Prevod knjige „Functional Design”

Funkcionalan dizajn

Principi, obrasci i praksa

Predgovor napisala Dženet A. Kar, nezavisna konsultantkinja za Clojure.

kompjuter biblioteka

Robert C. Martin

Funkcionalan dizajn

Praktičan vodič za bolji i čistiji kod uz funkcionalno programiranje

U knjizi *Funkcionalan dizajn*, poznati softverski inženjer Robert C. Martin (poznat i kao Ujka Bob) objašnjava kako i zašto funkcionalno programiranje služi za izgradnju boljih sistema za stvarne korisnike. Martin poredi konvencionalne strukture objektno-orijentisanog kodiranja u jeziku Java sa onima koje omogućavaju funkcionalni jezici, identifikuje najbolje uloge za svaku od njih i pokazuje kako da izgradite bolje sisteme mudrim korišćenjem istih u kontekstu.

Martinov pristup je pragmatičan, sa malo teorije i fokusom na rešavanju problema na terenu. Pomoću razumljivih i primenljivih primera, programeri će upoznati efikasnost i semantičku dubinu jezika Clojure, koji omogućava unapređenje kvaliteta koda, strukture i dizajna, kao i disciplinu pisanja koda, što dovodi do boljih krajnjih rezultata. Martin iz funkcionalne perspektive ispituje dobro poznate SOLID principe i projektne obrasce Velike četvorke i otkriva nam zašto su obrasci i dalje izuzetno važni za funkcionalno programiranje i kako da dođete do superiornih rezultata.

- Osnove funkcionalnog programiranja: nepromenljivost, trajnost podataka, rekurzija, iteracija, lenjost i čuvanje stanja
- Poređenje funkcionalnog i objektno-orijentisanog pristupa na studiozno kreiranim studijama slučaja
- Tehnika funkcionalnog dizajna za protok podataka
- Primena klasičnih SOLID principa za pisanje kvalitetnijeg Clojure koda
- Pragmatičan pristup funkcionalnom testiranju, grafički korisnički interfejsi i konkurentnost
- Maksimalna korist projektnih obrazaca u funkcionalnim okruženjima
- Praktičan vodič za izgradnju Clojure aplikacije visokog nivoa za preduzeća

Skenirajte QR kod, registrujte knjigu i osvojite nagradu

Funkcionalan dizajn odaje utisak klasika od trenutka objavljivanja. Bob otkriva kako elementi funkcionalnog programiranja čine dizajn softvera jednostavnim, a opet pragmatičnim. To postiže na način koji ne isključuje iskusne programere orijentisane na objektno programiranje na jezicima kao što su C#, C++ ili Java.

—Dženet A. Kar,
nezavisna Clojure konsultantkinja

Robert C. Martin („Ujak Bob”), je programer od 1970. godine, osnivač kompanije Uncle Bob Consulting, LLC, i suosnivač The Clean Coders, LLC, zajedno sa svojim sinom Mikom Martinom. Redovno govori na međunarodnim konferencijama i autor je renomiranih knjiga, među kojima se ističu Clean Code, Clean Architecture i The Clean Coder. Bio je glavni urednik časopisa C++ Report i prvi predsedavajući udruženja Agile Alliance.

kompjuter biblioteka
osnovana 1986.
www.kombib.rs

Pearson

ISBN: 978-86-7310-596-3

Funkcionalan dizajn

Principi, obrasci i praksa

Robert C. Martin

kompjuter
biblioteka

Izdavač:

kompjuter biblioteka

Obalskih radnika 4a
Beograd, Srbija
Tel: 011/2520272
e-pošta: kombib@gmail.com
veb-sajt: www.kombib.rs

Za izdavača:
Mihailo J. Šolajić, direktor

Autor:
Robert C. Martin

Prevod: Nemanja Lukić
Lektura: Nemanja Lukić
Recezent: Miroslav Ristić
Slog: Zvonko Aleksić

Znak Kompjuter biblioteke:
Miloš Milosavljević

Štampa: „Pekograf“, Zemun
Tiraž: 500
Godina izdanja: 2024.

Broj knjige: 573
Izdanje: Prvo
ISBN: 978-86-7310-596-3

Naslov originala:

Functional Design
Principles, Patterns, and Practices
by Robert C. Martin

ISBN-13: 978-0-13-817639-6

Funkcionalan dizajn

Principi, obrasci i praksa
Autorizovani prevod sa engleskog jezika.

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.42.046

МАРТИН, Роберт Сесил, 1952-
Funkcionalan dizajn : principi, obrasci i praksa /
Robert C. Martin; [prevod Nemanja Lukić]. - 1. izd. - Beograd: Kompjuter Biblioteka, 2024 (Zemun : Pekograf). - XVI, 347 str.: ilustr.; 26 cm. - (Kompjuter biblioteka; br. knj. 573)

Prevod dela: Functional Design. - Autorova slika. - Tiraž 500. - Str. XI-XII: Uvodna reč / Dženet A. Kar. - O autoru: str. [XIX]. - Napomene i bibliografske reference uz tekst. - Registar.

ISBN 978-86-7310-596-3

а) Функционално програмирање

COBISS.SR-ID 135191817

Kratak sadržaj

Sadržaj

Uvodna reč

Ujka Boba nije potrebno posebno predstavljati. On je istaknuta ličnost industrije razvoja softvera i autor više knjiga o dizajnu i isporuci softvera. Neka od njegovih dela se koriste na mnogobrojnim računarskim kursevima širom sveta.

Bila sam studentkinja kada sam počela da se bavim funkcionalnim programiranjem. Nisam pohađala elitni program gde bih učila Scheme i C, ali me je zanimalo sve što se tiče računarstva. Tada niko nije pričao o funkcionalnom programiranju. Videla sam trend programiranja koji nas čeka u budućnosti; budućnosti u kojoj će programeri više razmišljati o problemu koji rešavaju, nego o načinu da ga reše. Kada sam pročitala *Funkcionalan dizajn* poželela sam da sam ovu knjigu imala i tada i sada, u svakoj fazi svoje karijere, od kad sam bila studentkinja do sada kad sam profesionalac.

Funkcionalan dizajn deluje kao klasik u nastajanju. Deluje kao knjiga napisana tačno za profesionalnog dizajnera softvera. Bob preispituje osnove softverskog inženjeringa i nadograđuje ih, sažimajući u reči iskustva koja sam ja sticala godinama. On elegantno otkriva kako elementi funkcionalnog programiranja čine dizajn softvera jednostavnim, a pragmatičnim. To postiže na način koji ne isključuje iskusne programere orijentisane na objektno programiranje na jezicima kao što su C#, C++ ili Java.

Predstavljajući uporednu analizu sa Javom, knjiga *Funkcionalan dizajn* uvodi dizajn funkcionalnih sistema kroz Clojure, jedan od dijalekata Lispa. Clojure nije potpuno čist kao Haskell, koji zahteva isključivo korišćenje principa čistog funkcionalnog programiranja. Naprotiv, Clojure snažno naginje ka takvom pristupu, što ga čini idealnim za prve korake u funkcionalnom programiranju. *Funkcionalan dizajn* detaljno razmatra česte izazove na koje nailaze programeri u jeziku Clojure. Kao Clojure konsultantkinja, mogu to da potvrdim. Knjiga

objašnjava kako jezik (i programer) treba da bude diskretan u pristupu, umesto da teži pronalaženju rešenja za probleme koji se sklanjaju sa puta.

Kritičari jezika Clojure će reći da je Clojure neprikladan za bilo koji veći kod. Kao što ćete naučiti u narednim poglavljima, dizajnerski principi i obrasci su primenjivi na Clojure baš kao i na jezike Java, C# ili C++. SOLID principi će vam pomoći da funkcionalnim programiranjem izgradite bolji softver. Pobornici funkcionalnog programiranja odavno su ismejali projektne obrasce, ali knjiga *Funkcionalan dizajn* dekonstruiše takve kritike i pokazuje zašto su oni potrebni i kako ih programeri mogu samostalno implementirati.

Često sam pisala na internetu o tipičnim projektnim obrascima za Clojure, pa sam se oduševila kada sam otkrila da ova knjiga pristupa upotrebi projektnih obrazaca oslanjajući se na dobro osmišljene dijagrame, pre nego što čitaocu pokaže kod. Do trenutka kada dođete do tih poglavlja, već ćete moći da zamislite Clojure kod samo na osnovu dijagrama. Zatim sledi kod. Na kraju, *Funkcionalan dizajn* sve to povezuje kreiranjem „poslovne" aplikacije na jeziku Clojure, upotrebom dizajnerskih principa i obrazaca.

—Dženet A. Kar, nezavisna Clojure konsultantkinja

.

Uvod

Ovo je knjiga za programere koji žele da nauče da koriste jezike funkcionalnog programiranja za rešavanje konkretnih problema. Kao jedan od njih, neću trošiti mnogo vremena na teorijske aspekte funkcionalnog programiranja kao što su monade, monoidi, funktori, kategorije i slično. Ne zato što su ove ideje nevažne ili irelevantne, već zbog toga što obično nemaju veliki uticaj na svakodnevni rad programera. To je zato što su već „ugrađene u osnovu" uobičajenih jezika, biblioteka i radnih okvira. Ako vas zanima teorija funkcionalnog programiranja, preporučujem dela Marka Simana.

Ova knjiga govori o tome kako - i zašto – da koristimo funkcionalno programiranje u našim svakodnevnim naporima da izgradimo stvarne sisteme za stvarne klijente. Na stranicama koje slede, poredimo i suprotstavljamo strukture kodiranja uobičajene za objektno-orijentisane jezike, kao što je Java, sa onima koje su uobičajene za funkcionalne jezike, kakav je Clojure.

Izabrao sam ova dva jezika jer je Java veoma poznata i prihvaćena, a Clojure je izuzetno jednostavan za učenje.

Kratka istorija funkcionalnog i proceduralnog programiranja

Godine 1936, dvojica matematičara, Alan Tjuring i Alonzo Čerč, su nezavisno jedan od drugog rešili jedan od čuvenih problema Dejvida Hilberta: *Problem odlučivosti*. Detaljan opis ovog problema prelazi granice ovog uvoda, dovoljno je napomenuti da je bio povezan sa pronalaženjem opšteg rešenja za formule celih brojeva.[1] To je za nas važno jer je svaki program u digitalnom računaru jedna celobrojna formula.

Ovi matematičari su, nezavisno jedan od drugog, dokazali da opšte rešenje ne postoji, tako što su pokazali da postoje celi brojevi koji se ne mogu izračunati celobrojnom formulom koja je manja od samog tog celog broja.

Drugim rečima, postoje brojevi koje nijedan računarski program ne može da izračuna. Upravo je to bio pristup Alana Tjuringa. U svom čuvenom radu iz 1936. godine,[2] Tjuring je izmislio digitalni računar, a zatim pokazao da postoje brojevi koji se ne mogu izračunati - čak i za beskonačno vreme i prostor.[3]

S druge strane, Čerč je do istog zaključka došao zahvaljujući sopstvenom izumu - lambda računu, matematičkom formalizmu za manipulisanje funkcijama. Upotrebom manipulacija u logici svog formalizma, uspeo je da dokaže da postoje logički problemi koji ne mogu biti rešeni.

Tjuringov izum je predak svih savremenih digitalnih računara. Svaki digitalni računar je, u suštini, (konačna) Tjuringova mašina. Svaki program koji se ikada izvršavao na digitalnom računaru je, zapravo, program Tjuringove mašine.

Čerč i Tjuring su kasnije sarađivali da bi pokazali da su njihovi pristupi ekvivalentni. Da svaki program Tjuringove mašine može biti predstavljen lambda računom, i obrnuto.

---

1 Diofantove jednačine.

2 A. M. Turing, „On Computable Numbers, with an Application to the Entscheidungsproblem" (May 1936).

3 Za beskonačno vreme i prostor, računar bi mogao izračunati π, e ili bilo koji drugi iracionalan ili transcendentan broj za koji postoji formula. Ono što su Tjuring i Čerč dokazali je da postoje brojevi za koje takva formula ne postoji. Ti brojevi su „neizračunljivi".

Funkcionalno programiranje je, u suštini, programiranje u lambda računu.

Dakle, ova dva stila programiranja su ekvivalentna u matematičkom smislu. Svaki program može biti napisan ili proceduralnim (Tjuring) ili funkcionalnim (Čerč) stilom. Ono što ćemo ispitivati u ovoj knjizi nije ta ekvivalencija, već načini na koje korišćenje funkcionalnog pristupa utiče na strukturu i dizajn naših programa. Tragaćemo za odgovorom na pitanje da li su te različite strukture i dizajni u nekom smislu superiorni, ili inferiorni, u odnosu na one koje nastaju upotrebom Tjuringovog pristupa.

O jeziku Clojure

Izabrao sam Clojure za ovu knjigu jer je učenje novog jezika i nove paradigme duplo teži zadatak. Stoga sam težio da pojednostavim taj zadatak izborom jezika koji je dovoljno jednostavan da ne ometa učenje funkcionalnog programiranja i funkcionalnog dizajna.

Clojure je semantički bogat a sintaksički trivijalan. To znači da ima vrlo jednostavnu sintaksu koju je lako naučiti. Sav izazov učenja jezika Clojure leži u razumevanju njegove semantike. Usvajanje biblioteka i idioma zahteva značajan napor; ali sam jezik ne zahteva gotovo nikakav napor. Moj cilj je da uz ovu knjigu naučite i prihvatite funkcionalno programiranje, neometani sintaksom novog jezika.

Uz sve navedeno, ova knjiga nije kurs za jezik Clojure.[4] Objasniću osnove u početnim poglavljima i koristiću fusnote da razjasnim stvari, ali se takođe oslanjam na vas, dragi čitaoci, da uradite svoj deo posla i istražujete i sami. Postoji nekoliko veb stranica koje mogu biti od pomoći. Jedna od mojih omiljenih je https://clojure.org/api/cheatsheet.

Radni okvir za testiranje koji sam koristio u ovoj knjizi je `speclj`.[5] Kako poglavlja odmiču, viđaćete ga sve više i više. Vrlo je sličan drugim popularnim radnim okvirima za testiranje, tako da vam neće biti teško da postepeno spoznate njegove različite mogućnosti.

4 Na kraju ćete smatrati da sam lažov.
5 https://github.com/slagyr/speclj

O arhitekturi i dizajnu

Primaran fokus ove knjige je opis principa dizajna i arhitekture sistema izgrađenih funkcionalnim stilom. Za to ću koristiti UML dijagrame i reference na SOLID[6] principe dizajna softvera, *projektne obrasce*,[7] i koncepte *čiste arhitekture*. Nemojte da brinete, sve je objašnjeno, korak po korak, a imate i mnoge spoljne reference za dalje istraživanje, ukoliko vam to bude potrebno.

O objektnoj orijentaciji

Mnogi su izrazili mišljenje da su objektno-orijentisano i funkcionalno programiranje nekompatibilni. Stranice ove knjige bi trebalo da dokažu suprotno. Programi, dizajni i arhitekture koje ćete ovde videti su kombinacija funkcionalnih i objektno-orijentisanih koncepta. Na osnovu mog iskustva, i čvrstog uverenja, tvrdim da su ta dva stila potpuno kompatibilni i da dobri programeri mogu, i da bi trebalo, da ih kombinuju.

O „funkcionalnom”

U ovom tekstu koristim termin *funkcionalno*. Definisaću ga i proširiti njegovo značenje. Kako poglavlja budu odmicala, dozvoliću sebi određene slobode u vezi sa tim. Biće primera koji nisu *čisto* funkcionalni, iako su napisani funkcionalnim jezikom i stilom. U većini takvih slučajeva reč *funkcionalno* je pod navodnicima a fusnote ukazuju na slobode koje sam uzeo.

Zašto sam sebi dozvolio tu slobodu? Zato što je ovo knjiga o pragmatičnosti, a ne o teoriji. Više me interesuje da iskoristim prednosti funkcionalnog stila, nego da se strogo pridržavam ideala. Na primer, kao što ćemo videti u prvom poglavlju, „funkcije” koje uzimaju unos od korisnika nisu potpuno funkcionalne. Ipak, koristiću takve „funkcije” kada to bude prikladno.

Sav izvorni kod za primere iz svih poglavlja nalazi se u jednom GitHub spremištu pod nazivom https://github.com/unclebob/FunctionalDesign.

6 Robert C. Martin, *Čista arhitektura* (Pearson 2017), str. 57.

7 Erih Gama, Ričard Helm, Ralf Džonson, Džon Vlisides, *Gotova rešenja – Elementi objektno orijentisanog softvera* (Addison-Wesley, 1994).

Zahvalnosti

Hvala marljivim i profesionalnim ljudima iz izdavačke kuće Pearson, koji su mi pomogli da završim ovu knjigu: Džuli Fajfer, mom dugogodišnjem izdavaču, za neprocenjivu pomoć i podršku; i njenim saradnicima, Menki Meti, Džuli Nahil, Odri Dojl, Morin Foris, Marku Taberu i mnogim drugim. Uvek mi je zadovoljstvo da radim sa vama i radujem se mnogim sličnim poduhvatima u budućnosti.

Hvala Dženifer Konke, koja je kreirala većinu predivnih ilustracija za moje knjige tokom poslednje tri decenije. Još davne 1995. godine, suočeni sa istekom roka, Dženifer, Džim Njukirk i ja proveli smo celu noć u radu na formatiranju i organizaciji ilustracija za moju prvu knjigu, da bi bila upravo onakva kako sam je zamislio.

Hvala Majklu Federsu, koji mi je pre 20 godina predložio da istražim funkcionalno programiranje. On je tada učio Haskell i bio je oduševljen mogućnostima. Njegov entuzijazam je bio zarazan.

Hvala Marku Simanu (@ploeh) na njegovom dosledno pronicljivom radu, oštrim i razorno racionalnim recenzijama mojih radova, kao i moralnoj hrabrosti.

Hvala i Stjuartu Haloveju, autoru prve knjige o jeziku Clojure koju sam pročitao. Ta avantura je počela pre više od deceniju i po, i nikada se nisam osvrnuo. Stjuart je bio dovoljno ljubazan da nadgleda moje prve eksperimente sa funkcionalnim programiranjem. Stjuartu, takođe, dugujem i izvinjenje za davni, ishitreni komentar, izrečen u pogrešnom trenutku.

Hvala Riču Hikiju, s kojim sam početkom 90-ih raspravljao o jeziku C++ i objektno-orijentisanom dizajnu, a koji je, zatim, stvorio jezik Clojure i majstorski upravljao njegovim razvojem. Ričovi uvidi u softver i dalje me oduševljavaju.

Iako ih nikada nisam upoznao, dugujem zahvalnost Haroldu Abelsonu, Džeraldu Džej Sasmanu i Džuli Sasman za knjigu koja me je istinski inspirisala da se posvetim funkcionalnom programiranju. Ta knjiga, *Struktura i interpretacija računarskih programa (SICP)*, imala je najveći uticaj na mene, od svih knjiga o softveru koje sam pročitao. Dostupna je besplatno na internetu. Dovoljno je potražiti „SICP".

Hvala Dženet Kar, za njen predgovor. Jednog dana, pretražujući Tviter, naišao sam na njen rad i otkrio da smo došli do mnogo istih zaključaka o funkcionalnom programiranju i jeziku Clojure.

Za pisanje pogovora, hvala mojoj divnoj kćerki Đini Martini, uspešnom hemijskom i softverskom inženjeru. Više o njoj u mojoj posveti.

O autoru

Robert C. Martin, poznat i kao Ujka Bob, programiranjem se bavi još od 1970. godine. On je osnivač Uncle Bob Consulting-a, LLC, a zajedno sa svojim sinom Mikom Martinom suosnivač firme The Clean Coders, LLC. Martin je objavio desetine radova u različitim stručnim časopisima i redovan je govornik na međunarodnim konferencijama i sajmovima. Autor je i urednik mnogih knjiga, uključujući *Dizajniranje objektno-orijentisanih C++ aplikacija korišćenjem Booch metoda*, *Projektni jezici dizajniranja programa 3*, *Više C++ dragulja*, *Ekstremno programiranje u praksi* , *Agilni razvoj softvera: principi, obrasci i prakse*, *UML za Java programere*, *Čist kod* , *Čisto programiranje* , *Čista arhitektura*, *Čisto majstorstvo* i *Čisto agilno*. Kao lider u industriji razvoja softvera, Martin je tri godine bio glavni urednik časopisa *C++ Report* i bio je prvi predsednik Agile Alliance-a.

Neke od knjiga ujka Boba su prevedene na srpski jezik:
Čista arhitektura, izdavač Kompjuter biblioteka Beograd
Čisto agilno, izdavač Kompjuter biblioteka Beograd
Čisto majstorstvo, izdavač Kompjuter biblioteka Beograd
Jasan kod, izdavač Mikro knjiga Beograd

Svim knjigama koje su prevedene na srpski jezik možete da pristupite preko linka:
https://knjige.kombib.rs/oblasti-knjiga-179-1

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% popustai 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

Osnove funkcionalnog programiranja

1 Nepromenljivost

Šta je funkcionalno programiranje?

Ako pitate prosečnog programera šta je to funkcionalno programiranje, verovatno ćete dobiti neki od sledećih odgovora:

- Programiranje pomoću funkcija.
- Funkcije su elementi „prve klase".
- Programiranje sa referencijalnom transparentnošću.
- Stil programiranja koji se zasniva na lambda računu.

Iako ove tvrdnje mogu biti tačne, nisu ni posebno korisne. Mislim da je bolji odgovor: *programiranje bez naredbi dodeljivanja*.

Možda mislite da ni ovaj odgovor nije mnogo bolji. Možda vas čak i plaši. Konačno, kakve veze naredbe dodeljivanja imaju sa funkcijama; i kako je uopšte moguće programirati bez njih?

To su dobra pitanja. To su pitanja na koja nameravam da odgovorim u ovom poglavlju.

Razmotrite sledeći jednostavan C program:

```
int main(int ac, char** av) {
    while(!done())
        doSomething();
}
```

Ovaj program predstavlja osnovnu petlju gotovo svakog programa koji je ikad napisan. On bukvalno kaže: „Radi nešto dok ne završiš". Štaviše, ovaj program nema vidljivih naredbi dodeljivanja. Da li je to funkcionalno? A ako jeste, da li to znači da je svaki ikada napisan program funkcionalan?

Hajde da učinimo da ova funkcija zapravo nešto i izvršava. Neka izračunava zbir kvadrata prvih deset prirodnih brojeva [1..10]:

```
int n=1;
int sum=0;
```

```
int done() {
  return n>10;
}

void doSomething() {
  sum+=n*n;
  ++n;
}

void sumFirstTenSquares() {
    while(!done())
        doSomething();
}
```

Ovaj program nije funkcionalan jer koristi dve naredbe dodeljivanja u funkciji doSomething. Takođe je i prosto ružan sa te dve globalne promenljive. Hajde da ga poboljšamo:

```
int sumFirstTenSquares() {
  int sum=0;
  int i=1;
loop:
  if (i>10)
    return sum;
  sum+=i*i;
  i++;
  goto loop;
}
```

Ovako je bolje; dve globalne promenljive su postale lokalne. Ali, i dalje nije funkcionalan. Možda ste zabrinuti zbog te goto naredbe. Ona je tu sa dobrim razlogom. Strpite se dok razmatrate ovu malu modifikaciju koja koristi radnu funkciju da pretvori lokalne promenljive u argumente funkcije:

```
int sumFirstTenSquaresHelper(int sum, int i) {
loop:
  if (i>10)
    return sum;
```

```
  sum+=i*i;
  i++;
  goto loop;
}

int sumFirstTenSquares() {
  return sumFirstTenSquaresHelper(0, 1);
}
```

Ovaj program još uvek nije funkcionalan; ali je to važna *prekretnica* koju ćemo uskoro pomenuti. Međutim, uz još jednu promenu, dešava se nešto čarobno:

```
int sumFirstTenSquaresHelper(int sum, int i) {
  if (i>10)
    return sum;
  return sumFirstTenSquaresHelper(sum+i*i, i+1);
}

int sumFirstTenSquares() {
  return sumFirstTenSquaresHelper(0, 1);
}
```

Sve naredbe dodeljivanja su nestale i ovaj program je sada funkcionalan. Takođe je i rekurzivan. To nije slučajnost. Ako želite da se oslobodite naredbi dodeljivanja, *morate* da koristite rekurziju. Rekurzija omogućava da zamenite dodeljivanje lokalnih promenljivih *inicijalizacijom* argumenata funkcije.

Takođe, zauzima puno prostora na steku. Međutim, postoji mali trik kojim možemo da rešimo ovaj problem.

Obratite pažnju na to da je poslednji poziv funkcije `sumFirstTenSquaresHelper` takođe i poslednja upotreba promenljivih `sum` i `i` u toj funkciji. Čuvanje ove dve promenljive na steku nakon inicijalizacije dva argumenta rekurzivnog poziva je besmisleno; nikada više neće biti upotrebljene. A šta ako, umesto kreiranja novog stek okvira za rekurzivan poziv, jednostavno ponovo upotrebimo postojeći stek okvir skokom unazad na vrh funkcije pomoću goto naredbe, kao što smo uradili u programu *prekretnici*?

Ovaj mali, simpatični trik nazivamo *optimizacija repnog poziva (TCO)*, a koriste ga svi funkcionalni jezici.[1]

Obratite pažnju na to da TCO efektivno pretvara taj poslednji program u program *prekretnicu*. Poslednje tri linije funkcije `sumFirstTenSquaresHelper` u programu *prekretnici* su, zapravo, rekurzivni poziv funkcije. Da li to znači da je i program *prekretnica* funkcionalan? Ne, on se samo isto ponaša. Na nivou izvornog koda, taj program nije funkcionalan jer ima naredbe dodeljivanja. Ali ako se malo udaljimo i zanemarimo činjenicu da su se lokalne promenljive promenile, umesto da budu ponovo instancirane u novom stek okviru, onda se program *ponaša* kao funkcionalan program.

Kao što ćemo otkriti u sledećem odeljku, to nije beznačajna razlika. U međuvremenu, zapamtite da to što koristite rekurziju da eliminišete naredbe dodeljivanja, ne mora da znači da trošite puno prostora na steku. Jezik koji koristite skoro sigurno koristi TCO.

Problem sa dodeljivanjem vrednosti

Prvo, definisimo šta podrazumevamo pod *dodeljivanjem*. Dodeljivanje vrednosti promenljivoj *menja* originalnu vrednost te promenljive na novododeljenu vrednost. Promena je to što čini dodeljivanje.

U jeziku C ovako inicijalizujemo promenljivu:

```
int x=0;
```

Ali ovako dodeljujemo vrednost promenljivoj:

```
x=1;
```

1 Na ovaj ili onaj način. *Java virtuelna mašina (JVM)* malo komplikuje TCO. Jezik C, naravno, ne koristi TCO, pa će svi moji rekurzivni primeri na jeziku C povećavati stek.

U prvom slučaju, inicijalna vrednost promenljive `x` je `0`; pre inicijalizacije, promenljiva `x` nije postojala. U drugom slučaju, vrednost `x` se menja na `1`. Ovo možda ne izgleda značajno, ali implikacije su duboke.

U prvom slučaju, ne znamo da li je `x` zapravo promenljiva. Mogla bi biti konstanta. U drugom slučaju, nema sumnje. Mi menjamo `x` dodeljivanjem nove vrednosti. Stoga, možemo reći da je funkcionalno programiranje programiranje *bez promenljivih*. Vrednosti u funkcionalnim programima se *ne menjaju*.

Zašto je to poželjno? Razmotrimo sledeće:

```
.
//Blok A
.
x=1;
.
//Blok B
.
```

Stanje sistema tokom izvršavanja bloka A razlikuje se od stanja sistema u bloku B. To znači da blok A mora biti izvršen *pre* bloka B. Ako bi ova dva bloka zamenila mesta, sistem verovatno ne bi ispravno funkcionisao.

Ovo nazivamo *sekvencijalno* ili *privremeno sprezanje* - sprezanje u vremenu; a to je nešto sa čim ste, verovatno, već upoznati. `Open` mora biti pozvan pre `close`. `New` mora biti pozvan pre `delete`. `Malloc` mora biti pozvan pre `free`. Ovakva lista parova[2] je beskrajna. I u mnogim aspektima, oni su naša noćna mora.

Koliko ste puta zaboravili da zatvorite datoteku, oslobodite blok memorije, zatvorite grafički kontekst, ili oslobodite semafor? Koliko puta ste ispravljali neki opaki problem i na kraju otkrili da ga možete rešiti zamenom mesta dva poziva funkcije?

2 Oni su kao Siti; uvek dva ih je.

A tu je i skupljanje otpada.

Skupljanje otpada je užasan[3] hakovani pristup koji smo prihvatili u našim jezicima jer smo jednostavno veoma loši u upravljanju privremenim sprezanjima. Da smo vešti u praćenju alocirane memorije, ne bismo zavisili od nekog neprijatnog pozadinskog procesa da čisti za nama. Ali tužna činjenica je da smo zaista užasni u upravljanju privremenim sprezanjima do te mere da slavimo improvizovana rešenja koja smo stvorili da se zaštitimo od njih.

A to ne uzima u obzir višestruke niti. Kada se dve ili više niti takmiči za procesor, očuvanje ispravnog redosleda privremenih sprezanja postaje znatno veći izazov. Te niti će verovatno održavati ispravan redosled 99,99% vremena; ali jednom će se izvršiti pogrešnim redosledom i uzrokovati sveopšti haos. Te situacije nazivamo *uslovima trke*.

Privremena sprezanja i uslovi trke su prirodna posledica programiranja sa promenljivama - korišćenjem dodeljivanja. Bez dodeljivanja, ne postoje ni privremena sprezanja ni uslovi trke.[4] Ne možete imati problem konkurentnog ažuriranja ako nikada ništa ne ažurirate. Ne možete imati problem redosleda unutar funkcije ako se stanje sistema unutar te funkcije nikada ne menja.

Ali, možda je vreme za jednostavan primer. Evo opet našeg nefunkcionalnog algoritma; ovog puta bez goto naredbe:

```
1: int sumFirstTenSquaresHelper(int sum, int i) {
2:   while (i<=10) {
3:     sum+=i*i;
4:     i++;
5:   }
6:   return sum;
7: }
```

3 I ne, brojanje referenci nije ništa bolje.

4 Kasnije ćemo videti da ovo nije potpuno tačno. Kao što je Spok često govorio: „Uvek postoje opcije."

Pretpostavimo sada da želite da zabeležite napredak algoritma naredbom poput ove:

```
log("i=%d, zbir=%d", i, sum);
```

Gde biste postavili ovu liniju? Postoje tri mogućnosti. Ako dodate `log` naredbu posle linije 2 ili 4, onda će evidentirani podaci biti tačni, a razlika će biti samo u tome da li evidentirate pre ili posle izračunavanja. Ako ubacite `log` naredbu posle linije 3, evidentirani podaci će biti netačni. To je privremeno sprezanje - problem redosleda.

Sada razmotrite naše funkcionalno rešenje, sa jednom zanimljivom kozmetičkom promenom:

```
int sumFirstTenSquaresHelper(int sum, int i) {
  return (i>10) ? sum : sumFirstTenSquaresHelper(sum+i*i, i+1);
}
```

Postoji samo jedno mesto gde možemo da postavimo našu `log` naredbu i ona će evidentirati tačne podatke.

Pa zašto ga nazivamo funkcionalnim?

Funkcija je matematički objekat koji preslikava ulaze u izlaze. Ako je $y = f(x)$, tada postoji vrednost y za svaku vrednost x. Ništa drugo nije bitno za f. Ako date x funkciji f, svaki put ćete dobiti y. Stanje sistema u kojem se f izvršava je nebitno za f.

Ili drugim rečima, ne postoje privremena sprezanja sa f. Ne postoji poseban redosled po kom f mora biti pozvana. Ako pozovete f sa x, dobićete y bez obzira na sve što se možda promenilo.

Funkcionalni programi su prave funkcije u ovom matematičkom smislu. Ako rastavite funkcionalni program na mnogo manjih funkcija, svaka od njih će takođe biti prava funkcija u istom matematičkom smislu. To nazivamo *referencijalna transparentnost*.

Funkcija je referencijalno transparentna ako uvek možete da zamenite poziv funkcije njenom vrednošću. Hajde da probamo to sa našim funkcionalnim algoritmom za izračunavanje zbira kvadrata prvih deset prirodnih brojeva:

```
int sumFirstTenSquaresHelper(int sum, int i) {
  return (i>10) ? sum : sumFirstTenSquaresHelper(sum+i*i, i+1);
}

int sumFirstTenSquares() {
  return sumFirstTenSquaresHelper(0, 1);
}
```

Kada zamenimo prvi poziv `sumFirstTenSquaresHelper` njegovom implementacijom, to postaje:

```
int sumFirstTenSquares() {
  return (1>10) ? 0 : sumFirstTenSquaresHelper(0+1*1, 1+1);
}
```

Kada zamenimo sledeći poziv funkcije, to postaje:

```
int sumFirstTenSquares() {
  return
    (1>10) ? 0 :
      (2>10) ? 0+1*1
             : sumFirstTenSquaresHelper((0+1*1)+2*2,
                                        (1+1)+1);
}
```

Mislim da vidite gde to vodi. Svaki poziv `sumFirstTenSquaresHelper` jednostavno je zamenjen svojom implementacijom sa pravilno zamenjenim argumentima.

Obratite pažnju na to da ovu jednostavnu zamenu ne možete da izvršite sa nefunkcionalnom verzijom programa. Možete, naravno, da razmotate petlju, ako želite; ali to nije isto kao jednostavna zamena svakog poziva funkcije njenom implementacijom.

Zbog toga su funkcionalni programi sastavljeni od pravih matematičkih, referencijalno transparentnih funkcija. I zbog toga ga nazivamo funkcionalno programiranje.

Bez promene stanja?

Ako u funkcionalnim programima ne postoje promenljive, onda funkcionalni programi ne mogu da menjaju stanje. Kako možemo da očekujemo da program bude koristan ako ne može da menja stanje?

Odgovor je da funkcionalni programi izračunavaju novo stanje iz starog stanja, *bez promene starog stanja*. Ako vas ovo zbunjuje, sledeći primer bi trebalo sve da razjasni:

```
State system(State s) {
  return isFinal(s) ? s : system(s);
}
```

Možete da pokrenete funkciju `system` u nekom početnom stanju, i ona će se postepeno premeštati iz stanja u stanje dok ne dostigne krajnje stanje. Funkcija `system` ne menja promenljivu stanja. Umesto toga, u svakom ponavljanju, novo stanje je kreirano iz starog stanja.

Ako isključimo TCO i dozvolimo steku da raste sa svakim rekurzivnim pozivom, stek će sadržati sva prethodna stanja, nepromenjena. Štaviše, funkcija `system` funkcioniše kao prava funkcija u matematičkom smislu. Ako pozovete funkciju `system` sa `state1`, ona će uvek vratiti `state2`.

Ako pažljivo pogledate našu funkcionalnu verziju `sumFirstTenSquares`, videćete da koristi upravo ovaj pristup za promenu stanja. Ne postoje promenljive i nema unutrašnjeg stanja. Umesto toga, algoritam se pomera od početnog stanja do krajnjeg stanja, menjajući stanje po stanje.

Naravno, funkcija `system` izgleda kao da ne može da odgovori na bilo koji ulaz. Ona jednostavno počinje od nekog stanja i zatim se izvršava do kraja. Ali, jednostavnom modifikacijom možemo da stvorimo „funkcionalan" program koji prilično lepo reaguje na ulazne događaje:

```
State system(State state, Event event) {
  return done(state) ? state : system(state, getEvent());
}
```

Sada je izračunato sledeće stanje funkcije `system` funkcija trenutnog stanja i dolaznog događaja. I eto! Stvorili smo veoma tradicionalan konačan automat koji može da reaguje na događaje u realnom vremenu.

Obratite pažnju na navodnike pod koje sam stavio reč *funkcionalan*. To je zato što `getEvent` nije referencijalno transparentan. Svaki put kada ga pozovete, dobićete drugačiji rezultat. Dakle, ne možete da zamenite poziv njegovom povratnom vrednošću. Da li to znači da naš program zapravo nije funkcionalan? Strogo govoreći, svaki program koji prima ulaze na ovaj način ne može biti potpuno funkcionalan. Ali ovo nije knjiga o potpuno funkcionalnim programima. Ovo je knjiga o funkcionalnom *programiranju*. Stil programa iznad je „funkcionalan", čak i ako ulaz nije čist; a to je stil koji nas ovde zanima.

A sada, radi zabave, evo jednostavnog malog konačnog automata u realnom vremenu napisanog na jeziku C, a koji je „funkcionalan". To je čuveni primer za rotirajuće kapije u metrou. Uživajte.

```
#include <stdio.h>

typedef enum {locked, unlocked, done} State;
typedef enum {coin, pass, quit} Event;

void lock() {
  printf("Zaključavanje.\n");
}
```

```
void unlock() {
  printf("Otključavanje.\n");
}

void thankyou() {
  printf("Hvala.\n");
}

void alarm() {
  printf("Alarmiranje.\n");
}

Event getEvent() {
  while (1) {
    int c = getchar();
    switch (c) {
      case 'c': return coin;
      case 'p': return pass;
      case 'q': return quit;
    }
  }
}

State turnstileFSM(State s, Event e) {
  switch (s) {
    case locked:
    switch (e) {
      case coin:
      unlock();
      return unlocked;

      case pass:
      alarm();
      return locked;

      case quit:
      return done;
    }
```

```
    case unlocked:
    switch (e) {
      case coin:
      thankyou();
      return unlocked;

      case pass:
      lock();
      return locked;

      case quit:
      return done;
    }
    case done:
    return done;
  }
}

State turnstileSystem(State s) {
  return (s==done)? 0
                  : turnstileSystem(
                      turnstileFSM(s, getEvent()));
}

int main(int ac, char** av) {
  turnstileSystem(locked);
  return 0;
}
```

Ne zaboravite da C ne koristi TCO, pa će stek rasti dok se u potpunosti ne iscrpi - mada bi za to, u ovom slučaju, bilo potrebno prilično veliki broj operacija.

Nepromenljivost

Sve ovo znači da funkcionalni programi ne sadrže promenljive. Ništa u funkcionalnom programu ne menja stanje. Promene stanja se prenose sa jednog poziva rekurzivne funkcije na sledeći, bez izmene bilo kog od prethodnih stanja. Ako prethodna stanja nisu potrebna, TCO može da optimizuje njihovo uklanjanje; ali u suštini, sva ta stanja i dalje postoje, nepromenjena, negde u prethodnom stek okviru.

Ako u funkcionalnom programu ne postoje promenljive, onda su vrednosti koje imenujemo sve konstante. Jednom inicijalizovane, te konstante nikada ne nestaju i nikada se ne menjaju. U suštini, cela istorija svake od tih konstanti ostaje netaknuta, nepromenjena i nepromenljiva.