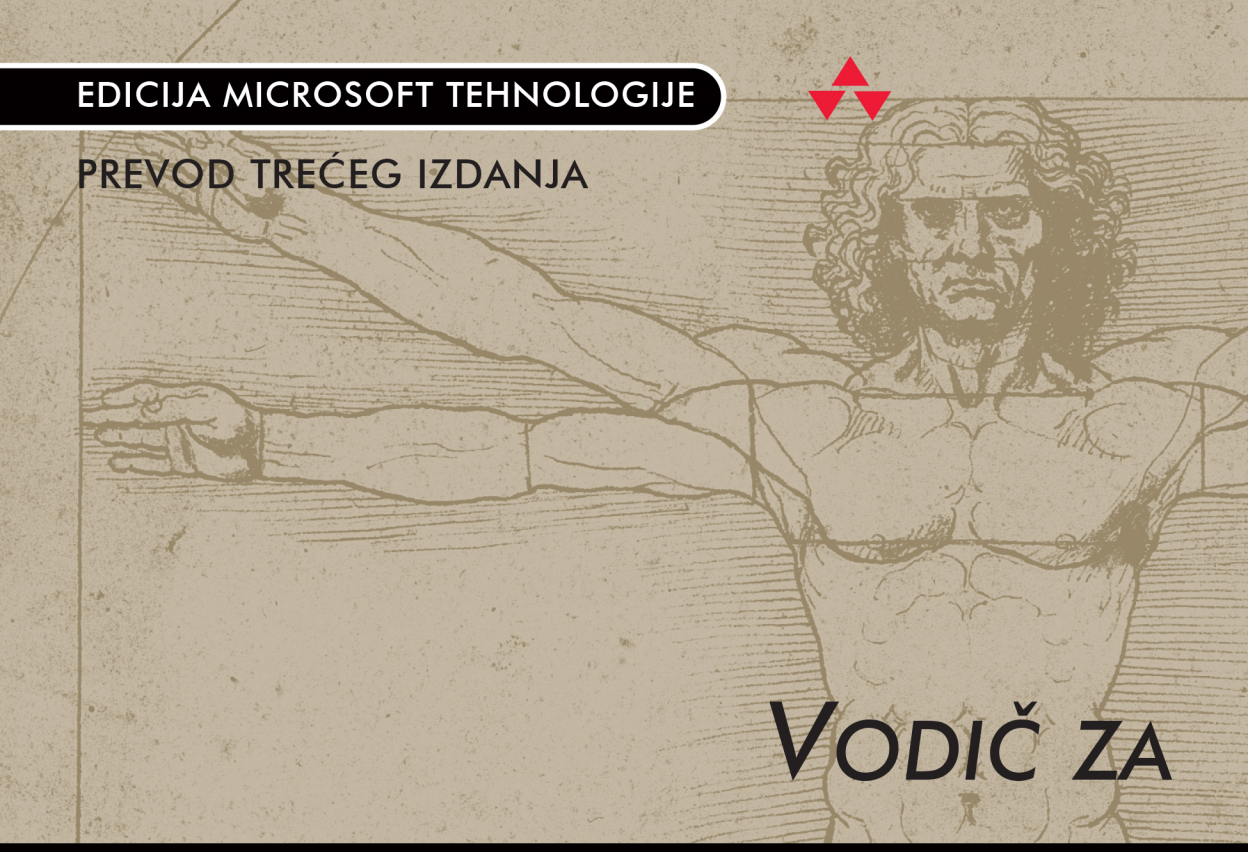
EDICIJA MICROSOFT TEHNOLOGIJE

PREVOD TREĆEG IZDANJA



VODIČ ZA

DIZAJNIRANJE FREJMVORKA

KONVENCIJE, IDIOMI I OBRASCI ZA VIŠEKRATNE .NET BIBLIOTEKE

KRZYSZTOF CWALINA
JEREMY BARTON
BRAD ABRAMS

kompjuter biblioteka

Pisci predgovora Scott Guthrie
Miguel de Icaza i Anders Hejlsberg

Vodič za

DIZAJNIRANJE FREJMVORKA

KONVENCIJE, IDIOMI I OBRASCI ZA VIŠEKRATNE .NET BIBLIOTEKE

Prevod III izdanja

- Krzysztof Cwalina
- Jeremy Barton
- Brad Abrams

kompjuter biblioteka

Addison-Wesley

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

Autori: Krzysztof Cwalina
Jeremy Barton
Brad Abrams

Prevod: Slavica Prudkov

Lektura: Nemanja Lukić

Slog: Zvonko Aleksić

Znak Kompjuter biblioteke:
Miloš Milosavljević

Štampa: „Pekograf“, Zemun

Tiraž: 500

Godina izdanja: 2021.

Broj knjige: 536

Izdanje: Prvo

ISBN: 978-86-7310-559-8

Framework Design Guidelines
Third Edition

Krzysztof Cwalina
Jeremy Barton
Brad Abrams

ISBN 978-0-13-589646-4

Slike

Tabele

Predgovor drugom izdanju

Kada smo projektovali .NET platformu želeli smo da ona bude najproduktivnija platforma za razvoj poslovnih aplikacija. Pre dvadeset godina, to je značilo klijentsko-serverske aplikacije hostovane na namenskom hardveru.

Danas se nalazimo usred jedne od najvećih promena paradigme u industriji: prelaz na *cloud* računarstvo. Takve transformacije donose nove mogućnosti za preduzeća, ali donose i poteškoće postojećim platformama koje, uglavnom, moraju da se prilagođavaju drugačijim zahtevima, nametnutim novom vrstom aplikacija koje programeri žele da pišu.

.NET platforma je prebačena prilično uspešno, a mislim da je jedan od glavnih razloga za to taj što smo je projektovali pažljivo i namerno smo se, pored produktivnosti, konzistentnosti i jednostavnosti, fokusirali i na mogućnost evolucije. .NET Core predstavlja tu evoluciju sa napretkom važnim za programere cloud aplikacija: performans, iskorišćenost resursa, podrška za kontejnere i ostalo.

U ovo, treće izdanje knjige Saveti za projektovanje radnih okvira dodati su saveti koji se odnose na promene koje je .NET tim usvojio tokom prelaza iz sveta klijentsko-serverske aplikacije u *cloud* svet.

- Scott Guthrie

Redmond, WA

January 2020

Predgovor drugom izdanju

Kada je .NET Framework prvi put publikovan, ja sam bio fasciniran tehnologijom. Prednosti CLR-a (Common Language Runtime), njegovi opširni API-ji i C# jezik su odmah postali očigledni. Ali, ispod te tehnologije je bio uobičajen dizajn API-ja i skup konvencija koje su svuda korišćene. To je bila .NET kultura. Kada ste naučili jedan njen deo, bilo je veoma jednostavno prevesti to znanje na druge oblasti radnog okvira.

U proteklih 16 godina radio sam na softveru otvorenog koda. Pošto saradnici dolaze iz različitih disciplina, a saradnja se proteže duži niz godina, upotreba istog stila i konvencija kodiranja uvek je bila veoma važna. Programeri koji održavaju softver rutinski prepisuju ili prilagođavaju unapređenja softvera da bi obezbedili da se kod pridržava standarda i stila kodiranja projekta. Uvek je bolje kada saradnici i ljudi koji se pridružuju softverskom projektu prate konvencije koje su upotrebljene u postojećem projektu. Što više informacija može da se prenese kroz praksu i standarde, to će budućim saradnicima biti jednostavnije da se prilagode projektu. To pomaže projektu da konvergira kod, i stari i novi.

Pošto su i .NET Framework i zajednica programera porasli, identifikovana je nova praksa, obrasci i konvencije. Brad i Krzysztof su postali kustosi, koji su pretvorili svo to novo znanje u aktuelni vodič. Oni održavaju blog na kom govore o novim konvencijama, traže povratne informacije od zajednice i prate te smernice. Po mom mišljenju, njihovi blogovi su dokumenti koje

bi trebalo da pročitaju svi oni koji su zainteresovani da dobiju maksimum iz .NET Framework-a.

Prvo izdanje knjige Smernice za projektovanje radnog okvira odmah je postalo popularno u Mono zajednici iz dva vredna razloga. Prvo, obezbeđuje sredstva za razumevanje zašto i kako su različiti .NET API-ji implementirani. Drugo, cenimo ovu knjigu zbog neprocenjivih smernica koje i mi težimo da pratimo u sopstvenim programima i bibliotekama. Ovo novo izdanje nadograđuje uspeh prvog izdanja, ali je i ažurirano novim lekcijama koje su od tada naučene. Objašnjenja obezbeđuju neki od vodećih .NET arhitekata i odličnih programera, koji su pomogli da se oblikuju ove konvencije.

Ovaj tekst je mnogo više od vodiča. To je knjiga koju ćete ceniti kao „klasik", jer vam je pomogla da postanete bolji programer, a takvih je svega nekoliko u našoj industriji.

- Miguel de Icaza

Boston, MA

Oktobar 2008.

Predgovor prvom izdanju

U ranim danima razvoja .NET Framework-a, pre nego što je i nazvan tako, proveo sam mnogo vremena sa članovima razvojnog tima, pregledajući dizajn da bismo bili sigurni da će krajnji rezultat biti koherentna platforma. Uvek sam osećao da ključna karakteristika radnog okvira mora da bude konzistentnost. Kada razumete jedan deo radnog okvira, drugi delovi bi odmah trebalo da postanu poznati.

Kao što i možete očekivati od velikog tima pametnih ljudi, naša mišljenja su se mnogo razlikovala – ništa ne pokreće toliko žive i žestoke rasprave kao što to čine konvencije kodiranja. Međutim, u ime konzistentnosti, postepeno smo uskladili različita mišljenja i kodifikovali rezultat u zajednički skup smernica koje omogućavaju programerima lako razumevanje i upotrebu radnog okvira.

Brad Abrams i Krzysztof Cwalina pomogli su da ove savete postavimo u živi dokument koji je u kontinuitetu ažuriran i redefinisan poslednjih šest godina. Knjiga koju držite u rukama je rezultat njihovog rada.

Smernice su nam dobro služile kroz tri verzije .NET Frameworka i brojnim manjim projektima i one vode razvoj sledeće generacije API-ja za Microsoft Windows operativni sistem.

Nadam se i očekujem da ćete pomoću ove knjige i vi biti uspešni u kreiranju radnih okvira, biblioteka klasa i komponenti jednostavnih za razumevanje i upotrebu.

Sa srećom i prijatno projektovanje.

- Anders Hejlsberg

Redmond, WA

Jun 2005.

Uvod

Ova knjiga, „*Vodič za dizajniranje frejmvorka*“, predstavlja najbolju praksu projektovanja radnih okvira, koji su višekratne objektno-orijentisane biblioteke. Smernice su primenjive na radne okvire različitih veličina i razmera ponovne upotrebe, uključujući:

- Velike sistemske radne okvire, kao što su osnovne biblioteke u .NET-u, koje se obično sastoje od više hiljada tipova i koje koriste milioni programera.
- Višekratni slojevi srednje veličine velikih distribuiranih aplikacija ili ekstenzija sistemskih radnih okvira, kao što su Azure SDK ili softveri za igre.
- Male komponente deljene između nekoliko aplikacija, kao što su biblioteke za kontrolu rasporeda elemenata.

Vredi istaći da se ova knjiga fokusira na pitanja dizajna koja direktno utiču na programibilnost radnog okvira (javno dostupni API-i[1]). Kao rezultat, generalno nećemo obuhvatiti mnogo detalja implementacije. Isto kao što knjiga o dizajnu korisničkog interfejsa ne obuhvata detalje kako da implementirate testiranje poseta, tako ova knjiga ne opisuje kako da implementirate binarno sortiranje, na primer. Ovaj obim nam omogućava da obezbedimo definitivan vodič za dizajnere radnog okvira, umesto da bude još jedna knjiga o programiranju. U knjizi pretpostavljamo da čitalac već ima osnovno znanje o programiranju u .NET-u.

1 Ovo uključuje javne tipove i javne, zaštićene i eksplicitno implementirane članove ovih tipova.

Ovi saveti su kreirani u ranim danima razvoja .NET Framework-a. Počelo je kao mali skup konvencija imenovanja i dizajna ali su poboljšani, detaljno proučeni i prerađeni, do tačke da ih generalno smatramo glavnim načinom projektovanja radnih okvira u Microsoft-u. Saveti obuhvataju iskustvo i mudrost stečene tokom hiljada programerskih sati, tokom dve decenije .NET-a. Pokušali smo da izbegnemo baziranje teksta isključivo na nekim idealističkim filozofijama projektovanja i mislimo da je svakodnevna upotreba od strane razvojnog tima u Microsoft-u učinila ovu knjigu pragmatičnom.

Ova knjiga sadrži mnoga objašnjenja kompromisa, pojašnjava istoriju, dopunjava ili daje kritički pogled na savete. Ova objašnjenja su pisali iskusni dizajneri radnih okvira, industrijski eksperti i korisnici. Oni su glavni akteri priče, koji joj daju boju i atmosferu.

Da biste ih jednostavnije razlikovali u tekstu, nazivi imenskih prostora, klase, interfejsi, metodi, svojstva i tipovi su napisani monospace fontom.

Predstavljanje saveta

Saveti su organizovani kao jednostavne preporuke upotrebom odeljaka **URADITE**, **RAZMOTRITE**, **IZBEGAVAJTE** i **NE RADITE**. Svaki savet opisuje dobru ili lošu praksu i svi imaju konzistentnu prezentaciju. Ispred dobre prakse se nalazi (oznaka potvrde), a ispred loše prakse se nalazi X. Formulacija svakog saveta takođe ukazujue na to koliko je preporuka jaka. Na primer, savet **URADITE** je onaj koji bi uvek trebalo da poslušate (svi primeri su iz ove knjige):

✓ **URADITE** Imenujte prilagođene klase atributa pomoću sufiksa „Attribute”. Imenujte prilagođene klase atributa pomoću sufiksa „Attribute”

```
public class ObsoleteAttribute : Attribute { ... }
```

Sa druge strane, savete **RAZMOTRITE** bi generalno trebalo da prihvatate, ali ako u potpunosti razumete razloge saveta i imate dobar razlog da ga ipak ne prihvatite, nemojte se osećati loše zbog kršenja pravila:

✓ **RAZMOTRITE** definisanje strukture umesto klase, ako su instance tipa male i uobičajeno kratkog veka, ili su uobičajeno ugrađene u druge objekte.

Slično, saveti **NE RADITE** ukazuju na nešto što skoro nikada ne bi trebalo da radite:

✓ **NE RADITE** – ne obezbeđujte svojstva samo za skup ili svojstva gde metod zadavanja vrednosti ima širu dostupnost od metoda za učitavanje vrednosti.

Manje strogi saveti **IZBEGAVAJTE** ukazuju na nešto što generalno nije dobra ideja, ali postoje poznati slučajevi u kojima kršenje datog pravila ima smisla:

✓ **IZBEGAVAJTE** upotrebu `ICollection<T>` ili `ICollection` kao parametra samo da biste pristupili `Count` svojstvu.

Neki kompleksniji saveti su praćeni dodatnim informacijama, ilustrativnim primerima koda i obrazloženjima:

✓ **URADITE** – implementirajte `IEquatable<T>` u tipovima vrednosti.

Metod `Object.Equals` u tipovima vrednosti dovodi do pakovanja i njegova podrazumevana implementacija nije veoma efikasna, zato što koristi refleksiju. `IEquatable<T>.Equals` metod može da obezbedi mnogo bolji performans i može da se implementira tako da ne izaziva pakovanje.

```
public struct Int32 : IEquatable<Int32> {
   public bool Equals(Int32 other){ ... }
}
```

Izbor jezika i primeri koda

Jedan od ciljeva Common Language Runtime-a (CLR-a) je da podržava više programskih jezika: onih sa implementacijama koje obebzeđuje Microsoft, kao što su C++, VB, C#, F#, IronPython i PowerShell, kao i drugih jezika, kao što su Eiffel, COBOL, Fortran i drugi. Prema tome, ova knjiga je pisana tako da bude primenjiva za širok skup jezika koji mogu da se upotrebe za razvoj i upotrebu modernih radnih okvira.

Da bismo pojačali poruku višejezičnog projektovanja radnih okvira, razmotrili smo pisanje primera koda upotrebom nekoliko različitih programskih jezika. Međutim, odlučili smo da ipak nećemo to raditi. Osećali smo da bi upotreba različitih jezika pomogla da prenesemo filozofsku poruku, ali bi nametnula čitaocima da uče nekoliko novih jezika, a to nije cilj ove knjige.

Odlučili smo da izaberemo jedan jezik koji će biti čitljiv najširem rasponu programera. Izabrali smo C#, zato što je to jednostavan jezik, a C familija jezika (C, C++, Java i C#), je familija sa bogatom istorijom razvoja radnih okvira.

Izbor jezika je blizak mnogim programerima, a izvinjavamo se onima kojima naš izbor ne odgovara.

O ovoj knjizi

U ovoj knjizi dati su saveti za projektovanje radnih okvira od vrha nadole.

Poglavlje 1, „Uvod“ – kratak pregled knjige, opisana je osnovna filozofija projektovanja radnog okvira. Ovo je jedino poglavlje bez saveta.

Poglavlje 2, „Osnove projektovanja radnog okvira“ – sadrži principe i savete koji su osnova za uopšteno projektovanje radnih okvira.

Poglavlje 3, „Saveti za imenovanje“ – sadrži uobičajene idiome projektovanja i savete za imenovanje za različite delove radnog okvira, kao što su imenski prostori, tipovi i članovi.

Poglavlje 4, „Saveti za projektovanje tipa“ – sadrži savete za osnovno projektovanje tipova.

Poglavlje 5, „Projektovanje člana“ – odlazi korak dalje i predstavlja savete za projektovanje članova tipova.

Poglavlje 6, „Projektovanje za proširivost“ – predstavljena su pitanja i saveti koji su važni za obezbeđivanje odgovarajuće proširivosti radnog okvira.

Poglavlje 7, „Izuzeci“ – predstavljeni su saveti za upotrebu izuzetaka, omiljenog mehanizma za izveštavanje o graškama.

Poglavlje 8, „Saveti za upotrebu“ – sadrži savete za proširenje i upotrebu tipova koji se obično pojavljuju u radnim okvirima.

Poglavlje 9, „Uobičajeni projektni obrasci“ – sadrži savete i primere uobičajenih projektnih obrazaca.

Dodatak A, „Konvencije stila kodiranja C# jezika“ – opisane su konvencije kodiranja koje je koristio tim koji proizvodi i održava osnovne biblioteke u .NET-u.

Dodatak B, „Zastareli saveti“ – sadrži savete iz prethodnih izdanja ove knjige koji se odnose na funkcije ili koncepte koji više nisu preporučljivi.

Dodatak C, „Primer API specifikacije“ – primer API specifikacije koju dizajneri radnog okvira unutar Microsoft-a kreiraju kada dizajniraju API-e.

Dodatak D, „Promene“ – opisuje različite vrste promena koje mogu negativno da utiču na korisnike iz jedne verzije u sledećoj verziji.

Registrujte kopiju Saveti za projektovanje radnih okvira, treće izdanje, na InformIT sajtu da biste imali pristup ažuriranjima i/ili ispravkama, kada su one dostupne. Da biste počeli proces registracije, otvorite stranicu informit.com/register i prijavite se, ili kreirajte nalog. Unesite ISBN (9780135896464) proizvoda i kliknite na Submit. Pogledajte karticu Registered Products i potražite link Access Bonus Content pored ovog proizvoda, a zatim pratite dati link da biste pristupili dostupnom dodatnom materijalu. Ako želite da budete obavešteni o ekskluzivnim ponudama, o novim izdanjima i ažuriranjima, potvrdite odobrenje za primanje email-ova od nas.

Priznanja

Ova knjiga je, po svojoj prirodi, prikupljena mudrost više stotina ljudi i mi smo im veoma zahvalni.

Mnogo ljudi unutar Microsoft-a radilo je dugo i vredno tokom godina, predlažući, raspravljajući i na kraju, pišući mnoge od ovih saveta. Iako je nemoguće imenovati svakog ko je bio uključen, nekolicina zaslužuje posebnu zahvalnost: Chris Anderson, Erik Christensen, Jason Clark, Joe Duffy, Patrick Dussud, Anders Hejlsberg, Jim Miller, Michael Murray, Lance Olson, Eric Gunnerson, Dare Obasanjo, Steve Starck, Kit George, Mike Hillberg, Greg Schecter, Mark Boulter, Asad Jawahar, Justin Van Patten i Mircea Trofin.

Takođe želimo da zahvalimo recenzentima: Marc Alcazar, Chris Anderson, Christopher Brumme, Pablo Castro, Jason Clark, Steven Clarke, Joe Duffy, Patrick Dussud, Kit George, Jan Gray, Brian Grunkemeyer, Eric Gunnerson, Phil Haack, Anders Hejlsberg, Jan Kotas, Immo Landwerth, Rico Mariani, Anthony Moore, Vance Morrison, Christophe Nassare, Dare Obasanjo, Brian Pepin, Jon Pincus, Jeff Prosise, Brent Rector, Jeffrey Richter, Greg Schechter, Chris Sells, Steve Starck, Herb Sutter, Clemens Szyperski, Stephen Toub, Mircea Trofin i Paul Vick. Njihov uvid obezbedio je potrebne komentare, boju, humor i istoriju, koji daju ogromnu vrednost ovoj knjizi.

Za svu pomoć, recenzije i podršku, tehničku i moralnu, zahvaljujemo se Martin-u Heller-u i Stephen-u Toub-u. Zahvalnost su zaslužili i Pierre Nallet, George Byrkit, Khristof Falk, Paul Besley, Bill Wagner i Peter Winkler, za svoje vredne i korisne komentare.

Takođe želimo posebno da zahvalimo zahvalimo Susann Ragsdale, koja je ovu knjigu polu-nasumične kolekcije nepovezanih misli pretvorila u besprekornu prozu. Njen besprekorni stil pisanja, strpljenje i fantastičan smisao za humor učinili su proces pisanja ove knjige mnogo jednostavnijim.

O autorima

Krzysztof Cwalina je softverski arhitekta u Microsoft-u. Bio je član-osnivač .NET Framework tima i u svojoj karijeri je projektovao mnoge .NET API-e. Trenutno pomaže različitim timovima u Microsoft-u u projektovanju višekratnih API-a za mnogo različitih programskih jezika. Krzysztof je diplomirao i završio master studije računarske nauke na Univerzitetu u Ajovi.

Jeremy Barton je inženjer u .NET Core Libraries timu. Nakon decenije projektovanja i razvoja malih radnih okvira u C#-u, pridružio se .NET timu 2015. godine, da bi pomogao da kriptografski tipovi funkcionišu na svim platformama u, tada novom, .NET Core projektu. Jeremy je diplomirao računarske nauke i matematiku na tehnološkom institutu Rose-Hulman.

Brad Abrams je član-osnivač Common Language Runtime i .NET Framework timova u Microsoft korporaciji. On je projektovao delove .NET Framework-a od 1998. godine. Brad je počeo karijeru projektovanja radnih okvira izgradnjom biblioteke Base Class Library (BCL), koja se isporučuje kao osnovni deo .NET Framework-a. Brad je, takođe, bio glavni urednik za Common Language Specification (CLS), za .NET Framework Design Guidelines i za biblioteke u ECMA/ISO CLI Standard-u. Brad je autor i koautor više publikacija, uključujući Programming in the .NET Environment i .NET Framework Standard Library Annotated Reference, Volume 1 i 2. Brad je diplomirao računarske nauke na Univerzitetu Severne Karoline. Možete

da pronađete njegova najnovija razmišljanja na njegovom blogu, na adresi http://blogs.msdn.com/BradA. Brad je trenutno Group Product Manager u Google-u, gde radi na novim projektima za Google Assistant.

O recenzentima

Mark Alcazar je u Microsoft-u poslednje 23 godine, gde je proveo većinu karijere u UI radnim okvirima i uglastim zagradama. Mark je radio na Internet Explorer-u, WPF-u, Silverlight-u i na poslednjih nekoliko izdanja Windows programerske platforme. Mark voli snoubording, jedrenje i kuvanje. Diplomirao je na Univerzitetu West Indies, a master studije je pohađao na Univerzitetu Pensilvanije. Živi u Sijetlu sa suprugom i dvoje dece.

Chris Anderson je radio za Microsoft 22 godine na različitim projektima, ali se specijalizovao za projektovanje i arhitekturu .NET tehnologija koje se koriste za implementiranje sledeće generacije aplikacija i usluga. Chris je napisao brojne članke i predstavljen je na brojnim konferencijama (na primer, Microsoft Professional Developers Conference, Microsoft TechEd, WinDev, DevCon) širom sveta, na kojima je bio glavni govornik. Ima veoma popularan blog na adresi www.simplegeek.com.

Christopher Brumme se pridružio Microsoft-u 1997. godine, kada je formiran Common Language Runtime (CLR) tim. Od tada je sarađivao na izvršenju mehaničkih delova osnove koda i dizajna. Trenutno je fokusiran na pitanja konkurentnosti u upravljanom kodu. Pre pridruživanja CLR timu, Chris je bio arhitekta u Borland-u i Oracle-u.

Pablo Castro je istaknuti inženjer u Microsoft-u. Trenutno je deo Azure Data grupe, gde je direktor inženjeringa za Azure Synapse/SQL i Azure Cognitive Search. Pre njegove aktuelne uloge Pablo je bio direktor inženjeringa i nauke o podacima za Applied AI grupu u Azure-u, a pre toga je radio na više proizvoda unutar grupe sistema baze podataka, uključujući SQL Server, .NET, Entity Framework/LINQ i OData.

Jason Clark radi kao softverski arhitekta za Microsoft. Njegova zasluga u Microsoft softverskom inženjeringu uključuje tri verzije Windowsa, tri izdanja .NET Frameworka i WCF. Godine 2000. je publikovao svoju prvu knjigu o softverskom razvoju i nastavio saradnju sa časopisima i drugim publikacijama. Trenutno je odgovoran za Visual Studio Team System Database Edition. Jason-ova jedina druga strast su njegova supruga i deca, sa kojima srećno živi u oblasti Sijetla.

Steven Clarke je iskusni istraživač korisničkog doživljaja u Microsoft Developer Division-u od 1999. godine. Njegovi glavni interesi su posmatranje, razumevanje i modelovanje doživljaja koje programeri imaju sa API-ima i, na taj način, pomaže dizajn API-a koji obezbeđuje optimalno iskustvo svojim korisnicima.

Joe Duffy je osnivač i CEO Pulumi-a, start-up preduzeća iz Sijetla koje programerima i timovima za infrastrukturu daje super moć cloud-a. Pre osnivanja Pulumi-a 2017. godine, Joe je imao vodeću ulogu u Microsoft Develeper Division-u, Operating System Group-u i Microsoft Research-u. U poslednje vreme, Joe je bio direktor za inženjering i tehničku strategiju za Microsoft-ove programerske alate i vodio ključne inicijative tehničke arhitekture, uz upravljanje grupama koje grade C#, C++, Visual Basic i F# jezike, kao i IoT i Visual Studio IDE, kompajler i usluge statičke analize. Joe je imao ključnu ulogu u Microsoft-ovoj transformaciji otvorenog koda i sastavio je početni tim koji je napravio .NET otvoreni kod i nove platforme. Joe ima više od 20 godina profesionalnog softverskog iskustva, napisao je dve knjige, a i dalje voli da kodira.

Patrick Dussud je tehnički kolega u Microsoft-u, gde radi kao glavni arhitekta u arhitekturnim grupama za CLR i .NET Framework. Radi na pitanjima .NET Frameworka u kompaniji, pomažući programerskim timovima da najbolje iskoriste CLR. On se konkretno fokusira na korišćenje apstrakcija koje CLR obezbeđuje za optimizovanje izvršenja programa.

Kit George je program menadžer u .NET Framework timu u Microsoft-u. Diplomirao je 1995. godine psihologiju, filozofiju i matematiku na Victoria univerzitetu u Velingtonu (Novi Zeland). Pre nego što se pridružio Microsoft-u, radio je kao predavač, primarno u Visual Basic-u. Učestvovao je u dizajnu i implementaciji prva dva izdanja .NET Framework-a u poslednje dve godine.

Jan Gray je softverski arhitekta u Microsoft-u koji sada radi na modelima i infrastrukturi konkurentnog programiranja. Prethodno je bio arhitekta CLR performansa, a 1990. godine je pomogao u pisanju ranih MS C++ kompajlera (na primer, semantika, model izvršnog objekta, prekompajlirana zaglavlja, PDB-ovi, inkrementalna kompilacija i povezivanje) i Microsoft Transaction Server-a. Jan-ova interesovanja obuhvataju izgradnju prilagođenih multiprocesora u FPGA-ovima.

Brian Grunkemeyer je inženjer softverskog dizajna u Microsoft .NET Framework timu od 1998. godine. Implementirao je veliki deo Framework Class Libraries-a i sarađivao na detaljima klasa u ECMA/ISO CLI standardu. Brian trenutno radi na budućim verzijama .NET Framework-a, uključujući oblasti kao što su generički tipovi, upravljana pouzdanost koda, verzionisanje, ugovori u kodu i poboljšanje programerskog iskustva. Ima dvostruku diplomu za računarske nauke i kognitivne nauke univerziteta Carnegie Mellon.

Eric Gunnerson se priključio Microsoft-u 1994. godine, nakon rada u vazduhoplovstvu i industrijama koja se gase. Radio je u timu C++ kompajlera, kao član tima za projektovanje C# jezika i kao sledbenik ranih razmišljanja o naporima DevDiv zajednice. Radio je na Windows DVD Maker UI-u tokom Vista verzije i pridružio se Microsoft Health-Vault timu početkom 2007. godine. Eric u slobodno vreme vozi bicikl, skija, lomi rebra, gradi terase, bloguje i piše o sebi u trećem licu.

Phil Haack je osnivač Haacked LLC-a, gde podučava softverske organizacije da postanu svoja najbolja verzija. Da bi to uradio, Phil primenjuje svoje 20-godišnje iskustvo u softverskoj industriji. Nedavno je bio direktor inženjeringa u GitHub-u i pomogao je da se GitHub prilagodi programerima na Microsoft platformi. Pre njegovog rada u GitHub-u, bio je senior program menadžer u Microsoft-u, odgovoran za isporuku ASP.NET MVC-a i NuGet-a, između ostalih projekata. Ovi proizvodi su imali licence otvorenog koda i uveli su Microsoft u eru otvorenog koda. Phil je koautor knjige GitHub For Dummies, kao i serije popularnih Professional ASP.NET MVC i redovni je

govornik na konferencijama širom sveta. Takođe je učestvovao u nekoliko tehnoloških podkasta, kao što su .NET Rocks, Hanselminutes, Herding Code i The Official jQuery Podcast. Možete čitati njegova razmišljanja na https://haacked.com ili na Twitter-u, https://twitter.com/haacked.

Anders Hejlsberg je tehnički kolega u Developer Division timu u Microsoft-u. On je glavni dizajner C# programskog jezika i ključni učesnik u razvoju .NET Framework-a. Pre nego što se pridružio Microsoft-u 1996. godine, Anders je bio glavni inženjer u Borland International-u. Kao jedan od prvih zaposlenih u Borland-u, on je bio originalni autor Turbo Pascal-a, a kasnije je radio kao glavni arhitekta Delphi linije proizvoda. Anders je studirao inženjering na Tehničkom univerzitetu u Danskoj.

Jan Kotas je radio na .NET Runtime-u u Microsoft-u od 2001. godine. On ima odličan osećaj za postizanje ravnoteže između produktivnosti, performansa, bezbednosti i pouzdanosti za .NET platformu. Tokom godina, imao je dodira skoro sa svim oblastima .NET izvršenja, uključujući i portove za nove arhitekture, savremene kompajlere i mnoge optimizacije. Diplomirao je 1998. godine master diplomom iz oblasti računarskih nauka na Charles Univerzitetu u Pragu (Češka).

Immo Landwerth je program menadžer .NET Framework tima u Microsoft-u. Specijalizovao se za API dizajn, Base Class Libraries (BCL) i otvoreni kod u .NET-u. Tvituje u GIF-ovima.

Rico Mariani kodira profesionalno od 1980. godine, sa iskustvom u svemu, od kontrolera u realnom vremenu do flagship razvojnih sistema. Rico je radio u Microsoft-u od 1988. do 2017. godine na jezičkim proizvodima, online svojstvima, operativnim sistemima, veb pretraživačima i tako dalje. Godine 2017. Rico se priključio Facebook-u i radi na Facebook Messenger-u, sa istom strašću, visokim kvalitetom i performansom. Njegovo interesovanje privlače kompajleri i teorija jezika, baze podataka, 3D umetnost i dobra fikcija.

Anthony Moore je glavni za razvoj u Connected System Division-u. Takođe je bio glavni za razvoj Base Class Libraries-a za CLR od 2001. do 2007. godine, tokom proširenja FX V1.0 u FX 3.5. Anthony je počeo da radi za Microsoft 1999. godine i prvo je radio na Visual Basic-u i ASP.NET-u. Pre toga, radio je kao korporativni programer osam godina u svojoj rodnoj Australiji, uključujući i period od tri godine kada je radio u industriji brze hrane.

Vance Morrison je arhitekta performansa za .NET Runtime u Microsoft-u. Uključen je u većinu aspekata performansa izvršenja, a trenutno je posvećen poboljšanju vremena pokretanja. Uključen je u dizajn komponenti .NET izvršenja od samog početka. Prethodno je radio na dizajnu .NET Intermediate Language-a (IL) i bio je vodeći programer za JIT kompajler za izvršenje.

Christophe Nasarre je softverski inženjer u Performance timu u Criteo-u. U slobodno vreme, Christophe piše postove vezane za .NET na https://medium.com/@chnasarre i obezbeđuje alate i primere koda na https://github.com/chrisnas. Takođe je Microsoft MVP u Developer Technologies-u.

Dare Obasanjo je program menadžer u MSN Communication Services Platform timu u Microsoft-u. On voli da rešava probleme XML-a za izgradnju infrastrukture servera koji koriste MSN Messenger, MSN Hotmail i MSN Spaces timovi. Prethodno je bio program menadžer u XML timu, odgovoran za osnovne interfejse programiranja XML aplikacija i tehnologije vezane za W3C XML Schema u .NET Framework-u.

Brian Pepin je softverski programer u Microsoftu i trenutno radi na softveru Xbox sistema. On se bavio razvojnim alatkama i radnim okvirima 16 godina i doprineo je dizajnu za Visual Basic 5, Visual J++, .NET Framework, WPF, Silverlight, a učestvovao je i u više nesrećnih eksperimenata koji, srećom, nikada nisu stigli do tržišta.

Jonatan Pincus je bio stariji istraživač u Systems and Networking Group-u Microsoft Research tima, gde se fokusirao na bezbednost, privatnost i pouzdanost softvera i sistema zasnovanih na softveru. Prethodno je bio osnivač i CTO Intrinsa i radio je u oblasti automatizacije dizajna (postavljanje i rutiranje za IC-ove i CAD radne okvire) u GE Calma i EDA Systems.

Jeff Prosise je suosnivač Wintellect-a i zarađuje pisanjem softvera i pomaganjem drugima da rade to isto. Napisao je devet knjiga i stotine članaka za časopise, obučio je hiljade programera u Microsoft-u i govorio na nekim od najvećih svetskih konferencija o softveru. Njegova strast je podučavanje softverskih programera da kreiraju aplikacije zasnovane na *cloud*-u upotrebom Microsoft Azure-a i predstavljanje čuda AI-ja i mašinskog učenja. U slobodno vreme, Jeff pravi velike avione koje pokreće daljinskim upravljačem i putuje na programerske radionice, univerzitete i istraživačke institucije širom sveta, podučavajući ih o Azure-u i AI-ju.

Brent Rector je program menadžer u Microsoft-u, za inkubaciju tehničke strategije. On ima više od 30 godina iskustva u industriji razvoja softvera u proizvodnji kompajlera programskih jezika, operativnih sistema, ISV aplikacija i drugih proizvoda. Brent je autor i koautor brojnih knjiga o razvoju softvera za Windows, uključujući ATL Internals, Win32 Programming (obe u izdanju Addison-Wesleya) i Introducing WinFX (Microsoft Press). Pre nego što se priključio Microsoftu, Brent je bio predsednik i osnivač Wise Owl Consulting, Inc., i glavni arhitekta njihove maske za .NET, Demeanor for .NET.

Jeffrey Richter je softverski arhitekta za Microsoft Azure, koji je poznat po tome što je autor najprodavanijih Windows via C/C i CLR via C# knjiga. Nedavno je kreirao besplatnu seriju video snimaka Architecting Distributed Cloud Applications, koja je dostupna na adresi http://aka.ms/RichterCloudApps i druge video snimke koji su dostupni na http://WintellectNOW.com. Jeffrey je bio konsultant i suosnivač Wintellect-a, kompanije za softverski konsalting i obuku.

Greg Schechter je veteran Big Tech industrije i radio je za Sun Microsystems od 1988. do 1994. godine, a za Microsoft-u od 1994. do 2010. godine, primarno na API implementaciji i API dizajnu, više od 20 godina. Njegovo iskustvo leži uglavnom u oblasti 2D i 3D grafike, ali takođe i u medijumima, slikama, sistemima osnovnog interfejsa i asinhronom programiranju. Godine 2011. Greg se priključio Facebook-u kao jedan od prve desetine zaposlednih iz Sijetla i trenutno je u Facebook-u London, gde radi kao softverski inženjer i u Ads Infrastructure organizaciji. Pored svega toga, Greg takođe voli da piše o sebi u trećem licu.

Chris Sells, u prošlom životu bio je uključen u .NET od beta verzije i pisao je i govorio mnogo o ovoj temi. Trenutno je Google Product Manager u Flutter development experience-u. I dalje uživa u dugim šetnjama plažom i u različitim tehnologijama.

Steve Starck je vodeći tehničar u ADO.NET timu u Microsoftu, gde je razvijao i dizajnirao tehnologije pristupa podacima, uključujući ODBC, OLE DB i ADO.NET, poslednjih deset godina.

Herb Sutter je vodeći stručnjak za razvoj softvera. Tokom karijere, Herb je bio kreator i glavni dizajner nekoliko glavnih komercijalnih tehnologija, uključujući PeerDirect sistem za peer replikaciju za heterogene distribuirane baze

podataka, ekstenzije jezika za C++/CLI za C++ za .NET programiranje, kao i najnoviji projekat Concur modela konkurentnog programiranja. Trenutno je softverski arhitekta u Microsoft-u i predsedavajući odbora ISO C++ standarda i autor četiri hvaljene knjige i stotine tehničkih novina i članaka na temu razvoja softvera.

Clemens Szyperski se pridružio Microsoft Research timu kao softverski arhitekta 1999. godine. Fokusira se na korišćenje softverskih komponenti za efikasnu izgradnju novih vrsta softvera. Clemens je suosnivač Oberon Microsystems-a i njihovog projekta Esmertec i vanredni je profesor u School of Computer Science, Queensland tehnološkog univerziteta u Australiji. On je autor knjige Component Sofware (Addison-Wesley) koja je osvojila Jolt nagradu i koautor knjige Software Ecosystem (MIT Press). Ima doktorat u oblasti računarskih nauka sa Swiss Federal Institute of Technology u Cirihu i master u elektro inženjeringu/računarskom inženjeringu tehnološkog univerziteta Aachen.

Stephen Toub je partnerski softverski inženjer u Microsoft-u. Stekao je diplomu u oblasti računarskih nauka na Harvard univerzitetu u Njujorku i proveo je mnogo godina razvijajući .NET, sa fokusom na biblioteke, a posebno na performans, paralelizam i asinhronost. Bio je glavni učesnik u prenosu .NET-a na otvoreni kod i više platformi i oduševljen je svim mogućnostima koje će .NET imati u budućnosti.

Mircea Trofin je softverski inženjer u Google-u, radi na problemima optimizacije kompajlera. Prethodno je radio u Microsoft-u kao deo .NET tima.

Paul Vick je bio jezički arhitekta za Visual Basic tokom prelaska na .NET i vodio je tim za projektovanje jezika za prvih nekoliko izdanja jezika. Paul je počeo svoju karijeru u Microsoft-u 1992. godine, u Microsoft Access timu, isporučujući verzije Access-a od 1.0 do 97. Godine 1998. se prebacio u Visual Basic tim i učestvovao u projektovanju implementacije Visual Basic kompajlera i vodio redizajn jezika za .NET Framework. Autor je Visual Basic .NET Language Specification-a i knjige The Visual Basic .NET Language, izdavačke kuće Addison-Wesley. Njegov veb blog možete pronaći na adresi www.panopticoncentral.net.

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

1
Uvod

Ako biste mogli da stanete uz rame svakom programeru koji koristi vaš radni okvir za pisanje koda i objasnite mu kako bi trebalo da ga koristi, knjiga ne bi bila potrebna. Saveti predstavljeni u ovoj knjizi daju vam, kao autoru radnog okvira, paletu alatki koje vam omogućavaju da kreirate zajednički jezik između autora radnih okvira i programera koji će upotrebljavati te radne okvire. Na primer, otkrivanje operacije kao svojstva, umesto kao metoda, prenosi vitalne informacije o načinu na koji će operacija biti upotrebljena.

U ranim danima PC ere glavne alatke za razvoj aplikacija su bili kompajleri programskog jezika, veoma mali skup standardnih biblioteka i neobrađeni interfejsi za programiranje aplikacija (API) operativnog sistema – osnovni skup programerskih alatki niskog nivoa.

Iako su programeri kreirali aplikacije upotrebom tih osnovnih alata, oni su otkrivali sve veću količinu koda koji se ponavljao i mogao je da bude uklonjen pomoću API-a višeg nivoa. Proizvođači operativnih sistema su primetili da oni mogu da ponude programerima jeftinije kreiranje aplikacija za njihove sisteme ako im obezbede takve API-e visokog nivoa. Broj aplikacija koje mogu da se pokreću u sistemu će se povećati, što sistem čini mnogo pogodnijim za krajnje korisnike koji su tražili različite aplikacije. Takođe, nezavisni proizvođači alatki i komponenti brzo su prepoznali poslovnu mogućnost koja je obezbeđena povećanjem nivoa API apstrakcije.

Paralelno, industrija je polako počela da prihvata objektno-orijentisano projektovanje i njegovo naglašavanje proširivosti i višekratnosti[1]. Kada su proizvođači višekratne biblioteke usvojili objektno-orijentisano programiranje (OOP) za razvoj njihovih API-a visokog nivoa, rođen je i koncept radnog okvira. Odnosno, programeri aplikacija više nisu morali da pišu većinu aplikacije od nule. Radni okvir obezbeđuje većinu potrebnih delova koji su, zatim, prilagođeni i povezani[2] tako da formiraju aplikacije.

Pošto je sve više proizvođača počelo da obezbeđuje komponente koje mogu ponovo da se upotrebe spajanjem u jednu aplikaciju, programeri su primetili da se neke od komponenti ne uklapaju međusobno. Njihove aplikacije su izgledale i funkcionisale kao kuća koju su gradili različiti izvođači radova, koji nikada nisu razgovarali međusobno. Slično, pošto je veći procenat izvornog koda aplikacije bio posvećen API pozivima, umesto standardnim jezičkim strukturama, programeri su počeli da se žale da je sada potrebno da čitaju i pišu više jezika: jedan programski jezik i nekoliko „jezika" komponenti koje su želeli ponovo da upotrebe. To je imalo značajan negativan uticaj na produktivnost programera – a produktivnost je jedan od glavnih faktora uspeha bilo kog radnog okvira. Postalo je jasno da postoji potreba za zajedničkim pravilima koja će obezbediti konzistentnost i bešavnu integraciju višekratnih komponenti.

Većina današnjih platformi za razvoj aplikacija ispisuje neku vrstu konvencija projektovanja, koje se koriste tokom projektovanja radnih okvira za platformu. Radni okviri koji ne prate te konvencije, pa se stoga ne integrišu dobro sa platformom, ili su izvor konstantne frustracije onih koji pokušavaju da ih upotrebe ili, zbog manjka konkurentnosti, na kraju propadaju na tržištu. Oni uspešni, često su opisani kao samokonzistentni, smisleni i dobro dizajnirani.

---

1 Objektno-orijentisani jezici nisu jedini jezici pogodni za razvoj proširivih i višekratnih biblioteka, već oni imaju ključnu ulogu u popularizaciji koncepta višekratnosi i proširivosti. Proširivost i višekratnost su veliki deo filozofije objektno-orijentisanog programiranja (OOP), a usvajanje OOP-a doprinelo je povećanoj svesti o njihovim prednostima.

2 U poslednje vreme postoji dosta kritike u vezi sa objektno-orijentisanim (OO) dizajnom, u kojima se navodi da obećanje višekratnosti nikada nije materijalizovano. Međutim, OO dizajn obezbeđuje prirodne strukture za izražavanje višekratnih jedinica (tipova), za komunikaciju i kontrolu tačaka proširivosti (virtuelni članovi) i za olakšavanje razdvajanja (apstrakcije).

1.1 Kvalitet dobro dizajniranog radnog okvira

Pitanje je, šta definiše dobro dizajniran radni okvir i kako da to ostvarite? Na kvalitet softvera utiče mnogo faktora, kao što su performans, pouzdanost, bezbednost, upravljanje zavisnostima i tako dalje. Radni okviri, naravno, moraju da se pridržavaju ovih standarda kvaliteta. Razlika između radnih okvira i drugih vrsta softvera je to što je radni okvir sastavljen od ponovo upotrebljivih API-a, faktor koji predstavlja skup specijalnih razmatranja u projektovanju kvalitetnih radnih okvira.

1.1.1 Dobro dizajnirani radni okviri su jednostavni

Većini radnih okvira ne nedostaje moć, jer je dodavanje funkcija jednostavno kada zahtevi postanu jasniji. Nasuprot tome, jednostavnost je često žrtvovana zbog pritiska u rasporedu, sporih funkcija ili kada želja za zadovoljenjem svakog mogućeg scenarija preuzme razvojni proces. Međutim, jednostavnost je obavezna karakteristika svakog radnog okvira. Ako sumnjate u kompleksnost dizajna, skoro je uvek bolje da uklonite funkciju iz aktuelnog izdanja i provedete više vremena u poboljšanju dizajna za sledeće izdanje. Kao što dizajneri radnog okvira često kažu, „Uvek možete da dodate; nikada ne možete da uklonite.” Ako vam se čini da dizajn nije dobar, a svakako ga isporučite, verovatno ćete se pokajati zbog toga.

CHRIS SELLS - Kao test da li je API „jednostavan", ja volim da ga podvrgnem testu koji nazivam „client-first programiranje". Ako kažete šta izvršava vaša biblioteka i zamolite programera da napiše program u odnosu na to kako očekuje da takva biblioteka izgleda (bez pregleda vaše biblioteke), da li će programer osmisliti nešto suštinski slično onome što ste vi kreirali? Isprobajte to sa nekoliko programera. Ako većina programera napiše sličan program bez podudaranja sa onim što ste vi kreirali, oni su u pravu, vi niste, a vaša biblioteka bi trebalo da bude ažurirana u skladu sa tim.

Smatram ovu tehniku toliko korisnom da često projektujem API-e biblioteke pisanjem klijentskog koda koji bih želeo da pišem, a zatim samo implementiram biblioteku u skladu sa tim. Naravno, trebalo bi da uravnotežite jednostavnost sa suštinskom složenošću funkcionalnosti koju pokušavate da obezbedite, ali za to služi vaša diploma iz oblasti računarskih nauka!

Mnogi saveti opisani u ovoj knjizi su motivisani željom uspostavljanja odgovarajuće ravnoteže između mogućnosti i jednostavnosti. Konkretno u Poglavlju 2, opisane su detaljno neke osnovne tehnike za kreiranje jednostavnosti i mogućnosti odgovarajućeg nivoa, koje koriste mnogi najuspešniji dizajneri radnih okvira.

1.1.2 Dobro dizajnirani radni okviri su skupi za projektovanje

Dobar dizajn radnog okvira se ne dobija magijom. To je težak posao koji zahteva puno vremena i resursa. Ako ne želite da investirate stvarni novac u dizajn, ne bi trebalo da očekujete da kreirate dobro dizajniran radni okvir.

- STEPHEN TOUB - Za nas koji smo u situaciji da učestvujemo na više sastanaka nego što želimo, izračunavanje troškova sastanka može biti interesantan hobi: broj ljudi u prostoriji, pomnožen sa procenjenom prosečnom zaradom po satu učesnika, pomnoženo sa učestalošću sastanaka. Prema tome, jedan od najskupljih sastanaka na kom sam učestvovao je sastanak povodom pregleda .NET API-a, na kom smo pregledali planirane nove oblasti API-a i odlučivali da li da, i kako, nastavimo rad. Međutim, taj trošak se isplati, jer nedoslednosti i greške u API dizajnu, na kraju, imaju veće negativne troškove, a dobro dizajniran, integrisan API, na kraju, ima vrednost koja je „veća od sume njegovih delova".

Projektovanje radnih okvira bi trebalo da bude eksplicitan i poseban deo procesa razvoja;[3] mora da bude eksplicitan zato što je potrebno da bude pravilno planiran i izvršen, a mora da bude poseban zato što ne može da bude samo sporedan efekat procesa implementacije. Često se dešava da se radni okvir sastoji od tipova i članova koji su ostali javni nakon završetka procesa implementacije.

Najbolje projektovane radne okvire kreiraju ljudi čija je eksplicitna odgovornost projektovanje radnih okvira, ili ljudi koji preuzimaju ulogu projektanta radnog okvira u odgovarajuće vreme u razvojnom procesu. Mešanje odgovornosti je greška i dovodi do toga da će dizajn otkriti detalje o implementaciji koji ne bi trebalo da budu vidljivi krajnjem korisniku radnog okvira.[4]

> **- JEREMY BARTON -** Iznenađujuće je teško kreirati dobar API kao stručnjak za tu oblast. Znate koliko je zapravo problematična oblast i vaša preporuka je svela problem na njegovu suštinu – to je vrhunac jednostavnosti. Ali ne. Neko ko nije upoznat sa detaljima će vam verovatno i dalje govoriti da je previše komplikovano i verovatno je u pravu.

Čak i API predlozi članova tima za pregled .NET API-a se menjaju tokom procesa pregleda. Proces pregleda API-a, koji uključuje nekog ko nije stručnjak u tom domenu, značajno poboljšava kvalitet bilo kog API predloga.

3 Nemojte ovo shvatiti pogrešno, kao odobrenje za teške procese projektovanja unapred. U stvari, teški procesi projektovanja API-a dovode do nepotrebnih troškova, jer je API često potrebno dodatno podešavati nakon implementacije. Međutim, proces projektovanja mora biti odvojen od procesa implementacije i trebalo bi da bude inkorporiran u svaki deo ciklusa proizvodnje: faza planiranja (koji su API-i potrebni našim korisnicima?), proces projektovanja (koje kompromise funkcionalnosti smo spremni da primenimo da bismo dobili odgovarajuće API-e radnog okvira?), proces razvoja (da li smo odvojili vreme da isprobamo upotrebu radnog okvira, da bismo videli kako izgleda krajnji rezultat?), beta proces (da li smo odvojili vreme za skupo redizajniranje API-a?) i održavanje (da li smanjujemo kvalitet dok razvijamo radni okvir?).

4 Izrada prototipa je jedan od najvažnijih delova procesa projektovanja radnog okvira, ali je izrada prototipa potpuno drugačija od implementacije.

1.1.3 Dobro dizajnirani radni okviri su puni kompromisa

Ne postoji savršen dizajn. Dizajn podrazumeva pravljenje kompromisa, a da biste doneli odgovarajuće odluke trebalo bi da razumete opcije, njihove prednosti i nedostatke. Ako se nađete u situaciji da uživate u tome što imate dizajn bez kompromisa i što ste stvorili nešto značajno, verovatnije je da vam nešto značajno nedostaje.

Praksa koja je opisana u ovoj knjizi predstavljena je u formi saveta, a ne pravila, upravo zato što projektovanje radnog okvira zahteva upravljanje kompromisima. Neki saveti opisuju kompromise koji su uključeni i čak obezbeđuju alternative koje bi trebalo razmotriti u specifičnim situacijama.

1.1.4 Dobro dizajnirane radne okvire pozajmite iz prošlosti

Najuspešniji radni okviri pozajmljuju od postojećih dokazanih projekata i grade se povrh njih. Moguće je – i često poželjno – predstaviti potpuno nova rešenja dizajna radnog okvira, ali bi to trebalo da se uradi veoma pažljivo. Pošto se broj novih koncepata uvećava, verovatnoća da će uopšteni dizajn biti dobar se smanjuje.

> **- CHRIS SELLS -** Molim vas ne uvodite inovacije u projektovanje biblioteka. Kreirajte API za svoju biblioteku tako da bude što je moguće dosadniji. Želite da funkcionalnost bude interesantna, a ne API.

> **- JEREMY BARTON -** Potpuno nova rešenja je najbolje rezervisati za unakrsne oblasti. Ona vam pomažu da razumete stvarnu vrednost novog rešenja, a vašim korisnicima da razumeju širi uticaj i zašto bi trebalo da nauče „još nešto".
>
> Promena je strašna, osim kad je sjajna. Biti samo malo bolji u maloj oblasti verovatno nije vredno vremena koje je korisnicima potrebno da nauče kako da koriste vaš novi pristup.

Saveti koji se nalaze u ovoj knjizi su zasnovani na iskustvima koja smo stekli dok smo dizajnirali osnovne biblioteke za .NET; oni podstiču na pozajmljivanje iz elemenata koji su funkcionisali i izdržali test vremena, a upozoravaju na one koji nisu. Podstičemo vas da upotrebite ovu dobru praksu kao početnu tačku i

da je nakon toga poboljšavate. U Poglavlju 9 govorićemo detaljno o uopštenim pristupima dizajnu koji funkcionišu.

1.1.5 Dobro dizajnirani radni okviri su dizajnirani da evoluiraju

Razmišljanje o tome kako da poboljšate radni okvir u budućnosti će zahtevati razmišljanje o tome koje kompromise je potrebno da napravite. Sa jedne strane, dizajner radnog okvira može sigurno da odvoji više vremena i da uloži mnogo više napora u proces dizajniranja, a povremeno se može uvesti dodatna kompleksnost, „za svaki slučaj". Sa druge strane, pažljivo razmatranje vam može pomoći u tome da ne isporučite nešto što će vremenom biti degradirano ili, još gore, nešto što neće moći da zadrži kompatibilnost sa starijim verzijama.[5] Kao osnovno pravilo, bolje je odložiti funkciju do sledećeg izdanja, nego je uraditi polovično u aktuelnom izdanju.

Kada god pravite kompromise u dizajnu, trebalo bi da odredite kako će odluka uticati na vašu mogućnost poboljšanja radnog okvira u budućnosti. Saveti koji su predstavljeni u ovoj knjizi uzimaju u obzir ovo važno pitanje.

1.1.6 Dobro dizajnirani radni okviri su integrisani

Moderni radni okviri bi trebalo da budu dizajnirani tako da se dobro integrišu sa velikim ekosistemom različitih programerskih alatki, programskih jezika, modela aplikacija i tako dalje. Cloud računarstvo i druga serverski-orijentisana radna opterećenja podrazumevaju da je era radnih okvira dizajniranih za specifične modele aplikacija završena. To takođe važi za radne okvire koji su dizajnirani bez razmišljanja o podršci za određenu alatku, ili o integraciji sa programskim jezicima koji se koriste u programerskoj zajednici.

1.1.7 Dobro dizajnirani radni okviri su konzistentni

Konzistentnost je ključna karakteristika dobro dizajniranog radnog okvira. To je najvažniji faktor koji utiče na produktivnost. Konzistentan radni okvir omogućava prenos znanja između delova radnog okvira koje programer poznaje na delove koje programer pokušava da nauči. Konzistentnost takođe pomaže programerima da brzo prepoznaju koji delovi dizajna su zaista jedinstveni

5 Kompatibilnost sa starijim verzijama nije detaljno objašnjena u ovoj knjizi, ali bi je trebalo razmatrati kao osnovu projektovanja radnih okvira, zajedno sa pouzdanošću, bezbednošću i performansom.

za određenu oblast funkcije i stoga zahtevaju posebnu pažnju, a koji su samo dobri stari uobičajeni projektni obrasci i idiomi.

Konzistentnost je verovatno glavna tema ove knjige. Skoro svaki savet je delimično motivisan konzistentnošću, ali Poglavlja 3 i 5 su verovatno najvažnija, zato što daju osnovne savete za konzistentnost. Ove savete smo obezbedili da bismo vam pomogli da vaš radni okvir bude uspešan. U sledećem poglavlju predstavićemo savete za dizajn osnovne biblioteke.

2.
Osnove projektovanja radnog okvira

Uspešan radni okvir za opštu namenu mora da bude dizajniran za širok raspon programera sa različitim zahtevima, veštinama i pozadinama. Jedan od najvećih izazova sa kojim se dizajneri suočavaju je da obezbede moć i jednostavnost raznovrsnoj grupi korisnika.

Još jedan važan cilj dizajnera radnog okvira je da mora da obezbedi jedinstveni model programiranja bez obzira na vrstu aplikacije[1] koju programer piše, ili u slučaju višejezičnog izvršenja, programski jezik koji programer koristi.

Koristeći opšte prihvaćene osnovne principe softverskog dizajna i prateći savete koji su opisani u ovom poglavlju, možete da kreirate radni okvir koji obezbeđuje konzistentnu funkcionalnost koja odgovara širokom rasponu programera koji grade različite vrste aplikacija upotrebom različitih programskih jezika.

✓ **URADITE** - dizajnirajte radne okvire koji su moćni i jednostavni za upotrebu.

1 Na primer, komponenta radnog okvira bi trebalo da ima isti model programiranja, bez obzira na to da li se koristi u konzoli, Windows Forms-u ili ASP.NET aplikaciji, ako je to moguće.

Dobro dizajnirani radni okvir olakšava implementiranje jednostavnih scenarija. Istovremeno, nije zabranjena implementacija naprednijih scenarija, mada to može biti težak zadatak. Kao što je Alan Kay rekao, „Jednostavne stvari bi trebalo da budu jednostavne, a kompleksne stvari bi trebalo da budu moguće".

Ovaj savet se, takođe, odnosi na Pareto princip (pravilo 80/20), koje govori da u svakoj situaciji, 20 procenata će biti važno, a 80 procenata će biti trivijalno. Kada dizajnirate radni okvir, koncentrišite se na važnih 20 procenata scenarija i API-e. Drugim rečima, investirajte u dizajn najčešće upotrebljavanih delova radnog okvira.

✓ **URADITE** – razumite i eksplicitno dizajnirajte za širok raspon programera sa različitim stilovima programiranja, zahtevima i nivoima veštine.

PAUL VICK - Ne postoji magično rešenje za dizajniranje radnih okvira za Visual Basic programere. Naši korisnici se kreću od ljudi koji prvi put koriste programerske alatke do industrijskih veterana koji grade velike komercijalne aplikacije. Ključ za dizajniranje radnog okvira koji odgovara Visual Basic programerima je da se fokusirate na to da im omogućite da urade posao što je moguće jednostavnije. Dizajniranje radnog okvira koji koristi minimalni broj koncepata je dobra ideja, ne zato što VB programeri ne mogu da obrade te koncepte, već zato što potreba da se zaustavite i razmislite o konceptima koji su irelevantni za dati zadatak prekida tok rada. Cilj VB programera obično nije da nauče neke interesantne ili uzbudljive koncepte niti da budu impresionirani intelektualnom čistotom i jednostavnošću vašeg dizajna, već da urade posao i nastave dalje.

KRZYSZTOF CWALINA - Veoma je lako dizajnirati za korisnike koji su kao vi, a veoma je teško dizajnirati za nekoga ko nije kao vi. Postoji previše API-a koje su dizajnirali stručnjaci za domene i, iskreno, oni su dobri samo za stručnjake domena. Problem je što većina programera to nije, nikada neće biti i nema potrebe da budu stručnjaci u svim tehnologijama koje se koriste u modernim aplikacijama.

BRAD ABRAMS -Iako je poznati moto Hewlett-Packard-a „Gradi za sledećeg inženjera" koristan za postizanje kvaliteta i kompletnosti u softverskim projektima, pogrešan je za API dizajn. Na primer, programeri u Microsoft Word timu potpuno razumeju da oni nisu ciljni kupci za Word. Moja majka je više ciljni kupac. Prema tome, Word tim dodaje mnoge funkcije koje će mojoj majci biti korisne, umesto funkcija koje su korisne programerskom timu. Iako je to očigledno u slučaju aplikacija kao što je Word, često smo skloni da propustimo ovaj princip tokom dizajniranja API-a. Skloni smo dizajniranju API-a samo za nas, umesto da jasno razmišljamo o scenarijima kupaca.

✓ **URADITE** – razumite i dizajnirajte za više različitih programskih jezika.

Dostupne su mnoge implementacije programskih jezika koje podržavaju Common Language Runtime (CLR) i .NET. Neki od ovih jezika mogu da se razlikuju od jezika koji koristite za implementaciju API-a. Često je potrebna posebna pažnja da biste obezbedili da API-i mogu da funkcionišu dobro u različitim jezicima.

Na primer, programeri koji koriste dinamički tipizirane jezike sa mogućnošću interakcije sa .NET-om (kao što su PowerShell, IronPython i drugi) mogu imati problema tokom upotrebe API-a koji zahteva da kreiraju prilagođeni tip pomoću atributa.

Kao još jedan primer je jezik F# koji ne poštuje korisnički-definisane operatore implicitne konverzije. Kao posledica toga, API-i dizajnirani sa implicitnom konverzijom za olakšavanje pozivanja obrazaca neće biti jednostavni za upotrebu u jeziku F#.

JAN KOTAS - Dizajniranje najmanjeg zajedničkog imenioca postojećih programskih jezika počelo je da koči razvoj .NET platforme. To je poslednjih godina deakcentovano da bi se omogućile inovacije kao što je Span<T>. C# i F# predstavili su nove funkcije jezika za omogućavanje funkcije Span<T>. Međutim, drugi tradicionalni jezici za .NET (kao što je Visual Basic) nisu omogućili ovu funkciju, pa se za njih ne mogu upotrebiti novi API-i zasnovani na Span-u.

JEREMY BARTON - Iako je istina da smo dodali Span<T> bez podrške jezika iz VB-a, savet za upotrebu Span tipova – koje ćete pronaći u odeljku 9.12 – preporučuju upotrebu alternativnih metoda zasnovanih na nizu. Ta preporuka je delimično zasnovana na upotrebljivosti, ali se na kraju vraća na ovaj savet i na raznolikost jezika koji mogu da komuniciraju sa .NET CLR-om.

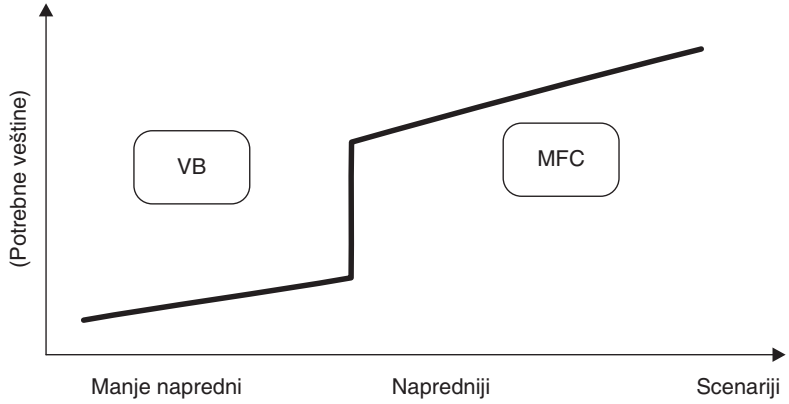
Druga specijalna razmatranja za različite programske jezike su opisana u ovoj knjizi.

2.1 Progresivni radni okviri

Dizajniranje jednog radnog okvira za širi raspon programera, scenarija i jezika je težak i skup posao. Istorijski gledano, proizvođači radnih okvira su obezbeđivali nekoliko proizvoda namenjenih specifičnim grupama programera, za specifične scnarije. Na primer, Microsoft je obezbeđivao Visual Basic API-e optimizovane za jednostavnost i relativno mali skup scenarija, kao i Win32 API-e optimizovane za moć i fleksibilnost, čak i ako je to značilo žrtvovanje lakoće upotrebe. Drugi radni okviri, kao što su MFC i ATL, su takođe bili namenjeni specifičnim grupama programera i scenarija.

Iako je ovaj pristup sa više radnih okvira dokazan kao uspešan način obezbeđivanja API-a koji su moćni i jednostavni za upotrebu za specifične grupe programera, ima i značajne nedostatke. Glavni nedostatak[2] je da mnoštvo radnih okvira otežava programerima da koriste jedan od radnih okvira za prenos svog znanja na sledeći nivo ili scenario (koji često zahteva drugi radni okvir). Na primer, kada bi trebalo da implementiraju drugu aplikaciju koja zahteva moćniju funkcionalnost, programeri se suočavaju sa veoma strmom krivom učenja, zato što je potrebno da nauče potpuno drugi način programiranja, kao što je prikazano na Slici 2-1.

2 Drugi nedostaci uključuju sporije stizanje na tržište radnih okvira koji su omotači povrh drugih radnih okvira, dupliranje rada i nedostatak zajedničkih alatki.



Slika 2-1: Kriva učenja platforme sa više radnih okvira

ANDERS HEJLSBERG - U ranim danima Windowsa, imali ste Windows API. Za pisanje aplikacija pokretali ste C kompajler, #included windows.h, kreirali ste winproc i obrađivali poruke u prozoru – u suštini stari Petzold stil Windows programiranja. Iako je ovo funkcionisalo, nije bilo posebno produktivno, niti posebno jednostavno.

Tokom vremena, pojavili su se različiti modeli programiranja povrh Windows API-a. VB je prihvatio Rapid Application Development (RAD). Pomoću VB-a mogli ste da instancirate obrazac, prevučete komponente u obrazac i napišete funkcije za obradu događaja; pomoću delegiranja, kod se izvršava.

U svetu C++, imali smo MFC i ATL sa drugačijim stavom. Ključni koncept je potklasiranje. Programeri su izvršavali potklasiranje iz postojećeg monolitnog, objektno-orijentisanog radnog okvira. Iako to pruža veću moć i izražajnost, ne podudara se sa lakoćom ili produktivnošću kompozitnog modela VB-a.

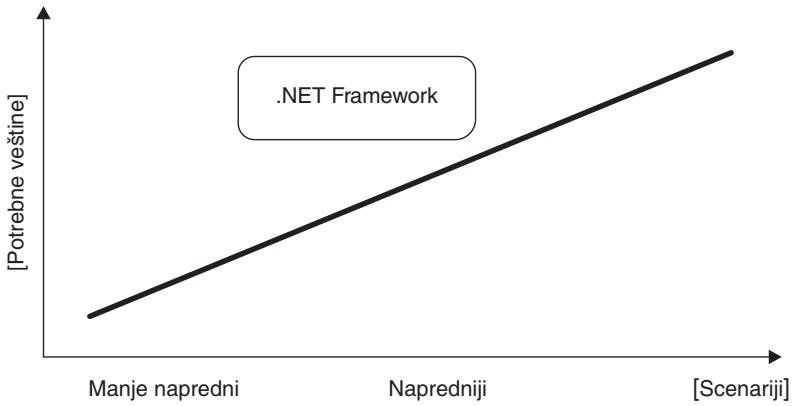
Ako pogledate ovu sliku, jedan od problema je to što vaš izbor modela programiranja takođe obavezno postaje vaš izbor programskog jezika. Ovo je loša situacija. Ako ste vešt MFC programer i trebalo bi da napišete kod u VB-u, vaše veštine se neće prevesti. Slično, ako znate mnogo o VB-u, neće se mnogo tog znanja preneti u MFC.

Takođe, ne postoji dosledna dostupnost API-a. Svaki od ovih modela osmišlja sopstvena rešenja za veliki broj problema koji su u stvari osnova svih modela – na primer, kako da koristim I/O fajl, kako da formatiram znakovni niz, kako da obezbedim sigurnost, obradu niti i tako dalje?

Ono što .NET Framework radi je da ujedinjuje sve ove modele. Daje vam konzistentan API koji je dostupan svuda, bez obzira na jezik koji koristite ili model programiranja koji ciljate.

PAUL VICK - Takođe vredi pomenuti da ovo ujedinjenje ima svoju cenu. Postoji nerešiva tenzija između pisanja radnih okvira koji otkrivaju veliku količinu moći i omogućavaju programeru veliku kontrolu nad ponašanjem i pisanje radnih okvira koji otkrivaju ograničeniju funkcionalnost na krajnje konceptualno jednostavan način. U većini slučajeva, ne postoji brzo rešenje i kompromisi se neizbežno moraju praviti između moći sa jedne strane i jednostavnosti sa druge strane. Ogromna količina posla je uložena u dizajniranje .NET Frameworka da bi se obezbedilo postizanje najbolje moguće ravnoteže između ova dva cilja, ali mislim da je to nešto na čemu ćemo nastaviti da radimo.

Mnogo bolji pristup je da obezbedite progresivni radni okvir, koji je jedan radni okvir namenjen za širok spektar programera i omogućava prenos znanja sa manje naprednih na naprednije scenarije. .NET Framework je progresivan radni okvir i obezbeđuje takvu postupnu krivu učenja (vidite Sliku 2-2).



Slika 2-2: Kriva učenja platforme progresivnog radnog okvira

Postizanje postupne krive učenja sa nižom tačkom unosa je teško, ali ne i nemoguće. Teško je zato što zahteva novi pristup za proces dizajna radnog okvira, zahteva mnogo veću disciplinu dizajna i ima više troškove dizajna. Srećom, saveti opisani u ovom poglavlju i u ovoj knjizi vodiće vas kroz težak proces dizajna, što će vam pomoći da dizajnirate odličan progresivni radni okvir.

Takođe, treba da imate na umu da je programerska zajednica ogromna. Kreće se u rasponu od kancelarijskih radnika, koji snimaju makro naredbe, do autora drajvera uređaja niskog nivoa. Svaki radni okvir koji pokušava da posluži svim ovim korisnicima na kraju će postati neuredan i neće zadovoljiti nijednog korisnika. Cilj progresivnog radnog okvira je da se proširi na širok raspon programera, ali ne na sve moguće programere. To jasno podrazumeva da oni programeri koji spadaju van te ciljne grupe zahtevaju specijalne API-e.

2.2 Osnovni principi projektovanja radnog okvira

Obezbeđivanje razvojne platforme koja je moćna i jednostavna za upotrebu je jedan od glavnih ciljeva .NET-a, a trebalo bi da bude i vaš cilj, ako je proširujete. Prva verzija .NET Frameworka je predstavila moćan skup API-a, ali su neki programeri smatrali delove radnog okvira previše teškim za upotrebu.

RICO MARIANI - Druga strana ovoga je da upotreba API-a ne samo da bi trebalo da bude jednostavna već bi takođe upotreba API-a na najbolji mogući način trebalo da bude jednostavna. Pažljivo razmislite o tome koji ćete obrazac ponuditi i uverite se da najprirodniji način za upotrebu vašeg sistema daje rezultate koji su korektni, da je zaštićen od napada i da ima dobru performansu. Otežajte izvršenje nečega na pogrešan način. Pre nekoliko godina sam napisao:

Put do uspeha

U potpunoj suprotnosti od penjanja na planinski vrh, ili puta kroz pustinju da bi se došlo do pobede, kroz mnoga iskušenja i iznenađenja, mi želimo da naši klijenti, jednostavno, nauče da pobeđuju tako što će koristiti našu platformu i naš radni okvir. Prava produktivnost dolazi onda kada možemo lako da kreiramo vrhunske proizvode – a ne kada možemo lako da kreiramo smeće. Izgradite put do uspeha.

Povratne informacije korisnika i studije upotrebljivosti pokazale su da je veliki segment VB programerske grupe imao problema u učenju VB.NET-a. Deo problema potiče iz jednostavne činjenice da je .NET drugačiji od VB 6.0 biblioteka, ali takođe postoji i nekoliko problema upotrebljivosti koji se odnose na dizajn API-a. Rešavanje ovih problema je postao prioritet za Microsoft u vremenskom okviru .NET Framework-a 2.0.

Principi opisani u ovom odeljku, identifikovani naslovom „Principi projektovanja radnog okvira", razvijeni su za rešavanje problema koje smo upravo pomenuli. Namenjeni su da pomognu dizajnerima radnih okvira da izbegavaju najteže greške u dizajnu, koje su zabeležene u mnogim studijama upotrebljivosti i u povratnim informacijama korisnika. Verujemo da su ovi principi ključni za projektovanje bilo kog radnog okvira za opštu namenu. Neki od principa i preporuka se preklapaju, što je verovatno i potvrda njihove validnosti.

2.2.1 Princip dizajna vođenog scenarijem

Radni okviri često sadrže veoma veliki skup API-a. Ovo je potrebno da bismo omogućili napredne scenarije koji zahtevaju moć i izražajnost. Međutim, većina razvoja se kreće oko malog skupa uobičajenih scenarija, koji se oslanjaju na relativno mali podskup potpunog radnog okvira. Da biste optimizovali uopštenu produktivnost programera koji koriste radni okvir, važno je da uložite mnogo u dizajn API-a koji se koriste u najčešćim scenarijima.

Zato, dizajn radnog okvira bi trebalo da se fokusira na skup uobičajenih scenarija, do tačke gde se ceo proces dizajna zasniva na scenariju. Preporučujemo da dizajneri radnog okvira prvo napišu kod koji će korisnici radnog okvira morati da napišu u glavnim scenarijima, a zatim da dizajniraju model objekta za podršku ovih primera koda.[3]

Principi projektovanja radnog okvira

Radni okviri moraju da budu projektovani počevši od skupa scenarija upotrebe i primera koji implementiraju te scenarije.

3 Ovo je slično procesima zasnovanim na razvoju vođenom testiranjem (TDD) ili na slučajevima upotrebe. Međutim, postoje neke razlike. TDD je teži, zato što ima druge ciljeve osim vođenja dizajna API-a. Slučajevi upotrebe opisuju scenarija na višem nivou od individualnih API poziva.

KRZYSZTOF CWALINA - Principu koji smo upravo ispisali želeo bih da dodam „Jednostavno, ne postoji drugi način projektovanja dobrog radnog okvira". Ako bih ja morao da biram samo jedan princip dizajna koji bi bio uključen u ovu knjigu, to bi bio upravo ovaj. Ako ne bih pisao knjigu već kraći članak o tome šta je važno u API dizajnu, izabrao bih ovaj princip

Dizajneri radnih okvira često prave greške, jer počinju dizajnom objektnog modela (koristeći različite metodologije dizajna), a zatim pišu primere koda na osnovu rezultirajućeg API-a. Problem je to što je većina metodologija dizajna (uključujući i najčešće upotrebljavane metodologije objektno-orijentisanog dizajna) optimizovana za održivost rezultirajuće implementacije, a ne za upotrebljivost rezultirajućih API-a. Oni su najpogodniji za interne arhitekture – ne za dizajn javnog API sloja velikog radnog okvira.

Kada projektujete radni okvir trebalo bi da počnete kreiranjem specifikacije API-a vođenog scenarijem (vidite Dodatak C). Ova specifikacija može da bude odvojena od funkcionalne specifikacije, ili dela većeg dokumenta specifikacija. U ovom drugom slučaju, specifikacija API-a bi trebalo da prethodi funkcionalnoj po lokaciji i vremenu.

Specifikacija bi trebalo da sadrži odeljak scenarija u kom je izlistano prvih pet do deset scenarija za datu tehnološku oblast i trebalo bi da prikaže primere koda koji implementiraju ove scenarije. Kada API ili primeri koda koriste nove, ili na drugi način neuobičajene jezičke funkcije, trebalo bi da razmotrite pisanje primera u drugom jeziku, barem zato što se kod napisan u tim jezicima ponekad značajno razlikuje.

Takođe je važno pisati ove scenarije upotrebom različitih stilova kodiranja koji su uobičajeni među korisnicima određenog jezika (upotrebom funkcija specifičnih za jezik). Primeri bi trebalo da budu napisani upotrebom veličine slova specifične za jezik. Na primer, VB.NET ne razlikuje veličinu slova, pa bi primeri trebalo to i da odražavaju. C# kod bi trebalo da prati standardnu upotrebu veličine slova, koja je opisana u Poglavlju 3.

✓ **URADITE** –uverite se da je specifikacija API dizajna ključni deo dizajna bilo koje funkcije koja uključuje javno dostupan API.

U Dodatku C pronaći ćete primer takve specifikacije.

✓ **URADITE** –definišite najbolje scenarije upotrebe za oblast glavne funkcije.

API specifikacija bi trebalo da uključuje odeljak koji opisuje glavna scenarija i prikazuje primere koda koji implementiraju ove scenarije. Ovaj odeljak bi trebalo da se prikaže odmah nakon odeljka izvršnog pregleda. Oblast prosečne funkcije (kao što je I/O fajl) trebalo bi da ima od pet do deset glavnih scenarija.

✓ **URADITE** –uverite se da scenariji odgovaraju određenom nivou apstrakcije. Trebalo bi približno da odgovaraju slučajevima upotrebe krajnjeg korisnika.

Na primer, čitanje iz fajla je dobar scenario. Otvaranje fajla, čitanje linije teksta iz fajla ili zatvaranje fajla nisu dobri scenariji; oni su previše detaljni.

✓ **URADITE** –dizajnirajte API-e tako što ćete prvo napisati primere koda za glavne scenarije, a zatim definišite objektni model za podršku primera koda.

Na primer, kada dizajnirate API za merenje proteklog vremena, možete da napišete sledeće primere koda scenarija:

```
// scenario #1 : measure time elapsed
Stopwatch watch = Stopwatch.StartNew();
DoSomething();
Console.WriteLine(watch.Elapsed);

// scenario #2 : reuse stopwatch
Dim watch As Stopwatch = Stopwatch.StartNew()
DoSomething();
Console.WriteLine(watch.ElapsedMilliseconds)

watch.Reset()
watch.Start()
DoSomething()
Console.WriteLine(watch.Elapsed)
```

Ovi primeri koda dovode do sledećeg objektnog modela:

```
public class Stopwatch {
  public static Stopwatch StartNew();

  public void Start();
  public void Reset();

  public TimeSpan Elapsed { get; }
  public long ElapsedMilliseconds { get; }
  ...
}
```

JOE DUFFY - Kao softverski programeri, mi uživamo da kreiramo nove, zabavne i moćne mogućnosti i da ih delimo sa drugim programerima. To je jedan od razloga zbog kojih je API dizajn toliko interesantan. Ali je takođe veoma teško objektivno proceniti da li je nova mogućnost, koja vas toliko zaokuplja, u stvari korisna u stvarnom svetu. Upotreba scenarija je najbolji način, koji ja znam, za identifikovanje potrebe za novim mogućnostima i njihove idealne upotrebe. Razvijanje scenarija je u stvari veoma teško, iz dobrog razloga: zahteva jedinstvenu kombinaciju tehničkih veština i razumevanja kupaca. Kada završite, možete da donesete niz odluka samo na osnovu osećaja i intuicije, pa možda i isporučite neke korisne API-e, ali je daleko veća mogućnost da ćete doneti odluke zbog kojih ćete kasnije zažaliti. Kada sumnjate, najbolje je da izostavite funkciju i odlučite da je dodate kasnije, kada budete bolje razumeli izraženu potrebu.

STEPHEN TOUB - Dodavanje novih API-a je vrlo interesantno. Ali svaki novi API ima svoju cenu. Ponekad je ta cena „samo" cena projektovanja, razvoja, testiranja, dokumentovanja i održavanja funkcionalnosti (plus povezani troškovi izvršnog okruženja). Međutim, nažalost, uobičajeno je da dodavanje API-a, zapravo, onemogućava nekome u budućnosti da doda drugu, potencijalno poželjniju ili uticajniju funkcionalnost, zato što takva funkcionalnost može imati konflikt sa funkcionalnošću koju otkriva API. To je jedan od razloga za oprez pri biranju novog API-a koji ćemo izložiti, jer možemo doći u situaciju da zapečatimo svoju sudbinu za neke buduće inovacije. Ako bi se postavilo pitanje, siguran sam da bi mnogi od nas imali omiljeni primer API-a koji želimo da nikada nismo dodali; ja znam da ih imam nekoliko.

✓ **URADITE** – napišite primere koda glavnog scenarija u najmanje dve različite familije jezika (na primer, C# i F#).

Najbolje je da se uverite da jezici koje ste izabrali imaju značajno različitu sintaksu, stil i mogućnosti.

PAUL VICK - Ako pišete radni okvir koji će upotrebljavati više jezika, korisno je da znate više od jednog programskog jezika (ne računa se poznavanje više od jednog jezika C stila). Otkrili smo da ponekad API funkcioniše dobro samo u jednom jeziku, a to je zato što osoba koja je projektovala API (i testirala API) zaista poznaje samo jedan jezik. Naučite nekoliko .NET jezika i koristite ih na način na koji su dizajnirani da se koriste. Očekivanje da ceo svet govori vašim jezikom ne funkcioniše dobro na višejezičkoj platformi kao što je .NET Framework.

✓ **RAZMOTRITE** -pisanje primera koda glavnog scenarija upotrebom dinamički tipiziranih jezika, kao što su PowerShell ili IronPython.

Veoma je lako dizajnirati API-e koji ne funkcionišu dobro sa dinamički tipiziranim jezicima. Takvi jezici često imaju problema u vezi sa nekim generičkim metodima, kao i API-e koji se oslanjaju na primenu atributa, ili kreiranje strogo tipiziranih tipova.

✗ **NEMOJTE** - se oslanjati na standardne metodologije dizajna kada dizajnirate javni API sloj radnog okvira.

Standardne metodologije dizajna (uključujući i metodologiju objektno-orijentisanog dizajna) optimizovane su za održivost rezultirajuće implementacije, a ne za upotrebljivost rezultirajućih API-a. Dizajn vođen scenarijem, zajedno sa izradom prototipa, studijama upotrebljivosti i nekim iteracijama, je mnogo bolji pristup.

CHRIS ANDERSON - Svaki programer ima svoju metodologiju i, iako ne postoji ništa suštinski pogrešno u upotrebi drugih pristupa modelovanja, generalni problem je ispis. Početi pisanje koda koji želite da programer napiše je skoro uvek najbolji pristup – zamislite to kao formu razvoja vođenog testiranjem. Napišite savršen kod, a zatim ga pregledajte unazad, da otkrjete koji objektni model želite.

2.2.1.1 Studije upotrebljivosti

Studije upotrebljivosti prototipa radnog okvira izvedene sa širokim rasponom programera je ključ za dizajn vođen scenarijem. API-i za najbolje scenarije mogu se činiti jednostavnim svojim autorima, ali možda neće biti jednostavni drugim programerima.

Razumevanje načina na koji programeri pristupaju svakom od glavnih scenarija obezbeđuje koristan uvid u dizajn radnog okvira i koliko dobro okvir ispunjava potrebe svih ciljnih programera. Zbog toga, izvođenje studija upotrebljivosti – formalno ili neformalno – je veoma važan deo procesa projektovanja radnog okvira.

Ako tokom studije upotrebljivosti otkrijete da većina programera ne može da implementira jedan od scenarija, ili ako je pristup koji koriste znatno drugačiji od onog koji je dizajner očekivao, API bi trebalo da bude redizajniran.

KRZYSZTOF CWALINA - Mi nismo testirali upotrebljivost tipova u System.IO imenskom prostoru pre isporuke verzije 1.0 .NET Framework-a. Ubrzo nakon isporuke, primili smo negativnu povratnu informaciju od korisnika o upotrebljivosti System.IO imenskog prostora. Bili smo prilično iznenađeni i odlučili smo da izvedemo studije upotrebljivosti sa osam prosečnih programera. Osmorica od njih osmoro nisu uspeli da pročitaju tekst iz fajla za 30 minuta, koje smo dodelili za izvršenje zadatka. Verujemo da je do toga došlo zbog problema sa pretraživačem dokumentacije i nedovoljnom pokrivenošću uzorka; međutim, jasno je da je sam API imao nekoliko problema upotrebljivosti. Da smo izveli studije pre isporuke proizvoda, mogli smo eliminisati značajan izvor nezadovoljstva kupca i izbeći troškove pokušaja ispravke oblasti glavne funkcije API-a bez uvođenja velikih promena.

BRAD ABRAMS - Ne postoji moćnije iskustvo koje bi API dizajneru dalo bolje razumevanje upotrebljivosti njegovog API-a, od sedenja iza jednosmernog ogledala i posmatranja programera iznerviranog zbog API-a koji je dizajnirao a koji na kraju ne uspeva da izvrši zadatak. Lično sam prošao kroz mnoga emotivna stanja dok sam gledao studije upotrebljivosti za System.IO imenski prostor, koje smo izvodili odmah nakon isporučivanja verzije 1.0. Dok programer za programerom nije uspevao da izvrši jednostavan zadatak, moje emocije su iz arogancije prešle u nevericu, zatim u frustraciju i, na kraju, u strogu odluku da rešim problem u API-u.

CHRIS SELLS - Studije upotrebljivosti mogu da budu formalne, ako imate tu vrstu vremena i novca. Ali dobićete 80 procenata povratnih informacija koje su vam potrebne samo ako predloženi API pokrene nekoliko programera bliskih ciljnoj publici vaše biblioteke. Nemojte dozvoliti da vas pojam „studija upotrebljivosti" zaplaši toliko da ne URADITE ništa: umesto toga, zamislite je kao studiju „hej, pogledaj ovo".

STEVEN CLARKE - Umesto da ulažete mnogo truda u planiranje, dizajniranje i pokretanje jedne velike studije sa mnogo učesnika i pokušate da obuhvatite što je moguće veću oblast API-a, otkrili smo da je mnogo vrednije pokrenuti niz manjih, fokusiranih studija kroz proces razvoja API-a. U svakoj studiji, tražimo od malog broja učesnika da se fokusira na jedno pitanje dizajna ili oblast API-a. Ono što naučimo iz te studije koristimo i ponavljamo u dizajnu, a zatim nedelju dana, ili dve nedelje kasnije, pokrenemo ponovo studiju na ažuriranom dizajnu ili drugoj oblasti API-a. Ovaj pristup kontinualnog učenja podrazumeva da postoji konstantan tok uvida kupaca koji nas informiše o procesu dizajna, umesto jedne velike doze koja se isporučuje u jednoj specifičnoj tački u procesu.

Studije upotrebljivosti API-a bi trebalo da budu izvršene upotrebom stvarnih razvojnih okruženja, editora koda i dokumentacije koje koristi ciljna programerska grupa. Međutim, najbolje je pokrenuti studije upotrebljivosti ranije, a ne kasnije, u ciklusu proizvoda – prema tome, nemojte da odlažete organizovanje studije samo zato što još nije gotov ceo proizvod.

STEPHEN TOUB - Nije vam čak potrebna ni prava implementacija da biste dobili povratnu informaciju za upotrebljivost vašeg API-a. Iako je najbolje da imate nešto što programeri mogu da pokrenu i vide rezultate njihovog eksperimenta, rane povratne informacije o dizajnu možete da dobijete i ako imate samo API u kom mogu da kodiraju i kompajliraju, što znači da sve vaše implementacije mogu da budu zaustavljene da ne postanu neuspešne; nije važno jer neće biti pozivane. Da li su programeri intuitivno pronašli uključene relevantne tipove? Da li su mogli da prepoznaju obrasce koji su upotrebljeni za pristup funkcionalnosti? Da li je IntelliSense mogao da im pomogne i na smislen način ih vodi kroz API? Da li su pristupili problemima na način na koji ste mislili da će pristupiti? Da li su rutinski pretraživali elemente imenovane drugačije?

Često su formalne studije upotrebljivosti nepraktične za male programerske timove i radne okvire koji su namenjeni relativno maloj grupi programera. U takvim slučajevima može da se izvede neformalna studija. Neformalna studija podrazumeva da nekome, ko nije upoznat sa dizajnom, date prototip biblioteke i zatražite da provede 30 minuta pišući jednostavan program, pa posmatrate kako se snalazi u dizajnu, što je odličan način da otkrijete neka problematična pitanja u vezi sa API dizajnom.

✓ **URADITE** –organizujete studije upotrebljivosti za testiranje API-a u glavnim scenarijima.

Ove studije bi trebalo da budu organizovane rano u ciklusu razvoja, zato što najteži problemi upotrebljivosti često zahtevaju znatne promene u dizajnu. Većina programera bi trebalo da bude u mogućnosti da piše kod za glavne scenarije bez većih problema; ako ne mogu, potrebno je da redizajnirate API. Iako je redizajn skupa praksa, otkrili smo da na duge staze štedi resurse, zato što je trošak ispravke neupotrebljivog API-a, bez uvođenja promena koje menjaju postojeći kod, ogroman.

U sledećem odeljku ćemo opisati važnost dizajniranja API-a tako da prvi susret ne bude obeshrabrujuć. To se naziva princip niske barijere za ulaz.

2.2.2 Princip niske barijere za ulaz

Danas, mnogi programeri očekuju da veoma brzo nauče osnove novih radnih okvira. Oni to žele da urade eksperimentisanjem sa delovima radnog okvira na ad hoc osnovi, a da odvoje vreme za potpuno razumevanje cele arhitekture ako otkriju da je određena funkcija interesantna, ili ako bi trebalo da se pomere van granica jednostavnih scenarija. Početni susret sa loše dizajniranim API-om može da ostavi trajan utisak kompleksnosti i da obeshrabri neke programere da koriste radni okvir. Zbog toga je veoma važno da radni okvir obezbedi veoma nisku barijeru za programere koji žele da eksperimentišu sa radnim okvirom.

Principi projektovanja radnog okvira

Radni okviri moraju da obezbede nisku barijeru za jednostavno eksperimentisanje za korisnike koji nisu stručnjaci.

Mnogi programeri žele da eksperimentišu sa API-om da bi otkrili šta on izvršava, a zatim da prilagode kod polako, da bi njihov program izvršavao ono što oni žele da izvršava.

> **PAUL VICK** - Većina programera, bez obzira na jezik u kom rade, uči radeći. Dokumentacija može da pomogne davanjem početne ideje o onome što bi trebalo da se desi, ali svi znamo da nikada nećete stvarno načiti kako nešto funkcioniše dok ne počnete da eksperimentišete u pokušaju da URADITE nešto korisno. Visual Basic, konkretno, podstiče ovu vrstu eksperimentalnog pristupa programiranju. Iako nikada ne izbegavamo razmišljanje i planiranje unapred, pokušavamo da proces učenja i programiranja učinimo kontinualnim. Ovaj tok podstiče pisanje API-a koji su jasni i ne zahtevaju kompleksno znanje interakcije više objekata. (U stvari, to važi za više jezika, ne samo za Visual Basic).

Neki API-i se podvrgavaju eksperimentisanju, a neki ne. Da bi bio jednostavan za eksperimentisanje, API mora da izvršava sledeće:

- Da omogućava jednostavnu identifikaciju odgovarajućeg skupa tipova i članova za uobičajene zadatke programiranja. Imenski prostori, namenjeni da sadrže API-e uobičajenih scenarija koji uključuju 500 tipova, od kojih je samo nekoliko važnih u uobičajenim scenarijima, nisu jednostavni za eksperimentisanje. Isto važi i za tipove glavnih scenarija sa mnogo članova koji su namenjeni samo za veoma napredne scenarije.

> **CHRIS ANDERSON** - U ranim danima Windows Presentation Foundation (WPF) projekta, suočili smo se upravo sa ovim problemom. Imali smo zajednički osnovni tip pod nazivom Visual iz kog su izvedeni skoro svi naši elementi. Problem je bio to što je ovaj tip uveo članove koji su direktno bili u konfliktu sa objektnim modelom u više izvedenih elemenata, posebno oko potomaka. Visual je imao jednu hijerarhiju vizuelnih elemenata potomaka za renderovanje, ali naši elementi su želeli da uvedu potomke specifične za domen (kao što TabControl prihvata samoTabPages). Naše rešenje je bilo da kreiramo VisualOperations klasu koja je imala statičke članove koji su funkcionisali u Visual tipu, umesto da komplikujemo objektni model svakog elementa.

- Da omogućava programeru da koristi API odmah, bez obzira na to da li izvršava ono što programer želi da izvršava. Radni okvir koji zahteva obimnu inicijalizaciju ili instanciranje nekoliko tipova, a zatim njihovo povezivanje, nije jednostavan za eksperimentisanje. Slično, API-i bez preklopljenih pogodnosti (preklopljeni članovi sa kratkim listama parametara) ili loša podrazumevana podešavanja za svojstva, postavljaju visoku barijeru za programere koji žele da eksperimentišu sa API-om.

CHRIS ANDERSON - Zamislite objektni model kao mapu: Trebalo bi da postavite jasne znakove koji objašnjavaju kako da se stigne sa jednog mesta na drugo. Želite svojstvo koje će jasno uputiti ljude šta ono izvršava, koje vrednosti prihvata i šta se dešava kad ga podesite. Ukazivanje na apstraktni osnovni tip, bez očiglednih izvođenja, je veoma loše. Primer za to je način na koji su animacije prikazane u WPF-u: tip animacije je bio Timeline, ali se ništa u imenskom prostoru ne završava rečju „Timeline." Ispostavlja se da je tip Animation izveden iz tipa Timeline i da postoji mnogo tipova kao što su DoubleAnimation, ColorAnimation, i tako dalje, ali ne postoji konekcija između tipa svojstva i validnih stavki kojima se popunjava svojstvo.

- Da omogućava jednostavno pronalaženje i ispravljanje grešaka izazvanih neodgovarajućom upotrebom API-a. Na primer, API-i bi trebalo da generišu izuzetak koji jasno opisuje šta je potrebno da se uradi da bi rešili problem.

CHRIS SELLS - U sopstvenom programiranju ja mnogo volim poruke o greškama koje govore šta sam pogrešio i kako da to ispravim. Najčešće sve što dobijem je informacija šta sam pogrešio, a mene u stvari zanima kako da to rešim.

Sledeći saveti će vam pomoći da budete sigurni da je radni okvir pogodan za programere koji žele da uče eksperimentisanjem.

✓ **URADITE** – uverite se da imenski prostor oblasti glavne funkcije sadrži samo tipove koji se koriste u najčešćim scenarijima. Tipovi koji se koriste u naprednim scenarijima trebalo bi da budu postavljeni u pod-imenski prostor.

Na primer, System.Net imenski prostor obezbeđuje API-e za glavne scenarije umrežavanja. U System.Net.Sockets pod-imenski prostor postavljeni su API-i naprednijih priključaka.

> **ANTHONY MOORE** - Suprotno ovome je takođe tačno, što bi moglo da se navede kao „Ne postavljaj uobičajno upotrebljavan tip u imenski prostor sa ređe upotrebljavanim tipovima". StringBuilder je primer nečega što smo, kasnije, želeli da smo uključili u System imenski prostor. Ovaj tip se nalazi u System.Text imenskom prostoru, ali se češće koristi od drugih tipova koji se u njemu nalaze i nije blisko povezan sa njima.

Prema tome, ovo je jedini element u System imenskom prostoru koji odstupa od pravila. Uglavnom, problem je što imamo previše retko upotrebljavanih tipova u tom imenskom prostoru.

✓ **URADITE** – obezbedite jednostavna preklapanja konstruktora i metoda. Jednostavno preklapanje ima veoma mali broj parametara, i svi parametri su osnovni oblici.

✗ **NEMOJTE**-imati članove namenjene za napredne scenarije u tipovima namenjenim za osnovne scenarije.

> **BRAD ABRAMS** - Jedan od važnih principa projektovanja .NET Framework-a bio je pojam sabiranja oduzimanjem. Odnosno, uklanjanjem funkcija iz radnog okvira (ili ne dodavanje funkcija), mogli smo programere da učinimo produktivnijim, zato što postoji manje koncepata koje bi trebalo da koriste. Izostavljanje višestrukog nasleđivanja je klasičan primer dodavanja oduzimanjem, na CLR nivou.

✗ **NEMOJTE** – zahtevati od korisnika da eksplicitno instanciraju više od jednog tipa u najosnovnijim scenarijima.

> **KRZYSZTOF CWALINA** - Izdavači knjiga kažu da je broj primeraka knjige koji će se prodati obrnuto proporcionalan broju jednačina u knjizi. Verzija ovog pravila za dizajnera radnog okvira je: Broj programera koji će koristiti radni okvir je obrnuto proporcionalan broju eksplicitnih poziva konstruktora potrebnih za implementiranje deset najboljih jednostavnih scenarija.

✗ **NEMOJTE** – da zahtevate da korisnici izvršavaju obimne inicijalizacije pre nego što mogu da počnu programiranje osnovnih scenarija.

API-i glavnog scenarija bi trebalo da budu projektovani tako da zahtevaju minimalnu inicijalizaciju. Idealno bi bilo da podrazumevani konstruktor ili konstruktor sa jednim jednostavnim parametrom bude dovoljan za početak rada upotrebom tipa koji je projektovan za osnovne scenarije.

```
var zipCodes = new Dictionary<string,int>();
zipCodes.Add("Redmond",98052);
zipCodes.Add("Sammamish",98074);
```

Ako je potrebna inicijalizacija, izuzetak koji dovodi do neizvršenja inicijalizacije bi trebalo jasno da objasni šta je potrebno da se izvrši.

STEVEN CLARKE - Od prvog izdanja ove knjige mi smo izvršili značajne studije upotrebljivosti u ovoj oblasti. Iznova i iznova smo primetili da tipovi koji zahtevaju obimnu inicijalizaciju značajno podižu barijeru za ulaz. Posledice toga su da će neki programeri odlučiti da ne upotrebe te tipove i potražiti nešto drugo što će upotrebiti za rad, a neki programeri će upotrebiti tip nepravilno, a samo nekolicina programera će, na kraju, shvatiti kako da koristi tip pravilno.

ADO.NET je primer oblasti funkcije koje naši korisnici smatraju teškom za upotrebu, zbog obimne inicijalizacije koju zahteva. Čak i u najjednostavnijim scenarijima, korisnici očekuju da razumeju kompleksne interakcije i zavisnosti između nekoliko tipova. Da bi upotrebili ovu funkciju, čak i u najjednostavnijim scenarijima, korisnici moraju da instanciraju i povežu nekoliko objekata (instance DataSet, DataAdapter, SqlConnection i SqlCommand). Imajte na umu da su mnogi od ovih problema rešeni u .NET Framework-u 2.0 dodavanjem pomoćnih klasa, koje znatno pojednostavljuju osnovne scenarije.

✓ **URADITE** –obezbedite dobra podrazumevana podešavanja za sva svojstva i parametre (upotrebom preklapanja pogodnosti), ako je moguće.

System.Messaging.MessageQueue je dobra ilustracija ovog koncepta. Komponenta može da šalje poruke nakon što konstruktoru prosledi znakovni niz putanje i pozove metod Send. Prioritet poruke, algoritmi enkripcije i druga svojstva poruke mogu da budu prilagođeni dodavanjem koda u jednostavan scenario.

```
var ordersQueue = new MessageQueue(path);
ordersQueue.Send(order); // uses default priority, encryption, etc.
```

Ove preporuke ne mogu se primenjivati naslepo. Dizajneri radnih okvira bi trebalo da izbegavaju obezbeđivanje podrazumevanih podešavanja, ako podrazumevana podešavanja mogu da vode korisnika u pogrešnom smeru. Na primer, podrazumevano podešavanje nikada ne bi trebalo da rezultira bezbednosnom rupom ili kodom koji se loše izvršava.

STEPHEN TOUB - Kada donosite odluke o „podrazumevanim podešavanjima", takođe je važno da razumete primarne slučajeve upotrebe za API – u meri u kojoj je to moguće – i da predvidite očekivane slučajeve upotrebe u budućnosti. Jedan od mojih deset omiljenih „voleo bih da to mogu da ponovim" slučajeva je iz System.Threading.Tasks imenskog prostora. Prvo smo projektovali API-e sa fokusom na paralelizam zasnovan na CPU-u, ali su vremenom primarni slučajevi upotrebe bili mnogo više usmereni na asinhronost zasvnovanu na IO-u, a neka od naših podrazumevanih podešavanja su bila mnogo bolje usklađena za paralelizam, a ne za asinhronost. Vremenom smo rešili ove probleme kao deo dodavanja API-a jednostavnijih za upotrebu, ali inicijalni problemi i rezultirajuće poteškoće su se zadržale za programere koji su koristili originalne API-e.

JEREMY BARTON - Suprotnost ovde je veoma važna i može biti veoma teško pronaći odgovarajuću ravnotežu. API-i za .NET Cryptography uključuju veliki broj tipova koji obezbeđuju podrazumevana podešavanja za pokušaj da budu i jednostavni i bezbedni. Nažalost, „jednostavni" je održivo, ali „bezbedni" je pokretna meta. Ponekad činite korisnicima uslugu obezbeđivanjem podrazumevanih podešavanja, a ponekad ćete im naneti štetu.

✓ **URADITE** –izveštavajte o pogrešnoj upotrebi API-a upotrebom izuzetaka.

Izuzeci bi trebalo jasno da opisuju uzrok, kao i način na koji bi programer trebalo da modifikuje kod da bi rešio problem. Na primer, `EventLog` komponenta zahteva da `Source` svojstvo bude podešeno pre nego što događaji budu napisani. Ako svojstvo `Source` nije podešeno pre nego što je pozvan metod `WriteEntry`, biće podignut izuzetak u kom je navedeno da, `"Source property was not set before writing to the event log."`.

> **STEVEN CLARKE** - Posmatrali smo mnoge programere, u našim studijama upotrebljivosti, koji su smatrali ovakve izuzetke najboljom vrstom dokumentacije koju API može da obezbedi. Pruženi savet je uvek u kontekstu onoga što programer pokušava da postigne i podržava pristup učenja kroz rad, koji je omiljen kod mnogih programera.

U sledećem odeljku ćemo opisati važnost maksimalnog samodokumentovanja objektnog modela.

2.2.3 Princip samodokumentovanja objektnih modela

Mnogi radni okviri se sastoje od više stotina, ako ne i hiljada, tipova i znatno više članova i parametara. Programeri koji koriste takve radne okvire zahtevaju mnogo uputstava i česte podsetnike namene i pravilne upotrebe API-a. Referentna dokumentacija sama po sebi ne može da zadovolji zahteve. Ako je potrebno referencirati dokumentaciju za pronalaženje odgovora na najjednostavnija pitanja, to može da bude prilično dugotrajno i da prekida tok rada programera. Štaviše, kao što je ranije pomenuto, mnogi programeri radije kodiraju po principu pokušaja i greške, a pribegavaju čitanju dokumentacije samo kada ih intuicija prevari.

Iz svih tih razloga, veoma je važno da dizajnirate API-e koji ne zahtevaju da programeri čitaju dokumentaciju svaki put kada žele da izvrše jednostavan zadatak. Otkrili smo da praćenje jednostavnog skupa saveta može da pomogne programerima da kreiraju intuitivne API-e koji su relativno samo-dokumentujući.

Princip projektovanja radnog okvira

U jednostavnim scenarijima, radni okviri moraju da budu uptorebljivi bez potrebe za dokumentacijom.

CHRIS SELLS - Nemojte nikada da potcenite moć IntelliSense-a kada procenjujete kako će programer učiti da koristi vaš radni okvir. Ako je API intuitivan, IntelliSense je 80 procenata svega što će novim programerima biti potrebno da budu srećni i uspešni tokom upotrebe vaše biblioteke. Optimizujte za IntelliSense.

KRZYSZTOF CWALINA - Referentna dokumentacija je i dalje veoma važan deo radnog okvira. Nemoguće je dizajnirati potpuno samo-dokumentujuć API. Različiti ljudi, u zavisnosti od svojih veština i iskustava koja su stekli, otkriće različite oblasti radnog okvira koji su jasni. Takođe, dokumentacija ostaje veoma važna za mnoge korisnike koji odvoje vreme da prouče da bi dobro razumeli ceo dizajn radnog okvira. Za te korisnike, informativna, koncizna i kompletna dokumentacija je važna kao i jasni objektni modeli.

✓ **URADITE** –uverite se da su API-i intuitivni i da mogu uspešno da se upotrebe u osnovnim scenarijima, bez pregleda referentne dokumentacije.

✓ **URADITE** –obezbedite dobru dokumentaciju za sve API-e.

✓ **URADITE** –obezbedite primere koda koji ilustruju kako se koriste najvažniji API-i u uobičajenim scenarijima.

Ne mogu svi API-i da budu jasni, a neki programeri će želeti bolje da razumeju API-e pre nego što počnu da ih koriste.

Da bi radni okvir bio samo-dokumentujuć, morate da obratite pažnju kada birate nazive i tipove, dizajnirate izuzetke i tako dalje. U sledećim odeljcima ćemo razjasniti neka najvažnija razmatranja koja se odnose na dizajn samo-dokumentujućih API-a.

2.2.3.1 Imenovanje

Najjednostanija prilika za kreiranje samo-dokumentujućeg radnog okvira, a koja ipak često nedostaje, jeste rezervisanje jednostavnih i intuitivnih naziva za tipove koje očekujete da će programeri upotrebiti (instancirati) u najuobičajenijim scenarijima. Dizajneri radnih okvira često „iskoriste" najbolje nazive za ređe upotrebljavane tipove, koje većina korisnika neće upotrebiti.

Na primer, imenovanje apstraktne osnovne klase File, a zatim obezbeđivanje konkretnog tipa `NtfsFile`, funkcioniše dobro, ako očekujete da će svi korisnici razumeti hijerarhiju nasleđivanja pre nego što počnu da koriste API-e. Ako korisnici ne razumeju hijerarhiju, prvo što će pokušati da upotrebe, najčešće neuspešno, je tip `File`. Iako ovo imenovanje funkcioniše dobro u pogledu objektno-orijentisanog dizajna (ipak, `NtfsFile` je vrsta `File`), test upotrebljivosti će biti neuspešan zato što je `File` naziv za koji će većina programera pomisliti da koristi za programiranje.

> **KRZYSZTOF CWALINA** - Dizajneri .NET Framework-a proveli su mnogo vremena u diskusiji o alternativama imenovanja za glavne tipove. Većina identifikatora u .NET-u ima dobro izabrane nazive. Slučajevi u kojima izbor naziva nije toliko dobar doveli su do fokusiranja na koncepte i apstrakcije, umesto na najbolje scenarije.

Još jedna preporuka je da upotrebite opisne nazive identifikatora, koji jasno navode šta svaki metod radi i šta predstavlja svaki tip i parametar. Dizajneri radnih okvira ne bi trebalo da se plaše da budu opširni prilikom biranja naziva identifikatora. Na primer, `EventLog.DeleteEventSource (string source, string machineName)` se možda smatra opširnim, ali mi mislimo da ima pozitivnu upotrebnu vrednost.

Opisni nazivi metoda su jedino mogući za metode koji imaju jednostavne i jasne semantike. To je još jedan razlog što je izbegavanje kompleksne semantike odličan osnovni princip dizajna koji bi trebalo da pratite.

Uopštena poenta je da bi trebalo da izaberete nazive identifikatora veoma pažljivo. Izbori naziva su jedan od najvažnijih odluka koje bi dizajner radnog okvira trebalo da donese. Veoma je teško i skupo promeniti nazive identifikatora nakon što je API izdat.

✓ **URADITE** –raspravljajte o izboru imenovanja identifikatora, jer je to značajan deo pregleda specifikacije.

Koji su tipovi kojim počinje većina scenarija? Koji su nazivi o kojima će većina ljudi razmišljati kada pokušava da implementira ovaj scenario? Da li su nazivi uobičajenih tipova ono na šta će korisnici prvo pomisliti? Na primer, pošto je „File" naziv koji će prvo pasti na pamet ljudima kada koriste I/O scenarije, glavni tip za pristup fajlovima bi trebao da ima naziv `File`.

Takođe, trebalo bi da diskutujete o najčešće upotrebljavanim metodima najčešće upotrebljavanih tipova i svim njihovim parametrima. Može li svako, ko je upoznat sa vašom tehnologijom, ali ne sa datim specifičnim dizajnom, da prepozna i pozove ove metode brzo, tačno i jednostavno?

✗ **NEMOJTE** se plašiti da upotrebite opširne nazive identifikatora ako to API čini samo-dokumentujućim.

Većina naziva identifikatora bi trebalo jasno da navede šta radi svaki metod i koje sve tipove i parametre predstavlja.

> **BRENT RECTOR** - Programeri čitaju nazive identifikatora stotinu, ako ne i hiljadu puta više nego što ih kucaju. Moderni editori čak i minimalizuju kucanje. Duži nazivi omogućavaju programerima da mnogo brže pronađu odgovarajući tip ili član pomoću IntelliSense-a. Osim toga, kod koji koristi tipove sa dobro izabranim nazivima identifikatora, mnogo je razumljiviji i lakši za održavanje od dugih termina.
>
> Napomena posebno za programere jezika zasnovanih na C-u: Oslobodite se okova smanjene produktivnosti izazvane navikom kriptičnog imenovanja identifikatora.

✓ **RAZMOTRITE** rano uključivanje stručnih pisaca u procesu dizajna. Oni mogu da budu odličan resurs za otkrivanje dizajna sa lošim izborom naziva i dizajna koji bi bilo veoma teško objasniti korisnicima.

✓ **RAZMOTRITE** rezervisanje najboljih naziva tipova za najčešće upotrebljavane tipove.

Ako planirate da dodajete više API-a visokog nivoa u buduću verziju, ne plašite se da rezervišete najbolje nazive u prvoj verziji radnog okvira za buduće API-e.

> **ANTHONY MOORE** - Postoje i drugi razlozi da izbegavate nazive koji su previše uopšteni, čak i ako ne planirate da upotrebite naziv kasnije. Najspecifičniji nazivi pomažu da API bude razumljiviji i čitljiviji. Ako neko vidi uopšten naziv u kodu, on će verovatno pretpostaviti da ima vrlo uopštenu aplikaciju, pa je stoga obmanjujuće upotrebiti uopšten naziv za nešto specijalizovano. Opisniji naziv takođe može da pomogne u identifikaciji sa kojim scenarijom ili tehnologijom je tip povezan.

2.2.3.2 Izuzeci

Izuzeci igraju veoma važnu ulogu u dizajniranju samo-dokumentovanih radnih okvira. Oni bi trebalo da prenose tačnu upotrebu programeru putem poruka izuzetaka. Na primer, sledeći primer koda bi trebalo da podigne izuzetak sledećom porukom: `"Source property was not set before writing to the event log."`

```
// C#
var log = new EventLog();
// The log source is not set yet.
log.WriteEntry("Hello World");
```

✓ **URADITE** –upotrebite poruke izuzetka da biste preneli programeru greške u upotrebi radnog okvira.

Na primer, ako korisnik zaboravi da podesi `Source` svojstvo `EventLog` komponente, svaki poziv metoda koji zahteva da bude podešen izvor, trebalo bi jasno da prikaže poruku izuzetka. U Poglavlje 7 uključeni su saveti o dizajnu izuzetaka i o porukama izuzetaka.

2.2.3.3 Strogo tipiziranje

Strogo tipiziranje je verovatno jednini najvažniji faktor u određivanju koliko su API-i intuitivni. Naravno, pozivanje metoda `Customer`.Name je jednostavnije od pozivanja metoda `Customer.Properties["Name"]`. Takođe, `Name` svojstvo koje vraća naziv kao String je upotrebljivije nego da svojstvo vraća `Object`.

Postoje slučajevi u kojima su tabele svojstva, kasno povezani pozivi i drugi labavo tipizirani API-i potrebni, ali bi oni trebalo da budu izuzetak pravila, umesto uobičajena praksa. Štaviše, dizajneri bi trebalo da razmotre obezbeđivanje strogo tipiziranih pomagača za najčešće operacije koje će korisnik vršiti na API sloju koji nije strogo tipiziran. Na primer, tip `Customer` može da ima property bag, ali takođe da obezbedi strogo tipizirane API-e za najčešće upotrebljavana svojstva, kao što su `Name`, `Address`, i tako dalje.

✓ **URADITE** –obezbedite strogo tipizirane API-e ako je to ikako moguće.

Nemojte se oslanjati samo na slabo tipizirane API-e kao što su property bags. U slučajevima u kojima je potreban property bag, obezbedite strogo tipizirana svojstva za najčešće upotrebljavana svojstva u tabeli.

VANCE MORRISON - Strogo tipiziranje (a stoga i mnogo bolji IntelliSense) je ključan razlog zašto je .NET radne okvire jednostavnije „učiti programiranjem" nego tipičan COM API. S vremena na vreme, ja i dalje moram da upotrebim funkcionalnost koja je izložena kroz COM i sve dok ostane strogo tipiziran, sve je u redu. Ali, previše često API-i vraćaju, ili koriste generički objekat ili parametar znakovnog niza, ili prosleđeni DWORD, kada je potrebno nabrajanje i potrebno mi je deset puta više vremena da otkrijem šta bi tačno trebalo da bude prosleđeno.

2.2.3.4 Konzistentnost

Konzistentnost sa postojećim API-ima koji su već poznati korisniku je još jedna moćna tehnika za dizajniranje samo-dokumentujućih radnih okvira. To uključuje konzistentnost sa drugim API-ima u .NET-u, kao i nekim zastarelim API-ima. Prema tome, ne bi trebalo da koristite zastarele API-e, ili loše dizajnirane postojeće API-e radnog okvira, kao izgovor za izbegavanje praćenja smernica opisanih u ovoj knjizi – ali isto tako ne bi trebalo proizvoljno da menjate dobro uspostavljene obrasce i dizajn bez razloga.

✓ **URADITE** –uverite se da postoji konzistentnost u .NET-u i drugim radnim okvirima koje će korisnici verovatno upotrebiti.

Konzistentnost je odlična za opštu upotrebljivost. Ako je korisnik upoznat sa nekim delom radnog okvira na koji podseća vaš API, korisnik će videti vaš dizajn kao prirodan i intuitivan. Vaš API bi trebalo da se razlikuje od drugih .NET API-a samo na mestima na kojima postoji nešto jedinstveno.

2.2.3.5 Ograničavanje apstrakcija

API-i uobičajenog scenarija ne bi trebalo da koriste mnogo interfejsa i apstraktnih klasa, ali bi trebalo da odgovaraju fizičkim i dobro poznatim logičkim delovima sistema.

Kao što je ranije pomenuto, standardne metodologije objektno-orijentisanog dizajna su namenjene za kreiranje dizajna koji je optimizovan za jednostavnije održavanje osnove koda. To ima smisla, zato što je trošak održavanja najveći deo ukupnog troška razvoja softverskog proizvoda. Jedan način da poboljšate održavanje je upotreba apstrakcije putem interfejsa ili apstraktnih klasa. Zbog toga, moderne metodologije dizajna teže da kreiraju mnogo njih.

Problem je to što radni okviri sa mnogo apstrakcije nameću korisnicima da postanu stručnjaci u arhitekturi radnog okvira, pre nego što počnu da implementiraju čak i najjednostavnije scenarije. Međutim, većina programera nema želju ili poslovno opravdanje da postanu stručnjaci u svim API-ima koje takav radni okvir obezbeđuje. Za jednostavne scenarije, programeri zahtevaju da API-i budu dovoljno jednostavni da se upotrebe bez potrebe razumevanja kako se uklapa cela oblast funkcije. To je nešto za šta nisu optimizovane standardne metodologije dizajna, a nikad ni neće biti optimizovane.

Naravno, apstrakcije imaju svoje mesto u dizajnu radnog okvira. Na primer, apstrakcije mogu da budu ekstremno korisne u poboljšavanju mogućnosti testiranja i uopštene proširivosti radnih okvira. Takva proširivost je često moguća zbog dobro dizajniranih apstrakcija. U Poglavlju 6 smo opisali projektovanje proširivih API-a što bi trebalo da vam pomogne da uspostavite odgovarajuću ravnotežu između previše i premalo proširivosti.

✗ **IZBEGAVAJTE** mnogo apstrakcija u API-ima glavnog scenarija.

KRZYSZTOF CWALINA - Apstrakcije su skoro uvek potrebne, ali previše apstrakcije ukazuje na preinženjerisane sisteme. Dizajneri radnih okvira bi trebalo da budu pažljivi i svesni da dizajniraju za kupce, a ne za svoje intelektualno zadovoljstvo.

JEFF PROSISE - Dizajn sa previše apstrakcije može da utiče i na performansu. Ja sam jednom radio sa klijentom koji je izvršavao reinženjering svojih proizvoda tako da u njih uključi objektno-orijentisani dizajn. Oni su modelovali sve kao klasu i završili su sa nekim smešno duboko ugnežđenim hijerarhijama objekata. Deo namere redizajna je bio da se poboljša performansa, ali „poboljšani" softver se pokretao četiri puta sporije od originala!

JVANCE MORRISON - Svako ko je imao „zadovoljstvo" da ispravlja greške C++ STL biblioteka zna da je apstrakcija mač sa dve oštrice. Ako ima previše apstrakcije, kod postaje veoma težak za razumevanje, zato što morate da pamtite šta zapravo znače apstraktni nazivi u scenariju. Preterivanje sa generičkim tipovima i nasleđivanjem su uobičajeni simptomi da ste pregeneralizovali kod.

CHRIS SELLS - Često se kaže da se bilo koji problem u računarskoj nauci može rešiti dodavanjem sloja apstrakcije. Nažalost, problemi obrazovanja programera često su uzrokovani njima.

2.2.4 Princip slojevite arhitekture

Nisu svi programeri zaduženi da rešavaju iste vrste problema. Različiti programeri često zahtevaju i očekuju različite nivoe apstrakcije i različite količine kontrole od radnih okvira koje koriste. To su programeri koji obično koriste API-e vrednosti C++ ili C#, koji su skupi i moćni. Ove API-e nazivamo API-ima niskog nivoa zato što često obezbeđuju nizak nivo apstrakcije. Nasuprot tome, neki programeri obično koriste API-e vrednosti C# ili VB.NET, koji su optimizovani za produktivnost i jednostavnost. Ove API-e nazivamo API-ima visokog nivoa zato što obezbeđuju viši nivo apstrakcije. Upotrebom slojevitog dizajna moguće je izgraditi jedan radni okvir koji ispunjava ove različite zahteve.

Princip projektovanja radnog okvira

Slojeviti dizajn omogućava da obezbedite moć i lakoću upotrebe u istom radnom okviru.

PAUL VICK - Deo razloga za prebacivanje Visual Basica u .NET platformu je bila činjenica da se mnogo VB programera suočavalo sa problemima kada je trebalo da upotrebe API-e niskog nivoa za pristup specifičnim funkcionalnostima koje nisu bile dostupne u API-ima visokog nivoa koje smo obezbedili. Činjenica da VP programeri mogu da provedu mnogo vremena na početku rada brzo razvijajući svoje aplikacije upotrebom API-a visokog nivoa, ne menja činjenicu da će pre ili kasnije većina programera morati da podesi svoje aplikacije, a to obično uključuje upotrebu API-a nižeg nivoa, da bi se postigli ti dodati bitovi funkcionalnosti. Prema tome, dizajn API-a niskog nivoa bi trebalo da uzme u obzir i VB programere.

Osnovna smernica za izgradnju jednog radnog okvira koji je namenjen širem spektru programera je faktorisanje skupa API-a na tipove niskog nivoa, koji otkrivaju bogatstvo i moć, i na tipove visokog nivoa, koji obuhvataju niže slojeve pomoću pomoćnih API-a.

Ovo je veoma moćna tehnika pojednostavljivanja. U jednoslojnom API-u često ste primorani da odlučite da li ćete imati kompleksniji dizajn ili dizajn koji neće podržavati neke scenarije. Ako postoje moćni slojevi niskog nivoa, to obezbeđuje slobodu primene API-a visokog nivoa za glavne scenarije.

U nekim slučajevima, jedan od slojeva možda neće biti potreban. Na primer, neke oblasti funkcije mogu da otkriju samo API-e niskog nivoa.

.NET JSON API-i su primer takvog slojevitog dizajna. Za masu snage i izražajnosti, tip `Utf8JsonReader` obezbeđuje JSON analizator niskog nivoa, koji omogućava programerima da kodiraju u pojedinačnim tokenima u JSON korisnim podacima. Međutim, .NET takođe ima tipove `JsonDocument` i `JsonElement`, koji se nadograđuju na `Utf8JsonReader` i omogućavaju programerima da kodiraju u konceptima višeg nivoa, kao što su strukture dokumenta, bez potrebe da brinu o dubini objekta ili o neobeleženim znakovnim nizovima. Tipovi imaju konzistentno ponašanje i identifikatore, ali su na različitim slojevima koji ciljaju različite scenarije i programersku publiku.

Postoje dva glavna pristupa faktorisanja imenskog prostora za API slojeve:

- Izlaganje slojeva u posebne imenske prostore.
- Izlaganje slojeva u isti imenski prostor.

2.2.4.1 Izlaganje slojeva u posebne imenske prostore

Jedan način faktorisanja radnog okvira je da postavite tipove visokog nivoa i niskog nivoa u različite, ali srodne imenske prostore. Prednost ovoga je skrivanje tipova niskog nivoa iz glavnih scenarija, ali se oni ne postavljaju predaleko van dometa kada bi programeri trebalo da implementiraju kompleksnije scenarije.

API-i umrežavanja .NET-a su faktorisani na ovaj način. Tip niskog nivoa, `System.Net.Sockets.Socket`, tip srednjeg nivoa, `System.Net.Security.SslStreami`, tip visokog nivoa, `System.Net.Http.HttpClient`, nalaze se u različitim imenskim prostorima. Naposletku, `HttpClient` tip se oslanja na `Socket` `iSslStream` u svojoj implementaciji, ali većina programera koji koriste HTTP, mogu da upotrebe `HttpClient` bez potrebe za tipovima niskog nivoa.

Većina radnih okvira bi trebalo da prati pristup faktorisanja imenskog prostora.

2.2.4.2 Izlaganje slojeva u isti imenski prostor

Drugi način za faktorisanje radnog okvira je postavljanje tipova visokog nivoa i niskog nivoa u isti imenski prostor. To ima prednosti, jer obezbeđuje automatsko vraćanje na kompleksniju funkcionalnost kada je to potrebno. Nedostatak je taj što kompleksniji tipovi u imenskim prostorima otežavaju neke scenarije, čak i ako nisu upotrebljeni kompleksni tipovi.

Ova faktorizacija najbolje funkcioniše za jednostavne funkcije. Na primer, `System.Text` imenski prostor uključuje tipove niskog nivoa, kao što su `Encoder` i `Decoder` klase, i hijerarhiju klase `Encoding` višeg nivoa.

> **STEVEN CLARKE** - Vodite računa da razmišljate i o ponašanju izvršenja slojevitog API-a. Na primer, uverite se da ako programer radi na jednom sloju, ne očekuje da će uhvatiti izuzetke podignute iz drugog sloja. Želećete da budete sigurni da, u pisanju, čitanju i razumevanju koda, programeri treba da brinu samo o tome šta se dešava na jednom sloju i da slobodno mogu da smatraju druge slojeve crnim kutijama.

✓ **RAZMOTRITE** upotrebu slojevitog radnog okvira sa API-ima visokog nivoa, koji su optimizovani za produktivnost i upotrebu API-a niskog nivoa, koji su optimizovani za moć i izražajnost.

✗ **IZBEGAVAJTE** mešanje API-a niskog i visokog nivoa u jednom imenskom prostoru ako su API-i niskog nivoa veoma kompleksni (na primer, sadrže mnogo tipova).

✓ **URADITE** –uverite se da su slojevi jedne oblasti funkcije dobro integrisani. Programeri bi trebalo da budu u mogućnosti da počnu programiranje upotrebom jednog od slojeva, a zatim da promene kod za upotrebu drugog sloja, bez ponovnog pisanja cele aplikacije.

REZIME

Kada dizajnirate radni okvir, veoma je važno da budete svesni da je publika veoma raznolika, u pogledu potreba i nivoa veštine. Sledeći principe koji su opisani u ovom poglavlju bićete sigurni da je vaš radni okvir upotrebljiv za različite grupe programera.