C# i .NET Core projektni obrasci

Gaurav Aroraa
Jeffrey Chilberto

Pišite čist i održiv kod upotrebom ponovo upotrebljivih rešenja za uobičajene probleme projektovanja softvera

kompjuter biblioteka

Packt›

C# i .NET CORE

projektni obrasci

Gaurav Aroraa
Jeffrey Chilberto

kompjuter biblioteka

Packt›

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: Gaurav Aroraa
Jeffrey Chilberto

Prevod: Slavica Prudkov

Lektura: Miloš Jevtović

Slog: Zvonko Aleksić

Znak Kompjuter biblioteke:
Miloš Milosavljević

Štampa: „Pekograf“, Zemun

Tiraž: 500

Godina izdanja: 2019.

Broj knjige: 519

Izdanje: Prvo

ISBN: 978-86-7310-542-0

Hands-On Design Patterns with C# and .NETCore

Gaurav Aroraa

Jeffrey Chilberto

ISBN 978-1-78913-364-6

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд,
се добија на захтев

PREDGOVOR

Kada dizajniraju dobar softver, inženjeri se obično odlučuju za rešenja kojima se izbegava dupliranje. Mi normalno primenjujemo princip DRY (Don't Repeat Yourself) - često bez razmišljanja! Programeri obično razdvajaju funkcionalnosti i kreiraju ponovo upotrebljive metode i korisne klase.

Međutim, tokom godina kreirani su mnogi projektni obrasci za softver. Oni su korisna i ponovo upotrebljiva rešenja za probleme sa kojima se svakodnevno susrećemo. Postoji sve više samoukih programera ili onih koji na univerzitetu ne uče klasično softversko inženjerstvo ili računarske nauke, ali bi svako trebalo da uživa u prednostima decenijskog razvoja ovih odličnih projektnih obrazaca.

Gaurav i Jeffrey su sklopili najbolje i najčešće upotrebljavane obrasce i primenili ih na svet otvorenog koda .NET Corea i jezika C#. Prvo ćemo opisati OOP, klase i objekte i nastaviti nasleđivanjem, kapsuliranjem i polimorfizmom, koji obuhvataju principe dizajna, kao što su DRY, KISS i SOLID, primenjujući ih u klasičnim obrascima, što pomaže da se kreiraju jasni i pouzdani softveri.

Ova knjiga je popunjena stvarnim kodom koji jasno ilustruje kako da primenimo znanje u .NET Core i C# softver. Učićete kako da upotrebite Builder obrazac, Decorator obrazac, Factory obrazac, Visitor obrazac, Strategy obrazac i tako dalje.

Ove tehnike će biti primenjene u jednostavnoj aplikaciji, zatim u veb aplikaciji i u složenijim problemima koji uključuju konkurentnost i paralelizam. Zatim ćemo primeniti obrasce na makronivou, koristeći obrasce koji će nam pomoći da prebacimo projekte u cloud na skalabilan i održiv način.

Nadam se da će vam se knjiga dopasti kao što se dopala i nama dok smo je pisali. Takođe se nadam da uživate u upotrebi .NETCorea.

Scott Hanselman

Partner Program Manager—Microsoft

.NET and Open Source Community

Ova knjiga je blagoslov za svakog programera koji želi da poboljša svoje veštine programiranja, ali i koji želi da izgradi skalabilnu i robusnu aplikaciju koja je jednostavna za održavanje. Većina industrijskih standarda i najbolje tehnike opisane su upotrebom jasnih primera.

Osim o razvoju projektnih obrazaca, u ovoj knjizi se govori i o osnovnim principima arhitekture i nekim ključnim aspektima clouda, kao što su bezbednost i skaliranje.

Kao arhitekta rešenja, ja sam uključen svakodnevno u dizajniranje end-to-end rešenja, od razvojnih aspekata, do osnovne infrastrukture i bitova bezbednosti, i još sam impresioniran kvalitetom sadržaja i opsegom ove knjige - ona sadrži veoma obimnu listu obrazaca koje bi svi trebalo da pogledaju kada razmatraju razvoj i raspoređivanje novog radnog opterećenja.

Ovaj detaljan pregled objektno-orijentisanog programiranja (OOP-a) i .NET Corea koristan je za svakoga ko je zainteresovan za pisanje najboljih aplikacija.

Stephane Eyskens

Azure MVP

O AUTORIMA

Gaurav Aroraa je magistrirao računarske nauke. On je bivši Microsoft MVP, AlibabaCloud MVP, sertifikovan je kao Scrum trener, doživotni je član Computer Society ofIndia (CSI), savetodavni je član IndiaMentor, XEN for ITIL-F i APMG za PRINCEF and PRINCE-P. Gaurav je programer otvorenog koda, saradnik za TechNet Wiki i osnivač je Ovatic Systems Private Limiteda. U karijeri dugoj 21 godinu bio je mentor hiljadama studenata i industrijskih profesionalaca. Možete mu poslati poruku na twitteru na @g_arora.

Mojoj ženi Shuby Aroraa i mom anđelu (ćerki) Aarchi Aroraa, koje su mi dozvolile da „ukradem“ vreme za ovu knjigu od vremena koje je trebalo da provedem sa njima. Zahvaljujem se celom „Packt“ timu, posebno Chaitanyai, Akshitai i Nehai, koji su mi puno pomogli, a komunikacija sa njima bila mi je od velike pomoći, i Denimu Pintou, koji me je preporučio za ovu knjigu.

Jeffrey Chilberto je softverski konsultant specijalizovan za Microsoft tehnološki stek, uključujući Azure, BizTalk, ASP.NET, MVC, WCF i SQL Server, sa iskustvom u širokom rasponu industrija, kao i u bankarstvu, telekomunikacijama i zdravstvenoj zaštiti na Novom Zelandu, u Evropi, u Australiji i u Sjedinjenim Američkim Državama. Diplomirao je informatiku i računarske nauke i ima master stepen u računarskim naukama i inženjerstvu.

O RECENZENTIMA

Sjoukje Zaal je savetnik za upravljanje, Microsoft cloud arhitekta i Microsoft AzureMVP, sa više od 15 godina iskustva u obezbeđivanju stručnosti u arhitekturi, razvoju, konsultacijama i dizajnu. Ona radi u kompaniji „Capgemini“, koja je svetski lider u konsaltingu, tehnološkim uslugama i digitalnim transformacijama. Voli da prenosi svoje znanje i aktivna je u „Microsoft“ zajednici kao suosnivač Dutch korisničkih grupa SP&C NL i MixUG. Takođe je član odbora u Azure Thursdays. Napisala je nekoliko knjiga, piše blogove i aktivna je u MicrosoftTech Communityju. Takođe je član Diversity and Inclusion Advisory odbora.

Ephraim Kyriakidis ima skoro 20 godina iskustva u razvoju softvera. Poseduje diplomu inženjera elektrotehnike i softverskog inženjera sa univerziteta Aristotle University u Solunu, u Grčkoj. Koristi .NET još od verzije 1.0. U karijeri je uglavnom fokusiran na Microsoft tehnologije. Trenutno je zaposlen u „Siemens AG-u“ u Nemačkoj kao viši softverski inženjer.

„PACKT“ TRAŽI AUTORE KAO ŠTO STE VI

Ako ste zainteresovani da postanete autor za „Packt“, prijavite se na stranicu authors.packtpub.com. Sarađujemo sa hiljadama programera i tehničkih profesionalaca da bismo im pomogli da podele svoje mišljenje sa globalnom tehničkom zajednicom. Možete da podnesete osnovnu prijavu, da se prijavite za specifičnu temu za koju tražimo autore ili da pošaljete neke svoje ideje.

UVOD

U ovoj knjizi ćemo čitaocima pojasniti obrasce u modernom razvoju softvera, uz detaljno opisivanje specifičnih primera. Broj obrazaca upotrebljenih u razvojnim rešenjima je ogroman i često ih programeri koriste, ne znjući da to, u stvari, rade. U ovoj knjizi opisani su obrasci od koda niskog nivoa do koncepata visokog nivoa koji su upotrebljeni u rešenjima koja se pokreću u cloudu.

Iako mnogi od predstavljenih obrazaca ne zahtevaju specifičan jezik, upotrebićemo C# i .NET Core da bismo ilustrovali primere za većinu ovih obrazaca. C# i .NET Core su izabrani zbog njihove popularnosti i dizajna, koji podržava rešenja izgradnje, od jednostavnih konzolnih aplikacija, do distribuiranih sistema velikih preduzeća.

U knjizi je opisan veliki broj obrazaca, pa, stoga, ona služi kao odličan uvod za mnoge od njih, dok su detaljno opisani praktični pristupi za izabranu kolekciju. Specifični obrasci koji su opisani izabrani su zato što ilustruju specifičnu tačku ili aspekt obrazaca. Reference za dodatne resurse obezbeđene su da bi omogućile korisniku da detaljno upozna obrasce koji ga interesuju.

Počev od jednostavnih veb sajtova, pa do distribuiranih sistema velikih preduzeća, odgovarajući obrasci mogu da predstavljaju razliku između uspešnog i dugotrajnog rešenja i rešenja koje se smatra neuspešnim zbog loše performanse i visokih troškova. U ovoj knjizi opisani su mnogi obrasci koji mogu da se primene za igradnju rešenja koje rukuje neizbežnim promenama potrebnim za održavanje konkurentnosti u poslovanju, kao i za postizanje robusnosti i pouzdanosti koje se očekuju od modernih aplikacija.

KOME JE NAMENJENA OVA KNJIGA

Ciljna publika su programeri modernih aplikacija koji rade u kolaborativnom okruženju, koje predstavlja veliki broj pozadina i industrija, jer obrasci mogu da budu primenjeni na širok raspon rešenja. Pošto je u ovoj knjizi opisan kod da bismo bolje objasnili obuhvaćene obrasce, trebalo bi da čitaoci poznaju razvoj softvera - ne opisujemo kako da programirate, već kako da bolje programirate. Zbog toga će ciljna publika biti u rasponu od mladih programera do iskusnih programera i do softverskih inženjera i dizajnera. Za neke čitaoce sadržaj će biti nov, a za druge će biti podsetnik

ŠTA OBUHVATA OVA KNJIGA

Poglavlje 1, „Pregled OOP-a u .NET Coreu i C#-u“, sadrži pregled objektno-orijentisanog programiranja (OOP) i način kako se ono primenjuje u C#-u. Ovo poglavlje služi kao podsetnik na važne konstrukcije i funkcije OOP-a i C#-a, uključujući nasleđivanje, enkapsulaciju i polimorfizam.

Poglavlje 2, „Projektni obrasci i principi modernog softvera“, sadrži kataloge i različite obrasce upotrebljene u razvoju modernog softvera. Istražićemo veliki broj obrazaca i kataloga, kao što su SOLID, Gang of Four i obrasci za integrisanje u preduzeću, i opisaćemo „životni ciklus“ razvoja softvera i druge načine za razvoj softvera.

Poglavlje 3, „Implementiranje projektnih obrazaca (osnove - 1. deo)“, sadrži detalje o projektnim obrascima koji su upotrebljeni za izgradnju aplikacija u C#-u. Korišćenjem primera aplikacije predstavićemo razvoj vođen testiranjem, minimum viable product i druge obrasce iz knjige „Gang of Four“.

Poglavlje 4, „Implementiranje projektnih obrazaca (osnove - 2. deo)“, sadrži nastavak detaljnog opisa projektnih obrazaca upotrebljenih za izgradnju aplikacija u C#-u. Predstavićemo koncepte kao što su Dependency Injection i Inversion of Control i nastavićemo istraživanje projektnih obrazaca, uključujući obrasce Singleton i Factory.

Poglavlje 5, „Implementiranje projektnih obrazaca - .NET Core“, se „nadgrađuje“ na poglavlja 3 i 4 istraživanjem obrazaca koje obezbeđuje .NET Core. Nekoliko obrazaca, uključujući Dependency Injection i Factory, biće ponovo upotrebljeno u .NET Core radnom okviru.

Poglavlje 6, „Implementiranje projektnih obrazaca za veb aplikacije - 1. deo“, sadrži nastavak istraživanja .NET Corea pregledom funkcija koje su podržane u razvoju veb aplikacije u izradi primera aplikacije. U ovom poglavlju prikazane su smernice za kreiranje inicijalne veb aplikacije, opisane su važne karakteristike veb aplikacije i predstavljeno je kako se kreiraju CRUD stranice veb sajta.

U Poglavlju 7, „Implementiranje projektnih obrazaca za veb aplikacije - 2. deo“, nastavljamo istraživanja razvoja veb aplikacija pomoću .NET Corea pregledom različitih arhitekturnih obrazaca i rešenja bezbednosnih obrazaca. Takođe su opisane autentifikacija i autorizacija. Testiranje koda je dodato upotrebom Moq mocking radnog okvira.

Poglavlje 8, „Konkurentno programiranje u .NET Coreu“, sadrži detaljno predstavljen razvoj veb aplikacije za opis konkurentnosti u razvoju C# i .NET Core aplikacija. Istražićemo async/await obrazac i višenitni i konkurentni rad. Takođe je opisan parallel LINQ, uključujući odloženo izvršenje i prioritete niti.

U Poglavlju 9, „Funkcionalno programiranje“, istražićemo funkcionalno programiranje u .NET Coreu, što uključuje ilustrovanje funkcija C# jezika koje podržavaju funkcionalno programiranje i primenu ovih funkcija na primer aplikacije, uključujući i primenu Strategy obrasca.

U Poglavlju 10, „Obrasci i tehnike reaktivnog programiranja“, nastavljamo izgradnju .NET Core veb aplikacije istraživanjem obrazaca i tehnika reaktivnog programiranja, koji su upotrebljeni za izgradnju prilagodljivih i skalabilnih veb sajtova. Istražićemo principe reaktivnog programiranja, uključujući obrasce Reactive i IObservable. Opisani su i različiti radni okviri, uključujući popularni .NET Rx Extensions i ilustraciju Model-view-viewmodel (MVVM) obrasca.

U Poglavlju 11, „Napredne tehnike projektovanja i primene baze podataka“, istražićemo obrasce upotrebljene u projektovanju baze podataka. Prikazan je praktičan primer primene Command Query Responsibility Segregation obrasca, uključujući i projektovanje baze podataka ledger-style.

U Poglavlju 12, „Kodiranje za cloud“, opisaćemo razvoj aplikacije, uz primenu rešenja zasnovanih na cloudu, uključujući i ključna pitanja skalabilnosti, dostupnosti, bezbednosti, dizajna aplikacije i DevOpsa. Objašnjen je veliki broj obrazaca upotrebljenih u rešenjima zasnovanim na cloudu, uključujući različite tipove skaliranja i obrasce upotrebljene u arhitekturi vođenoj događajima, bezbednost, keš i telemetriju.

Dodatak A, „Najbolja praksa“, sadrži opis dodatnih obrazaca i najbolje prakse. Jedan odeljak posvećen je upotrebi modelovanja, najboljoj praksi i dodatnim obrascima, kao što su space-based architecture i kontejnerizovane aplikacije.

DA BISTE DOBILI MAKSIMUM IZ OVE KNJIGE:

U ovoj knjizi pretpostavljamo da poznajete OOP i C#. Iako su u njoj opisane napredne teme, ona nije namenjena da bude sveobuhvatan vodič za programiranje. Umesto toga, cilj je da se unaprede veštine programera i dizajnera, obezbeđujući širok raspon obrazaca, prakse i principa. Knjiga obezbeđuje veliki broj alatki za programera moderne aplikacije, od koda niskog nivoa, do arhitekture višeg nivoa, kao i važne obrasce i principe koji se u današnje vreme često koriste.

Ova knjiga obezbeđuje:

- detaljno predstavljanje SOLID principa i najbolje prakse pomoću primera kodiranja upotrebom C#7.x i .NET Corea 2.2
- detaljno razumevanje klasičnih projektnih obrazaca (Gang of four obrazaca)
- principe funkcionalnog programiranja i primere upotrebom C# jezika
- primere iz stvarnog sveta za Architectural obrasce (MVC, MVVM)
- razumevanje native clouda i mikroservisa

Preuzimanje fajlova sa primerima koda

Fajlove sa primerima koda za ovu knjigu možete da preuzmete sa našeg sajta:

http://bit.ly/2lM1any

Kada je fajl preuzet, raspakujte ili ekstrahujte direktorijum, koristeći najnoviju verziju:

- WinRAR/7-Zip za Windows
- Zipeg/iZip/UnRarX za Mac
- 7-Zip/PeaZip za Linux

Code in Action

Da biste videli Code in Action, kliknite na sledeći link:
http://bit.ly/2KUuNgQ

Preuzmite kolorne slike

Takođe smo obezbedili PDF fajl koji sadrži kolorne slike snimaka ekrana/dijagrama upotrebljenih u ovoj knjizi. Možete da ga preuzmete na adresi:
http://bit.ly/2kpFqxo

UPOTREBLJENE KONVENCIJE

Postoji veliki broj konvencija teksta koje su upotrebljene u ovoj knjizi.

`Code InText` - Ukazuje na reči koda u tekstu, nazive tabela baze podataka, nazive direktorijuma, nazive fajlova, ekstenzije fajlova, nazive putanje, skraćene URL-ove, korisnički unos i Twitter postove. Evo i primera: „Tri metoda `CounterA()`, `CounterB()` i `CounterC()` predstavljaju individualni brojač kolekcije karata.“

Blok koda je prikazan na sledeći način:

```
3-counters are serving...
Next person from row
Person A is collecting ticket from Counter A
Person B is collecting ticket from Counter B
Person C is collecting ticket from Counter C
```

Kada želimo da privučemo pažnju na određeni deo bloka koda, relevantne linije ili stavke su ispisane zadebljanim slovima:

```
    public bool UpdateQuantity(string name, int quantity)
    {
        lock (_lock)
        {
            _books[name].Quantity += quantity;
        }

        return true;
    }
```

Unos u komandnoj liniji napisan je na sledeći način:

```
dotnet new sln
```

Zadebljana slova - Ukazuju na novi termin, važnu reč ili reči koje vidite na ekranu. Na primer, reči u menijima ili okvirima za dijalog prikazane su u tekstu zadebljanim slovima. Evo i primera: „U okviru za dijalog Create New Product možete da dodate novi proizvod, a Edit će vam omogućiti da ažurirate postojeći proizvod.“

Napomene ili važna obaveštenja prikazani su ovako.

Saveti i trikovi su prikazani ovako.

POVRATNE INFORMACIJE

Povratne informacije od naših čitalaca su uvek dobrodošle.

Osnovne povratne informacije - Ako imate bilo kakva pitanja o bilo kom aspektu ove knjige, pošaljite nam e-mail na adresu customercare@packtpub.com i u naslov napišite naslov knjige.

Štamparske greške - Iako smo preduzeli sve mere da bismo obezbedili tačnost sadržaja, greške su moguće. Ako pronađete grešku u ovoj knjizi, bili bismo zahvalni ako biste nam to prijavili. Posetite stranicu http://www.packt.com/submit-errata, izaberite knjigu, kliknite na link Errata Submission Form i unesite detalje.

Piraterija - Ako pronađete ilegalnu kopiju naših knjiga u bilo kojoj formi na Internetu, molimo vas da nas o tome obavestite i da nam pošaljete adresu lokacije ili naziv veb sajta. Kontaktirajte sa nama na adresi copyright@packt.com i pošaljite nam link ka sumnjivom materijalu.

Ako ste zainteresovani da postanete autor - Ako postoji tema za koju ste specijalizovani i zainteresovani ste da pišete ili sarađujete na nekoj od knjiga, pogledajte vodič za autore na adresi authors.packtpub.com.

RECENZIJA

Kada pročitate i upotrebite ovu knjigu, zašto ne biste napisali vaše mišljenje na sajtu sa kojeg ste je poručili? Potencijalni čitaoci tada mogu da vide i upotrebe vaše mišljenje da bi se odlučili o kupovini, mi u „Packtu“ možemo da znamo šta mislite o našim proizvodima, a naši autori mogu da vide povratne informacije o svojoj knjizi.

Više informacija o „Packtu“ naći ćete na sajtu packt.com.

Postanite član Kompjuter biblioteke

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

Skenirajte QR kod
registrujte knjigu
i osvojite nagradu

Deo 1

Osnove projektnih obrazaca u C# i .NET Coreu

U ovom delu ćete steći novu perspektivu o projektnim obrascima. Učićete o OOP-u, obrascima, praksi i SOLID principima. Do kraja ovog dela bićete spremni da kreirate sopstvene projektne obrasce.

Ovaj deo ima dva poglavlja:

Poglavlje 1, „Pregled OOP-a u .NET Coreu i C#-u“

Poglavlje 2, „Projektni obrasci i principi modernog softvera“

1

Pregled OOP-a u .NET Coreu i C#-u

Već više od 20 godina najpopularniji programski jezici su zasnovani na principima **objektno-orijentisanog programiranja (OOP)**. Rast popularnosti OOP jezika možemo zahvaliti prednostima mogućnosti apstrahovanja kompleksne logike u strukturu koja se naziva objekat, koji se može lakše objasniti, i, što je još važnije, ponovo upotrebiti unutar aplikacije. U suštini, OOP je pristup projektovanja softvera, odnosno, obrazac za razvoj softvera upotrebom koncepta objekata koji sadrže podatke i funkcionalnost. Kako je softverska industrija sazrevala, obrasci su se pojavljivali u OOP-u za probleme koji obično nastaju, jer su efikasno rešavali te probleme, ali u različitim kontekstima i industrijama. Kako se softver prebacivao sa mainframea na klijentske servere, a zatim u cloud, pojavili su se dodatni obrasci koji pomažu u smanjivanju troškova razvoja i poboljšanju pouzdanosti. U ovoj knjizi ćemo istražiti projektne obrasce, od nastanka OOP-a, do projektnih obrazaca arhitekture za softver zasnovan na oblaku.

OOP je zasnovan na konceptu objekta. Ovaj objekat generalno sadrži podatke, koji se nazivaju svojstva i polja, a kod ili ponašanje se nazivaju metodi.

Projektni obrasci su rešenja za generalne probleme sa kojima se programeri susreću u toku razvoja. Izgradili su ih programeri, rukovodeći se iskustvom onoga što funkcioniše i što ne funkcioniše. Ova rešenja su isprobali i testirali brojni programeri u različitim situacijama. Prednost upotrebe obrasca na osnovu ove prethodne aktivnosti testiranja obezbeđuje da isti zadaci neće biti iznova izvršavani. Osim toga, upotreba obrazaca obezbeđuje da će problem biti rešen bez izazivanja drugih problema.

U ovom poglavlju govorićemo o OOP-u i kako se on primenjuje na C#. Imajte na umu da je poglavlje namenjeno da bude kratak uvod, a ne da bude kompletan primer za OOP ili C#; umesto toga, u poglavlju ćemo opisati aspekte OOP-a i C#-a dovoljno detaljno da bismo predstavili projektne obrasce koji će biti opisani u narednim poglavljima. U ovom poglavlju obradićemo sledeće teme:

- opis OOP-a i kako klase i objekti funkcionišu
- nasleđivanje
- kapsuliranje
- polimorfizam

TEHNIČKI ZAHTEVI

Ovo poglavlje sadrži različite primere koda za objašnjenje ovih koncepata. Kod je jednostavan i služi samo za prikaz. Većina primera uključuje .NET Core konzolnu aplikaciju napisanu u C#-u.

Da biste pokrenuli i izvršili kod, potrebno vam je sledeće:

- Visual Studio 2019 (takođe možete da pokrenete aplikaciju pomoću Visual Studio 2017 verzije 3 ili novije)
- .NET Core
- SQL Server (u ovom poglavlju upotrebljen je Express Edition)

Instaliranje Visual Studioa

Da biste pokrenuli ove primere koda, potrebno je da instalirate Visual Studio (takođe možete da upotrebite omiljeni IDE). Da biste to uradili, pratite sledeće instrukcije:

1. Preuzmite Visual Studio sa linka:

   http://bit.ly/2kQoMab

2. Pratite instrukcije za instalaciju koja je uključena na ovom linku. Dostupno je više verzija Visual Studioa; u ovom poglavlju upotrebićemo Visual Studio za Windows.

Podešavanje .NET Corea

Ako nemate instaliran .NET Core, potrebno je da pratite sledeće instrukcije:

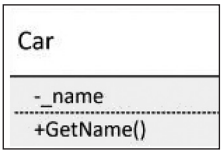
1. Preuzmite .NET Core sa linka:
   http://bit.ly/2koDHs8
2. Pratite instrukcije za instalaciju u povezanoj biblioteci:
   http://bit.ly/2mlOqUQ

Izvorni kod koji je prikazan u ovom poglavlju možda nije kompletan, pa preporučujem da preuzmete izvorni kod sa GitHuba da biste pokrenuli primere
http://bit.ly/2kNI2Fi

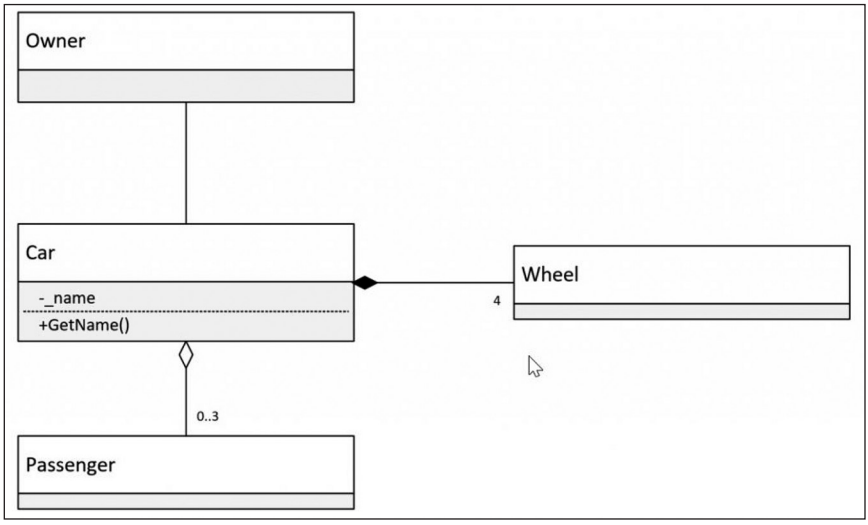
Modeli upotrebljeni u ovoj knjizi

Kao pomoć za učenje, ova knjiga sadrži mnogo primera koda u C#-u, zajedno sa dijagramima i slikama koje pomažu da se opišu specifični koncepti gde je to moguće. Ovo nije **Unified Modeling Language (UML)** knjiga; međutim, za one čitaoce koji poznaju UML mnogi dijagrami bi trebalo da izgledaju poznati. U ovom odeljku ćemo obezbediti opis dijagrama klase koji će biti upotrebljeni u ovoj knjizi.

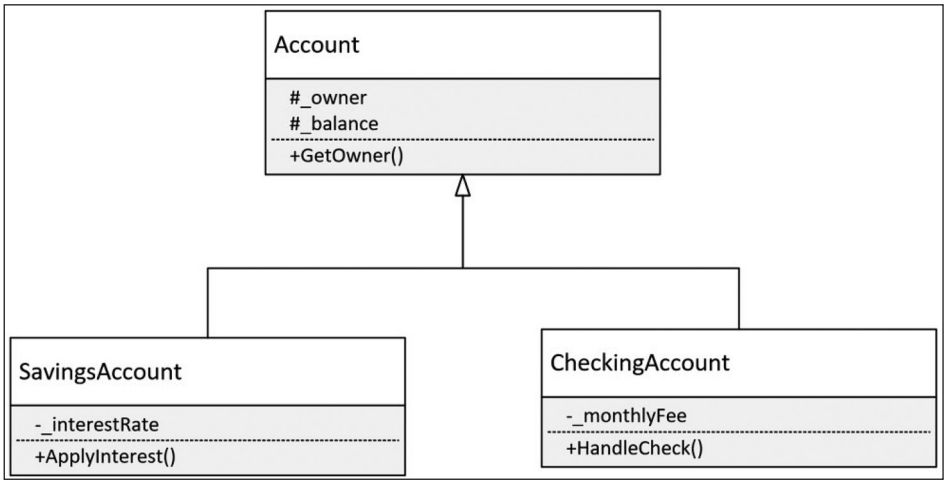
Klasa će biti definisana tako da uključuje i polja i metode razdvojene isprekidanom linijom. Ako je važno za temu, onda će dostupnost biti istaknuta kao - za privatno, + za javno, # za zaštićeno i ~ za interno. Na sledećoj slici je to ilustrovano prikazom klase `Car` sa privatnom `_name` promenljivom i javnim `GetName()` metodom.



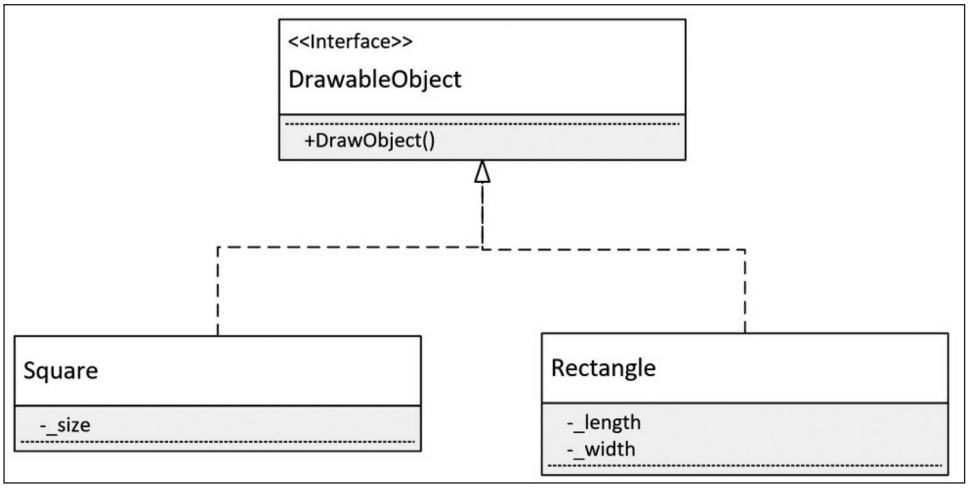
Kada prikazujemo odnose između objekata, povezanost je predstavljena kao puna linija, grupisanje kao prazni romb, a kompozicija kao popunjeni romb. Kada je važno za temu, multiplikativnost će biti prikazana pored konkretne klase. Na sledećem dijagramu ilustrovana je klasa `Car` koja ima jednog vlasnika (**Owner**) i do tri putnika (**Passengers**); sastoji se od četiri točka (**Wheels**).



Nasleđivanje je prikazano pomoću praznog trougla na osnovnoj klasi upotrebom pune linije. Na sledećem dijagramu prikazan je odnos između `Account` osnovne klase i `CheckingAccount` i `SavingsAccount` klasa „potomaka“.



Interfejsi su prikazani na sličan način kao i nasleđivanje, ali se koristi isprekidana linija kao i dodatna oznaka `<<interface>>`, kao što je prikazano na sledećem dijagramu:



U ovom odeljku obezbeđen je pregled modela upotrebljenih u ovoj knjizi. Ovaj stil/pristup je izabran zato što je, nadam se, poznat većini čitalaca.

OOP I KAKO KLASE I OBJEKTI FUNKCIONIŠU

OOP se odnosi na programiranje softvera koji koristi objekte definisane kao klase. Ove definicije uključuju polja, koja se ponekad nazivaju atributi, za skladištenje podataka i metoda za obezbeđivanje funkcionalnosti. Prvi OOP jezik je bio simulacija stvarnih sistema, poznat kao Simula (https://en.wikipedia.org/wiki/Simula), a razvio ga je Norwegian Computing Center 1960. godine. Prvi čist OOP jezik nastao je 10 godina kasnije kao Smalltalk (https://en.wikipedia.org/wiki/Smalltalk) jezik. Ovaj jezik je bio dizajniran da programira Dynabook (http://history-computer.com/ModernComputer/Personal/Dynabook.html), personalni računar koji je kreirao Alan Kay. Iz ovoga je nastalo nekoliko OOP jezika, a najpopularniji su Java, C++, Python i C#.

OOP je zasnovan na objektima koji sadrže podatke. OOP paradigma omogućava programerima da urede/organizuju kod u apstraktnu ili logičku strukturu, koja se zove objekat. Objekat može da sadrži podatke i ponašanje.

Upotrebom OOP pristupa mi izvršavamo sledeće:

- **modularizaciju** - Ovde je aplikacija rastavljena na različite module.
- **ponovnu upotrebu softvera** - Ovde ponovo gradimo ili sklapamo aplikaciju iz različitih (odnosno, postojećih ili novih) modula.

U sledećim odeljcima ćemo detaljnije opisati koncepte OOP-a.

Objašnjenje OOP-a

Ranije su pristupi programiranja imala ograničenja i često su bili teški za održavanje. OOP obezbeđuje novu paradigmu u razvoju softvera koja je imala prednosti nad drugim pristupima. Koncept organizovanja koda u objekte nije teško objasniti, što je velika prednost za usvajanje novog obrasca. Mogu da se upotrebe mnogi primeri iz stvarnog sveta radi objašnjenja koncepta. Složeni sistemi takođe mogu da budu opisani upotrebom manjih gradivnih blokova (odnosno, *objekata*). Objekti omogućavaju programerima da pregledaju odeljke rešenja pojedinačno, dok pojašnjavaju kako se uklapaju u celo rešenje.

Sa ovim na umu, definišimo program na sledeći način: „Program je lista instrukcija koje kompajleru jezika ukazuju šta treba da radi."

Objekat je način organizovanja liste instrukcija na logički način. Instrukcije arhitekte nam pomažu da izgradimo kuću, ali one nisu sama kuća. One su apstraktna reprezentacija kuće. Klasa je slična, jer definiše funkcije objekta. Objekat je kreiran iz definicije klase. To se često naziva **instanciranje objekta**.

Da biste bolje razumeli OOP, treba da pomenemo dva druga značajna programska pristupa:

- **strukturirano programiranje** - Ovaj termin osmislio je Edsger W. Dijkstra 1966. godine. Strukturirano programiranje je paradigma programiranja koja rešava problem rukovanja sa 1.000 linija koda koje deli na manje delove. Ovi mali delovi se najčešće zovu podrutine, strukture bloka, for i while petlje i tako dalje. Jezici koji koriste tehnike strukturiranog programiranja uključuju ALGOL, Pascal, PL/I i tako dalje.
- **proceduralno programiranje** - Ovo je paradigma izvedena iz strukturiranog programiranja i jednostavno je zasnovana na tome kako izvršavamo poziv (poznat i kao proceduralni poziv). Jezici koji koriste tehnike proceduralnog programiranja uključuju COBOL, Pascal i C. Novi primer Go programskog jezika izdat je 2009. godine.

Proceduralni pozivi

Proceduralni poziv je mesto gde je aktivirana kolekcija iskaza, poznata kao procedura. Ponekad se kaže da je procedura pozvana.

Glavni problem navedenih pristupa je što programi nisu lako održivi kada rastu. Programi sa složenijim i većim osnovama koda proširuju ova dva pristupa, što dovodi do teško razumljivih i teško održivih aplikacija. Da bismo prevazišli takve probleme, OOP obezbeđuje sledeće funkcije:

- nasleđivanje
- kapsuliranje
- polimorfizam

U sledećim odeljcima ćemo govoriti detaljnije o ovim funkcijama.

Nasleđivanje, kapsuliranje i polimorfizam se ponekad nazivaju tri stuba OOP-a.

Pre nego što započnemo pregled ovih funkcija, opisaćemo neke strukture koje se nalaze u OOP-u.

Klasa

Klasa je definicija metoda i promenljivih koji opisuju objekat. Drugim rečima, klasa je nacrt koji sadrži definiciju promenljivih i metoda koji su zajednički za sve instance klase koja se naziva objekat.

Pogledajte primer u sledećem kodu:

```
public class PetAnimal
{
    private readonly string PetName;
    private readonly PetColor PetColor;

    public PetAnimal(string petName, PetColor petColor)
    {
        PetName = petName;
        PetColor = petColor;
    }

    public string MyPet() => $"My pet is {PetName} and its color is
{PetColor}.";
}
```

U prethodnom kodu imamo klasu `PetAnimal` koja ima dva privatna polja, pod nazivima `PetName` i `PetColor`, i jedan metod, pod nazivom MyPet`()`.

Objekat

U stvarnom svetu objekti imaju dve karakteristike: stanje i ponašanje. Drugim rečima, možemo reći da svaki objekat ima naziv, boju i drugo; ove karakteristike su jednostavno stanje objekta. Upotrebićemo primer bilo koje vrste ljubimca: pas i mačka će imati svoja imena - moj pas se zove Ace, a moja mačka Clementine. Slično tome, psi i mačke imaju specifično ponašanje, na primer, psu laje, a mačke mjauču.

U odeljku „Objašnjenje OOP-a“ istakli smo da je OOP model programiranja koji treba da kombinuje stanje ili strukturu (podatke) i ponašanje (metod) za isporučivanje funkcionalnosti softvera. U prethodnom primeru različita stanja ljubimaca čine aktuelne podatke, dok je ponašanje ljubimaca metod.

Objekat skladišti informacije (koje su obični podaci) u atributima i otkriva njegovo ponašanje pomoću metoda.

U pogledu OOP jezika, kao što je C#, objekat je instanca klase. U prethodnom primeru stvarni objekat `Dog` je objekat klase `PetAnimal`.

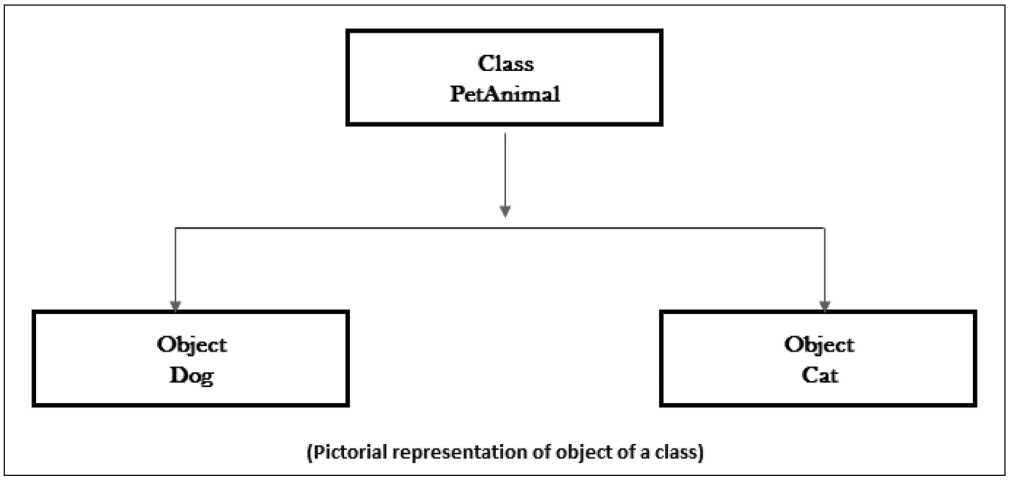
Objekti mogu da budu konkretni (odnosno, objekti iz stvarnog sveta, kao što su pas ili mačka, ili bio koji tip fajla, kao što su fizički fajl ili računarski fajl) ili mogu da budu konceptualni, kao što su šeme baze podatraka ili nacrti koda.

U sledećem isečku koda prikazano je kako možemo da upotrebimo objekat:

```
namespace OOPExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("OOP example");
            PetAnimal dog = new PetAnimal("Ace", PetColor.Black);
            Console.WriteLine(dog.MyPet());
            Console.ReadLine();
            PetAnimal cat = new PetAnimal("Clementine", PetColor.Brown);
            Console.WriteLine(cat.MyPet());
            Console.ReadLine();
        }
    }
}
```

U prethodnom isečku koda kreirali smo dva objekta: `dog` i `cat`. Oni su dve različite instance klase `PetAnimal`. Možemo da vidimo da su poljima ili svojstvima koja sadrže podatke o životinji date vrednosti upotrebom konstruktorskog metoda. Konstruktorski metod je specijalni metod koji se koristi za kreiranje instance klase.

Sada ćemo vizuelizovati ovaj primer na sledećem dijagramu.



Prethodni dijagram je slikovna reprezentacija prethodnog primera koda u kojem smo kreirali dva različita objekta (`Dog` i `Cat`) klase `PetAnimal`. Dijagram je prilično jasan; ukazuje da je objekat klase `Dog` instanca klase `PetAnimal`, kao što je i objekat `Cat`.

Povezivanja

Povezivanja objekta su važna funkcija OOP-a. U stvarnom svetu postoje odnosi između objekata; u OOP-u povezivanja omogućavaju da definišemo odnos *has a* (ima) - na primer, bicikl *has a* vozača ili mačka *has a* nos.

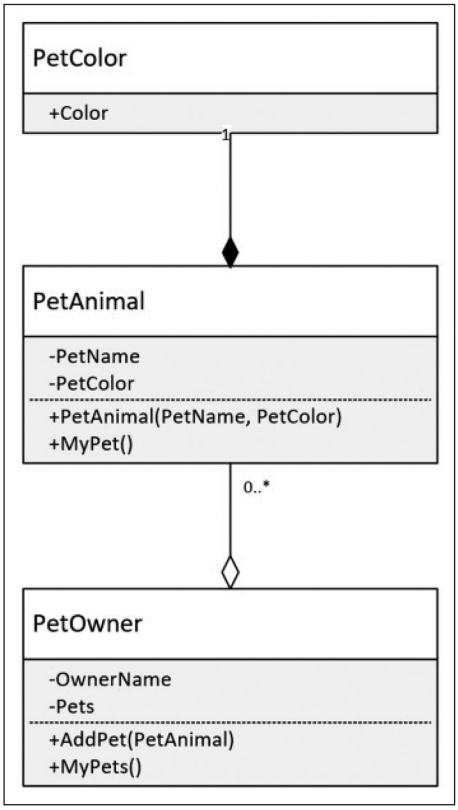
Tipovi has a odnosa su sledeći:

- **povezivanje** - Povezivanje se koristi za opis odnosa između objekata tako da ne postoji opisano vlasništvo, na primer, odnosa između automobila i osobe. Automobil i osoba imaju opisani odnos, kao što je vozač. Osoba može da vozi više automobila, a automobil može da vozi više ljudi.
- **agregacija** - Agregacija je specijalizovana forma povezivanja. Slično povezivanjima, objekti imaju sopstveni „životni ciklus“ u agregaciji, ali uključuju vlasništvo. To znači da objekat „potomak“ ne može da pripada drugom objektu „roditelju“. Agregacija je jednosmerni odnos, u kojem su „životi“ objekata nezavisni jedan od drugog. Na primer, odnos „potomak“ i „roditelj“ je grupa, jer svaki „potomak“ ima „roditelja“, ali nije neophodno da svaki „roditelj“ ima „potomka“.

- **kompozicija** - Kompozicija se odnosi na „odnos smrti“; predstavlja odnos između dva objekta u kojem jedan objekat („potomak“) zavisi od drugog objekta („roditelja“). Ako je objekat „roditelj“ izbrisan, svi njegovi „potomci“ će automatski biti izbrisani. Uzmimo za primer kuću i prostoriju. Jedna kuća ima više prostorija, ali jedna prostorija ne može da pripada u više kuća. Ako demoliramo kuću, prostorije će automatski biti izbrisane.

Ilustrovaćemo ove koncepte u C#-u proširivanjem prethodnog primera kućnih ljubimaca i predstavljanjem klase `PetOwner`. Klasa `PetOwner` može da bude povezana sa jednom ili više instanci `PetAnimal` klase. Pošto klasa `PetAnimal` može da postoji i ako imamo vlasnika i ako ga nemamo, odnos je agregacija. `PetAnimal` klasa je povezana sa klasom `PetColor`; u ovom sistemu klasa `PetColor` postoji samo ako je povezana sa `PetAnimal` klasom, čineći tako kompoziciju.

U sledećem dijagramu ilustrovani su agregacija i kompozicija.



Prethodni model je zasnovan na UML-u i možda vam nije poznat; stoga ćemo istaći neke važne stavke ovog dijagrama. Klasa je predstavljena okvirom koji sadrži naziv klase i njene atribute i metode (razdvojene isprekidanom linijom). Za sada, ignorišite simbol ispred naziva - na primer, + ili -, jer ćemo opisati modifikatore pristupa kasnije kada budemo govorili o kapsuliranju. Povezivanje je prikazano pomoću linije koja povezuje klase. U slučaju kompozicija upotrebljen je popunjeni romb na strani „roditelja“, dok je prazan romb upotrebljen na strani „roditelja“ za prikaz agregacija. Osim toga, vidite da dijagram podržava vrednost multiplikativnosti koja ukazuje na broj mogućih „potomaka“. U dijagramu klasa `PetOwner` može da ima 0 ili više `PetAnimal` klasa (imajte na umu da * ukazuje da ne postoji granica u broju povezivanja).

UML

UML je jezik modelovanja, specifično razvijen za softversko inženjerstvo pre više od 20 godina. Održava ga Object Management Group (OMG). Za više informacija možete da pogledate stranicu: http://www.uml.org/

Interfejs

U C#-u interfejs definiše šta sadrži objekat ili njegov ugovor – konkretno, metode, svojstva, događaje ili indekse objekta. Međutim, ne obezbeđuje implementaciju. Interfejsi ne mogu da sadrže atribute. To je suprotno osnovnoj klasi, jer ona obezbeđuje i ugovor i implementaciju. Klasa koja implementira interfejs mora da implementira sve što je specifikovano u interfejsu.

Apstraktna klasa

Apstraktna klasa je hibrid između interfejsa i osnovne klase, jer obezbeđuje i implementacije, atribute i metode koji moraju da budu definisani u klasama „potomcima“.

Potpis

Termin potpis takođe može da se upotrebi za opis ugovora objekta.

NASLEĐIVANJE

Jedan od najvažnijih koncepata u OOP-u je nasleđivanje. Nasleđivanje između klasa omogućava da se definiše *tip odnosa*; na primer, *automobil je tip* vozila. Ovaj koncept je važan zato što omogućava da objekti istog tipa dele slične funkcije. Recimo da imamo sistem za organizovanje različitih proizvoda online knjižare. Možemo imati jednu klasu za skladištenje informacija o fizičkoj knjizi i drugu klasu za skladištenje informacija o digitalnoj ili online knjizi. Funkcije koje su slične između ove dve klase, kao što su naslov, izdavač i autor, mogu da se sačuvaju u drugoj klasi. I klasa fizičke knjige i klasa digitalne knjige mogu da nasleđuju iz druge klase.

Postoje i drugi termini za opis klasa u nasleđivanju: *„potomak“* ili izvedena klasa *nasleđuje* iz druge klase, dok klasa iz koje se nasleđuje može da se naziva *„roditeljska“* ili *osnovna* klasa.

U sledećim odeljcima ćemo govoriti detaljnije o nasleđivanju.

Tipovi nasleđivanja

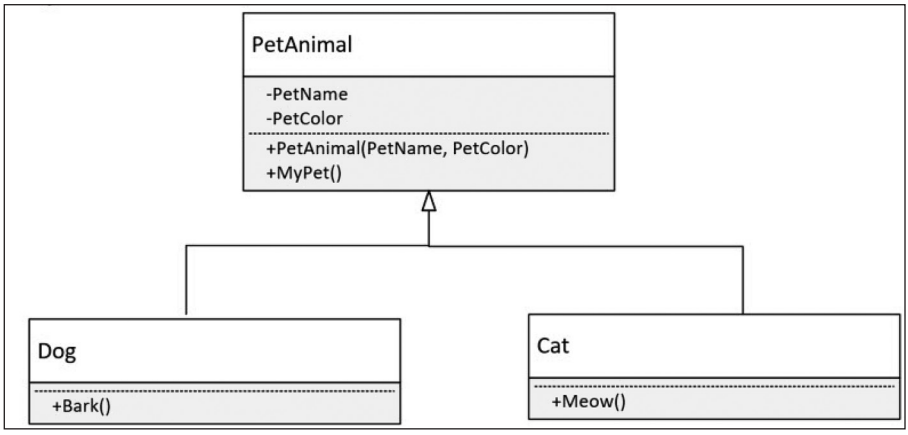
Nasleđivanje nam pomaže da definišemo klasu „potomak“. Ova klasa nasleđuje ponašanje „roditeljske“ ili osnovne klase.

U C#-u nasleđivanje je simbolički definisano pomoću dvotačke (:).

Pogledajte koji su različiti tipovi nasleđivanja:

- **jedno nasleđivanje** - Kao najuobičajeniji tip nasleđivanja, opisuje jednu klasu koja je izvedena iz druge klase.

Pogledajte prethodno pomenutu klasu `PetAnimal` i upotrebite nasleđivanje za definisanje klasa `Dog` i `Cat`. Pomoću nasleđivanja mogu da se definišu neki atributi koji su zajednički za obe klase. Na primer, naziv i boja kućnog ljubimca mogu da budu zajednički, pa će se nalaziti u osnovnoj klasi. Specifičnosti za mačku ili psa mogu da budu definisane u specifičnoj klasi; na primer, glasovi kojima se oglašavaju mačka i pas. Na sledećem dijagramu ilustrovana je `PetAnimal` osnovna klasa sa dve klase „potomka“.



C# podržava samo jedno nasleđivanje.

- **višestruko nasleđivanje** - Višestruko nasleđivanje se dešava kada izvedena klasa nasleđuje više osnovnih klasa. Podržavaju ga jezici kao što je C++. C# ne podržava višestruko nasleđivanje, ali može da se postigne slično ponašanje pomoću interfejsa.

Za više informacija o C#-u i višestrukom nasleđivanju možete da pogledate post:
http://bit.ly/2m0ecxx

- **hijerarhijsko nasleđivanje** - Hijerarhijsko nasleđivanje se dešava kada više klasa nasleđuje iz druge klase.
- **višeslojno nasleđivanje** - Kada je klasa izvedena iz klase koja je već izvedena, to se naziva višeslojno nasleđivanje.
- **hibridno nasleđivanje** - Hibridno nasleđivanje je kombinacija više nasleđivanja.

C# ne podržava hibridno nasleđivanje.

- **implicitno nasleđivanje** - Svi tipovi u .NET Coreu implicitno nasleđuju iz klase System.Object i njenih izvedenih klasa.

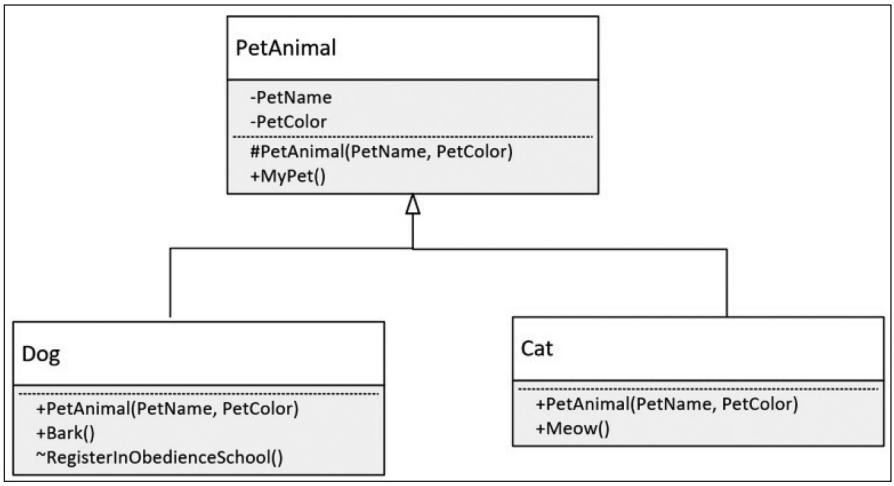
KAPSULIRANJE

Kapsuliranje je još jedan od osnovnih koncepata OOP-a, gde detalji klase, odnosno atributi i metodi, mogu da budu vidljivi ili nevidljivi van objekta. Pomoću kapsuliranja programer obezbeđuje smernice kako klasa treba da se upotrebi i pomaže nam da sprečimo nekorektno rukovanje klasom. Na primer, recimo da želimo da omogućimo dodavanje `PetAnimal` objekata samo pomoću `AddPet`(`PetAnimal`) metoda. To ćemo uraditi tako što ćemo `AddPet` (`PetAnimal`) metod klase `PetOwner` učiniti dostupnim, dok će Pets atribut biti zabranjen za sve van klase `PetAnimal`. U jeziku C# to je moguće uraditi ako Pets atribut učinimo privatnim. To treba da uradimo ako je potrebna dodatna logika kad god je dodata klasa `PetAnimal`, kao što su evidentiranje ili potvrđivanje da klasa `PetOwner` može da ima ljubimca.

C# podržava različite nivoe pristupa koji mogu da budu podešeni za stavku. Stavka može da bude klasa, atribut ili metod klase ili nabrajanje:

- **Public** - Ukazuje da je pristup dostupan van stavke.
- **Private** - Ukazuje da samo objekat ima pristup stavci.
- **Protected** - Ukazuje da samo objekat i objekti klasa koje proširuju klasu mogu da pristupe atributu ili metodu.
- **Internal** - Ukazuje da samo objekti unutar istog sklopa imaju pristup stavci.
- **Protected Internal** - Ukazuje da samo objekat i objekti klase koja proširuje klasu mogu da pristupe atributu ili metodu unutar istog sklopa.

Na sledećem dijagramu primenjeni su modifikatori pristupa za klasu `PetAnimal`.



U ovom primeru ime i boja kućnog ljubimca su privatni atributi da bismo sprečili pristup van klase `PetAnimal`. U konkretnom slučaju zabranjujemo pristup `PetName` i `PetColor` svojstvima, pa samo klasa `PetAnimal` može da im pristupi da bismo obezbedili da samo ona može da promeni njihove vrednosti. Konstruktor funkcija klase `PetAnimal` je zaštićena da bismo obezbedili da samo klasa „potomak“ može da joj pristupi. U ovoj aplikaciji samo klase unutar iste biblioteke u kojoj se nalazi klasa `Dog` imaju pristup metodu `RegisterInObedienceSchool()`.

POLIMORFIZAM

Mogućnost rukovanja različitim objektima upotrebom istog interfejsa naziva se polimorfizam - on obezbeđuje programerima mogućnost da izgrade fleksibilnost u aplikacijama pisanjem jedne funkcionalnosti koja može da bude primenjena na različite obrasce, sve dok dele zajednički interfejs. Postoje različite definicije za polimorfizam u OOP-u, a mi ćemo istaći dva glavna tipa:

- **statičko ili rano povezivanje** - Ovaj oblik polimorfizma se dešava kada je aplikacija prevedena.
- **dinamičko ili kasno povezivanje** - Ovaj oblik polimorfizma se dešava kada je aplikacija pokrenuta.

Statički polimorfizam

Statički polimorfizam, ili rano povezivanje, dešava se u vreme prevođenja i primarno se sastoji od preklapanja metoda, gde klasa ima više metoda istog naziva, ali sa različitim parametrima. To je često korisno za prenos značenja metoda ili za pojednostavljivanje koda. Na primer, u kalkulatoru čitljivije je da imamo više metoda za dodavanje različitih tipova brojeva, nego da imamo različite nazive metoda za svaki scenario; pogledajmo sledeći kod:

```
int Add(int a, int b) => a + b;
float Add(float a, float b) => a + b;
decimal Add(decimal a, decimal b) => a + b;
```

U sledećem kodu je ponovo prikazana ista funkcionalnost, ali bez preklapanja `Add()` metoda:

```
int AddTwoIntegers(int a, int b) => a + b;
float AddTwoFloats(float a, float b) => a + b;
decimal AddTwoDecimals(decimal a, decimal b) => a + b;
```

U primeru kućnih ljubimaca vlasnik će upotrebiti različitu hranu da bi nahranio objekte `cat` i `dog` klase. Možemo da definišemo ovo, jer klasa `PetOwner` ima dva metoda za `Feed()`, kao što je prikazano u sledećem kodu:

```
public void Feed(PetDog dog)
{
    PetFeeder.FeedPet(dog, new Kibble());
}

public void Feed(PetCat cat)
{
    PetFeeder.FeedPet(cat, new Fish());
}
```

Oba metoda koriste `PetFeeder` klasu za hranjenje kućnih ljubimaca, dok je klasi `dog` dat `Kibble`, a `cat` instanci `Fish`. Klasa `PetFeeder` je opisana u odeljku „*Generička klasa*“.

Dinamički polimorfizam

Dinamički polimorfizam, ili kasno povezivanje, dešava se dok je aplikacija pokrenuta. Postoje brojne situacije gde ovo može da se desi i opisaćemo tri uobičajene forme u C#-u: interfejs, nasleđivanje i generička klasa.

Polimorfizam interfejsa

Interfejs definiše potpis koji klasa mora da implementira. U primeru `PetAnimal` zamislićemo da smo hranu za kućne ljubimce definisali prema količini energije koju obezbeđuje na sledeći način:

```
public interface IPetFood
{
    int Energy { get; }
}
```

Sam po sebi, interfejs ne može da bude instanciran, ali opisuje šta instanca `IPetFood` mora da implementira. Na primer, `Kibble` i `Fish` mogu da obezbeđuju različit nivo energije, kao što je prikazano u sledećem kodu:

```
public class Kibble : IPetFood
{
    public int Energy => 7;
}

public class Fish : IPetFood
{
    int IPetFood.Energy => 8;
}
```

U prethodnom isečku koda `Kibble` obezbeđuje manje energije nego `Fish`.

Polimorfizam nasleđivanja

Polimorfizam nasleđivanja omogućava da funkcionalnost bude određena u vreme pokretanja na sličan način kao i interfejs, ali primenjuje se na nasleđivanje klase. U našem primeru kućni ljubimac može da bude nahranjen, pa možemo da definišemo to kao novi metod `Feed` (`IPetFood`), koji koristi prethodno definisani interfejs:

```
public virtual void Feed(IPetFood food)
{
    Eat(food);
}

protected void Eat(IPetFood food)
{
    _hunger -= food.Energy;
}
```

U prethodnom kodu ukazano je da će sve implementacije klase `PetAnimal` imati `Feed(IPetFood)` metod, a klase „potomci“ mogu da obezbede drugačiju implementaciju. Metod `Eat(IPetFood food)` nije označen kao virtuelni, jer svaki `PetAnimal` objekat treba da upotrebi metod bez potrebe da promeni njegovo ponašanje. Osim toga, metod je označen kao zaštićen da bismo sprečili pristup van objekta.

Virtuelni metod ne treba da bude definisan u klasi „potomku“, za razliku od interfejsa, gde svi metodi interfejsa moraju da budu implementirani.

Klasa `PetDog` neće promeniti ponašanje osnovne klase, jer će pas jesti i `Kibble` i `Fish`. Mačka je mnogo mudrija, kao što je prikazano u sledećem kodu:

```
public override void Feed(IPetFood food)
{
    if (food is Fish)
    {
        Eat(food);
    }
    else
    {
        Meow();
    }
}
```

Upotrebom ključne reči override `PetCat` klasa će promeniti ponašanje osnovne klase, što dovodi do toga da mačka jede samo ribu.

Generička klasa

Generička klasa definiše ponašanje koje može da bude primenjeno na klasu. Obično se ovaj oblik polimorfizma koristi u kolekcijama, gde može da se primeni isti pristup rukovanja objektom, bez obzira na tip objekta. Na primer, liste znakovnih nizova ili celih brojeva mogu da se obrade upotrebom iste logike, bez potrebe da se razlikuju specifični tipovi.

Možemo da definišemo generičku klasu za hranjenje kućnih ljubimaca. Ova klasa jednostavno hrani kućne ljubimce nekom hranom, kao što je prikazano u sledećem kodu:

```
public static class PetFeeder
{
    public static void FeedPet<TP, TF>(TP pet, TF food) where TP :
PetAnimal
                                                    where TF : IPetFood
```

```
        {
            pet.Feed(food);
        }
    }
```

Na kraju ćemo istaći nekoliko interesantnih detalja. Pre svega, klasa ne treba da bude instancirana, jer su i klasa i metod označeni kao statički. Generički metod je opisan upotrebom potpisa metoda `FeedPet<TP, TF>`. Ključna reč `where` je upotrebljena za ukazivanje na dodatne zahteve, šta treba da budu `TP` i `TF`. U ovom primeru ključna reč `where` definiše `TP` kao tip klase `PetAnimal`, dok TF mora da implementira `IPetFood` interfejs.

REZIME

U ovom poglavlju opisali smo OOP i njegove tri glavne funkcije: nasleđivanje, kapsuliranje i polimorfizam. Kada se koriste ove funkcije, klase unutar aplikacije mogu da budu apstrahovane da bi obezbedile definicije koje su razumljive i zaštićene od upotrebe na način koji nije konzistentan sa njihovom namenom. To je osnovna razlika između OOP-a i nekih ranijih tipova jezika za razvoj softvera, kao što su strukturalno i proceduralno programiranje. Zahvaljujući mogućnosti apstrakcije funkcionalnosti, povećana je mogućnost ponovne upotrebe i održavanja koda.

U sledećem poglavlju ćemo opisati različite obrasce koji se koriste u razvoju poslovnog softvera. Opisaćemo obrasce programiranja i principe i obrasce razvoja softvera upotrebljene u **„životnom ciklusu“ razvoja softvera (SDLC)**.

PITANJA

Sledeća pitanja će vam omogućiti da proverite svoje znanje koje ste stekli u ovom poglavlju:

1. Na šta se odnose termini kasno i rano povezivanje?
2. Da li C# podržava višestruko nasleđivanje?
3. Koji nivo kapsuliranja u C#-u može da se upotrebi za sprečavanje pristupa u klasu van biblioteke?
4. U čemu je razlika između agregacije i kompozicije?
5. Mogu li interfejsi da sadrže svojstva?
6. Da li psi jedu ribu?