zavisi od toga koliko puta se desi.

Korak 2: Klijent šalje poruku serveru.

Slanje poruke je *akcija*, ali sama poruka predstavlja *podatke* (inertne bajtove koji se moraju interpretirati).

Korak 3: Server prima poruku.

Primanje poruke je *akcija*, jer zavisi od toga koliko puta se dogodi.

Korak 4: Server vrši promenu u svojoj bazi podataka.

Promena unutrašnjeg stanja je *akcija*.

Korak 5: Server donosi odluku koga će obavestiti.

Donošenje odluke je *izračunavanje*. S obzirom da postoje isti ulazi, vaš server bi doneo istu odluku.

Korak 6: Server šalje obaveštenje e-poštom.

Slanje e-poruke je *akcija*, jer se slanje iste e-poruke dva puta razlikuje od slanja jednom.

Ako vam ovo sada nema smisla, ne brinite, jer ćemo potrošiti ceo prvi deo ove knjige da naučite kako se i zašto pravi razlika i kako se zbog toga poboljšava kod. Kao što smo ranije govorili, razlikovanje akcija, izračunavanja i podataka prva je velika ideja u ovoj knjizi.

Tri kategorije koda u FP-u

FP ima alatke za korišćenje svake kategorije

Pređimo na karakteristike tri kategorije:

1. Akcije

Sve što zavisi od toga kada je pokrenuto, ili koliko puta je pokrenuto, ili od oboje, je *akcija*. Ako danas pošaljem hitnu e-poštu, to se mnogo razlikuje od slanja e-pošte sledeće nedelje. Naravno, slanje iste e-pošte 10 puta razlikuje se od slanja 0 puta ili jedanput.

Akcije

- alatke za bezbednu promenu stanja tokom vremena
- načini za garantovanje redosleda
- alatke za osiguravanje da se radnje dogode tačno jednom

3. Izračunavanja

Izračunavanja su proračuni od ulaza do izlaza. Uvek generišu isti izlaz kada im zadate isti ulaz. Možete ih pozvati bilo kada i bilo gde i to neće uticati na bilo šta izvan njih. To ih čini zaista jednostavnim za testiranje i sigurnim za upotrebu, bez obzira koliko puta ili kada su pozvani.

Izračunavanja

- statička analiza za pomoć u ispravnosti
- matematičke alatke koje dobro funkcionišu za softver
- strategije testiranja

4. Podaci

Podaci su evidentirane činjenice o događajima. Podatke razlikujemo, jer nisu toliko složeni kao izvršivi kod. Imaju dobro shvaćena svojstva. Podaci su zanimljivi, jer su smisleni bez pokretanja. Mogu se interpretirati na više načina. Uzmimo za primer račun iz restorana: menadžer restorana može da ga koristi da bi utvrdio koje namirnice su popularne. Korisnik može da ga koristi za praćenje svog budžeta za večeru.

Podaci

- načini organizovanja podataka za efikasan pristup
- discipline za dugoročno vođenje evidencije
- principi za beleženje onoga što je važno korišćenjem podataka

Kako nam razlikovanje akcije, izračunavanja i podataka pomaže?

FP odlično funkcioniše za distribuirane sisteme, a većina softvera napisanih u današnje vreme se distribuira.

FP je popularan pojam u današnje vreme. Važno je znati da li je FP samo trend koji će jednog dana zamreti ili u njemu postoji nešto važno.

FP nije trend. To je jedna od najstarijih programskih paradigmi, koja vuče korene iz još starije matematike. Razlog zašto tek sada dobija na popularnosti je taj što nam je, zbog Interneta i ekspanzije uređaja, kao što su telefoni, prenosivi računari i serveri u „oblaku", potreban novi način na koji možemo sagledati softver koji uzima u obzir više delova softvera koji komuniciraju pomoću mreža.

Kada računari komuniciraju pomoću mreže, okolnosti postaju haotične. Poruke stižu bez redosleda, dupliraju se ili nikada ne stižu. Razumevanje onoga što se dogodilo kada se u osnovi modeliranje menja tokom vremena je veoma važno, ali i teško. Što više možemo da eliminišemo zavisnost od toga kada i koliko puta se nešto pokreće, lakše ćemo izbeći ozbiljne greške.

Podaci i izračunavanja ne zavise od toga koliko puta su pokrenuti ili koliko puta im je pristupljeno. Premeštanjem više koda u podatke i proračune sklanjamo taj kod od problema koji su svojstveni distribuiranim sistemima.

Problemi i dalje ostaju u akcijama, ali mi smo ih identifikovali i izolovali. Osim toga, FP ima skup alatki za upotrebu akcija da bismo bili sigurniji, čak i kada su distribuirani sistemi nesigurni. A pošto smo kod premestili iz akcija u proračune, možemo posvetiti više pažnje akcijama.

Tri pravila distribuiranih sistema

1. Poruke stižu nepravilno.
2. Svaka poruka može da ne stigne, može da stigne jednom ili više puta.
3. Ako nemate povratne informacije, nećete znati šta se dogodilo.

kada koristite distribuirane sisteme, „stvari" se zaista zakomplikuju

Zašto se ova knjiga razlikuje od ostalih FP knjiga?

Ova knjiga je praktična za softversko inženjerstvo.

Mnogobrojne rasprave koje se odnose na FP su akademske. Istraživači FP-a istražuju teoriju. Teorija je sjajna, sve dok ne morate da je primenite u praksi.

Mnoge knjige posvećene FP-u fokusirane su na akademsku stranu. One nas uče o rekurziji i o stilu prosleđivanja narednog koraka. Ova knjiga je drugačija. „Otkrivanje jednostavnosti" destiluje pragmatično iskustvo mnogih profesionalnih funkcionalnih programera - oni poštuju teorijske ideje, ali naučili su da se pridržavaju onoga što funkcioniše.

U ovoj knjizi se koriste realni scenariji

U ovoj knjizi nećete naći ni jednu Fibonačijevu definiciju ili sortiranje spajanja. Umesto toga, u scenarijima u ovoj knjizi oponašaju se situacije sa kojima biste se zaista mogli susresti na svom poslu. Funkcionalno razmišljanje primenjujemo na postojeći kod, na novi kod i na arhitekturu.

Ova knjiga se fokusira na dizajn softvera

Lako je rešiti mali problem i pronaći elegantno rešenje. Nema potrebe za arhitekturom dok pišete FizzBuzz. Tek kada „stvari" postanu velike, potrebni su principi dizajna.

Mnoge knjige o FP-u nikada ne treba dizajnirati, jer su programi veoma mali. U stvarnom svetu moramo da dizajniramo naše sisteme kako bi bili dugoročno održivi. Ova knjiga vas uči o principima funkcionalnog dizajna za svaki nivo skale, od linije koda, do celokupne aplikacije.

Ova knjiga prenosi bogatstvo FP-a

Funkcionalni programeri akumuliraju principe i tehnike još od 50-ih godina prošlog veka. Mnogo štošta se promenilo u računarstvu, ali mnogo je i izdržalo test vremena. U ovoj knjizi detaljno je pokazano kako je funkcionalno razmišljanje sada relevantnije nego ikad.

Ova knjiga je jezički agnostička

Mnoge knjige nas uče o karakteristikama određenog funkcionalnog jezika. To često znači da ljudi koji koriste drugi jezik ne mogu imati koristi.

U ovoj knjizi se koristi JavaScript za primere koda, što nije dobro za FP. Međutim, ispostavilo se da je JavaScript odličan za *podučavanje* o FP-u upravo zato što nije savršen. Ograničenja jezika dovešće do toga da zastanemo i razmislimo.

Iako se koristi JavaScript za primere, ovo nije knjiga o FP-u u JavaScriptu. Fokusirajte se na razmišljanje, a ne na jezik.

Primeri kodova napisani su radi preglednosti, a ne da bi predložili neki određeni JavaScript stil. Ako znate da čitate C, Java, C # ili C ++, moći ćete da pratite kodove.

Šta je funkcionalno razmišljanje?

Funkcionalno razmišljanje je skup veština i ideja koje funkcionalni programeri koriste za rešavanje problema koji se odnose na softver. To je veliki skup veština. U ovoj knjizi predstavljamo dve moćne ideje koje su veoma važne u funkcionalnom programiranju: razlikovanje akcija, izračunavanja i podataka i korišćenje apstrakcija prve klase. Ovo nisu jedine ideje u FP-u, ali one će vam obezbediti solidnu i praktičnu osnovu na kojoj možete „graditi" znanje. I vodiće vas od početnika do profesionalnog funkcionalnog programera.

Svaka ideja sa sobom nosi povezane veštine. One takođe korespondiraju sa dva dela ove knjige. Svaki deo vas uči o praktičnim veštinama, liniju po liniju i funkciju po funkciju, a na kraju imaju jedno ili dva poglavlja o dizajnu, radi „slike" na višem nivou o tome šta se dešava.

Sada ćemo preći na dve velike ideje i veštine koje ćete naučiti u svakom delu.

Deo 1: Razlikovanje akcija, izračunavanja i podataka

Kao što ste videli, funkcionalni programeri odvajaju sav kod u jednu od tri kategorije: akcije, izračunavanja ili podaci. Oni možda ne koriste ove iste pojmove, ali mi ćemo te tri kategorje tako nazivati u ovoj knjizi. One su u skladu sa tim koliko je kod teško razumeti, testirati i ponovo koristiti. Već smo počeli da otkrivamo tu važnu razliku u ovom poglavlju. U ovom prvom delu knjige naučićete da identifikujete kategoriju bilo kojeg dela koda, da refaktorišete akcije u proračune i da olakšate rad pomoću akcija. U poglavljima o dizajnu razmatramo kako prepoznavanje slojeva u kodu može da ga učini održivim, testiranim i ponovno upotrebljivim.

Deo 2: Korišćenje apstrakcija prve klase

Programeri svih vrsta naći će uobičajene procedure i imenovaće ih da bi kasnije mogli ponovo da ih koriste. Funkcionalni programeri rade isto, ali često mogu ponovo da koriste više procedura prosleđivanjem procedura na procedure. Ideja možda zvuči ludo, ali je izuzetno praktična. Naučićete kako to da radite i kako da ne preterate. Završićemo ovaj deo uobičajenim dizajnima koji se nazivaju reaktivna arhitektura i slojevita arhitektura.

Biće to dugačko „putovanje". Međutim, nećemo žuriti. Još uvek smo u prvom delu! Pre nego što započnemo „putovanje", postavićemo neka osnovna pravila za ono što ćete naučiti.

Osnovna pravila za ideje i veštine u ovoj knjizi

FP je velika tema. Nećete moći da naučite „celu stvar". Moraćete da izaberete šta ćete naučiti. Ova osnovna pravila će vam pomoći da izaberete praktično znanje za radnog programera.

1. Veštine se ne mogu zasnivati na jezičkim funkcijama

Postoji mnogo funkcionalnih programskih jezika koji imaju funkcije namenjene da podrže FP. Na primer, mnogi funkcionalni jezici imaju veoma moćne sisteme tipa. Ako koristite jedan od tih jezika, to je sjajno! Međutim, čak i ako ih ne koristite, funkcionalno razmišljanje vam može pomoći. Fokusiraćemo se samo na veštine koje su agnostičke za jezičke funkcije. To znači da ćemo ih, iako su sistemi tipa odlični, pomenuti samo ponekad.

2. Veštine moraju imati neposrednu praktičnu korist

FP se koristi i u softverskoj industriji i u akademskoj zajednici. Akademska zajednica se ponekad fokusira na teorijski važne ideje. To je sjajno, ali želimo da to postane praktično. Ova knjiga će vas podučavati samo za veštine koje vam mogu koristiti ovde i sada. U stvari, ideje koje ćete naučiti su korisne, čak i ako se ne pojavljuju u vašem kodu. Na primer, samo mogućnost identifikovanja akcija može vam pomoći da bolje razumete određene greške.

3. Veštine se moraju primeniti, bez obzira na vašu trenutnu situaciju sa kodom

Neki od nas započinju potpuno nove projekte, a da pri tom još uvek nije napisan kod. Neki od nas rade na postojećim bazama kodova sa stotinama hiljada linija koda. A neki od nas su „negde između". Ideje i veštine iz ove knjige trebalo bi da vam pomognu bez obzira na vašu situaciju. Ovde ne govorimo o funkcionalnom prepisivanju. Morate da napravite pragmatične izbore i da koristite kod koji imate.

Predah

Ima još mnogo štošta, ali napravimo pauzu za pitanja

P: Koristim objektno-orijentisani (OO) jezik. Da li će mi ova knjiga biti korisna?

O: Da, knjiga bi trebalo da bude korisna. Principi u knjizi su univerzalno primenljivi. Neki od njih su slični principima OO dizajna koji su vam možda poznati. A neki od njih su različiti, tj. proizilaze iz drugačije fundamentalne perspektive.

Funkcionalno razmišljanje je dragoceno, bez obzira koji jezik koristite.

P: Svaki put kada bih „zaronio" u FP, bio je veoma akademski i zasnovan na matematici. Da li je ova knjiga slična?

O: Ne! Naučnici vole FP, jer su proračuni apstraktni i lako se analiziraju u radovima. Nažalost, istraživači dominiraju u razgovoru o FP-u.

Međutim, postoji mnogo ljudi koji produktivno rade koristeći FP. I ako obratimo pažnju na akademsku literaturu, videćemo da funkcionalni programeri uglavnom kodiraju svoje svakodnevne poslove poput većine profesionalnih programera. Međusobno delimo znanje o rešavanju svakodnevnih problema. Deo tog znanja naći ćete u ovoj knjizi.

P: Zašto JavaScript?

O: To je zaista dobro pitanje. JavaScript je opštepoznat i dostupan. Ako programirate na Vebu, poznajete bar malo JavaScripta. Sintaksa je prilično poznata većini programera. I, verovali ili ne, JavaScript ima sve što vam je potrebno za FP, uključujući funkcije i neke osnovne strukture podataka.

JavaScript je daleko od savršenog za FP. Međutim, njegove nesavršenosti dopuštaju nam da koristimo principe FP-a. Učenje implementacije principa u jeziku koji ih ne implementira podrazumevano je korisna veština, pogotovo zato što većina programskih jezika nema podrazumevanu implementaciju principa.

P: Zašto postojeća definicija o FP-u nije dovoljna? Zašto se koristi termin „funkcionalno razmišljanje"?

O: To je takođe sjajno pitanje. Standardna definicija je korisna ekstremna pozicija koju treba zauzeti za pronalaženje novih puteva naučnog istraživanja. U osnovi se pitamo šta možemo postići ako ne dozvolimo sporedne efekte. Ispostavilo se da možemo učiniti mnogo, a mnogo štošta je zanimljivo za industrijski softverski inženjering.

Međutim, standardna definicija iznosi neke implicitne pretpostavke koje je teško otkriti. Ova knjiga vas prvo uči o tim pretpostavkama. „Funkcionalno razmišljanje" i „funkcionalno programiranje" su, uglavnom, sinonimi. Novi pojam jednostavno podrazumeva novi pristup.

Zaključak

Funkcionalno programiranje je veliko, bogato polje tehnika i principa. Međutim, sve započinje razlikovanjem akcija, izračunavanja i podataka. Ova knjiga vas učio praktičnoj strani FP-a. Može se primeniti na bilo kojem jeziku i za bilo koji problem. Postoje hiljade funkcionalnih programera i nadam se da će vas ova knjiga navesti da se ubrojite među njih.

Rezime

- Knjiga je organizovana u dva dela koja odgovaraju dvema velikim idejama i srodnim veštinama: razlikovanje akcija, izračunavanja i podataka i korišćenje apstrakcija višeg reda.
- Tipična definicija funkcionalnog programiranja služila je akademskim istraživačima, ali do sada nije postojala zadovoljavajuća definicija za softverski inženjering. Ovo objašnjava zašto FP može da bude apstraktan i nepraktičan.
- Funkcionalno razmišljanje su veštine i koncepti FP-a. To je glavna tema ove knjige.
- Funkcionalni programeri razlikuju tri kategorije koda: akcije, izračunavanja i podatke.
- Akcije zavise od vremena, pa su komplikovanije. Razdvajamo ih kako bismo mogli da im posvetimo veći fokus.
- Izračunavanja ne zavise od vremena. Želimo da napišemo još koda u ovu kategoriju, jer je to baš jednostavno.
- Podaci su inertni i zahtevaju tumačenje. Jednostavni su za razumevanje, čuvanje i prenos.
- U ovoj knjizi se koristi JavaScript za primere, jer on ima poznatu sintaksu. Naučićete nekoliko JavaScript koncepata kada vam budu potrebni.

Sledeće...

Sada, kada smo napravili dobar prvi korak u funkcionalnom razmišljanju, možda se pitate kako, zapravo, izgleda programiranje pomoću funkcionalnog razmišljanja. U sledećem poglavlju videćete primere rešavanja problema pomoću velikih ideja koje čine strukturu ove knjige.