Sadržaj

Uvod **1**

Potrebno predznanje 3
Objektno orijentisana analiza i dizajn 3
UML 4
Principi kvalitetnog dizajna 4
SOLID principi 5
GRASP principi 5
Dizajn šabloni 6

1. Šabloni za kreiranje **7**

Singlton 11
Sinhronizacija 14
Eksterno definisanje objekta 16
ASP.NET inverzija kontrole 17
Kada treba koristiti Singlton? 19
Prototip 20
MemberwiseClone 22
Teški objekti 23
Eksterno konfigurisanje objekata 25
Kada treba koristiti Prototip? 26
Deljeni objekti 27
Vremenske zone 29
Deljene konekcije 30
Registracija stringova 30
Deljeni konteksti baze 31
Kada koristiti deljene objekte? 32
Graditelj 33
StringBuilder 34
IQueryable 35
Generisanje UI komponenti 36
Konfigurisanje web aplikacije 37
Mokovi za testiranje 38
Kada treba koristiti šablon Graditelja? 39
Apstraktna fabrika 39
Pristup različitim bazama podataka 41

Implementacija UI frejmvorka 43
Kada treba koristiti Apstraktnu Fabriku? 46
Metode fabrike 46
Pristup različitim bazama podataka 48
Proksi za serijalizaciju 50
Kada treba koristiti Metode Fabrike? 52

2. Strukturni šabloni 53

Fasada 54
ASP.NET MVC Kontroler 57
Fabrika – Fasada za kreiranje 58
Kada treba koristiti šablon Fasade? 59
Most 60
Pristup različitim bazama podataka 62
Višestruko nasleđivanje 63
Metode za proširenje 65
Kada treba koristiti šablon Mosta? 66
Adapter 67
Logovanje poruka 69
Funkcije omotači 70
Zadaci (Task<T>) 72
Kada treba koristiti Adapter? 73
Omotač 73
Tokovi 76
Aspektno orijentisano programiranje 77
Kada treba koristiti Omotače? 80
Proksi 81
Keširanje podataka 82
Fajl sistem 84
Objektni Model 84
Refleksija 85
Proksi baza podataka 86
Kada treba koristiti Proksi šablon? 87
Kompozitna struktura (Stablo) 88
Fajl sistem 90
Xamarin 90
Sintaksna analiza 92
Kada treba koristiti Kompozitnu strukturu? 94

3. Šabloni ponašanja 95
Domenski model 96
ASP.NET model strana 98
SQL Server Management Objects 99
Kada treba koristiti Domenske Objekte? 100
Monad 101
IEnumerable<T> 102
Maybe<T> 103
Kada treba koristiti Monade? 105
Komanda 105
.Net komande 109
Povratne funkcije 110
Pristup bazama podataka 110
Kada treba koristiti Komande? 111
Posmatrač 112
Događaji 114
Aktivni redovi 115
Delegati 118
Neblokirajuće notifikacije 119
ASP.NET Core akcioni filteri 120
Kada treba koristiti Posmatrače? 121
Lanac odgovornosti 121
Switch/Case naredba 124
ASP.NET Posrednici 125
Dekorateri 129
Reaktivna ekstenzija 129
Kada treba koristiti Lanac odgovornosti? 130
Podsetnik 131
ASP.NET Forms ViewState 132
Diferencijalna stanja 134
Transakcije 135
Kada treba koristiti Podsetnik? 138
Šablonski metod 138
Pristup različitim bazama podataka 142
Delegati 143
Kada treba koristiti Šablonski metod? 145
Strategija 145
Obrada stabala 146
Saga 149
Kada treba koristiti Strategije? 150

Stanje 151
Konekcija 154
Leksički analizator 155
Kada treba koristiti Stanje? 158
Medijator 159
Arhitekture zasnovane na događajima 161
SignalR 162
Kada treba koristiti Medijatore? 163
Iterator 164
Yield naredba 166
Iterator stabla 167
Čitanje rezultata iz baze podataka 167
Kada treba koristiti Iterator? 168
Interpreter 168
ASP.NET Kontrole 170
Roslin 171
SMO 171
Kada treba koristiti Interpreter? 172
Posetilac 172
Uparivanje šablona 177
Procesiranje heterogenih kolekcija 179
Svrha metode Accept 182
Sintaksna stabla 183
Kada treba koristiti Posetioce? 186
Zaključavanje 187
Monitor 189
Semafor 191
Čitači i pisači 192
Barijere 193
Kada treba koristiti zaključavanje? 195

4. Šabloni za pristup podacima 197

Aktivni zapis 200
Kada treba koristiti Aktivne zapise? 201
Objekat za pristup podacima (DAO) 202
Kada treba da se koristi DAO? 206
Maperi 206
Mapiranje metapodataka 209
Mapiranje svojstava i relacija 211
Odloženo očitavanje 212

Kada treba koristiti mapere? 212
Upiti 213
Azure Mobile Apps 214
Entity Framework LINQ upiti 214
Kada treba koristiti upite? 215
Skladište 216
Metode za proširenje 218
Kada treba koristiti skladište? 219
Jedinica posla 220
Entity Framework kontekst 221
Kada treba koristiti jedinicu posla? 222
CQRS 223
Kada treba koristiti CQRS? 225

5. Šabloni za prikaz podataka 227

Raspored elemenata 228
Definisanje elemenata 229
Povezivanje podataka 231
Prezentaciona logika 232
Šablonski prikaz 234
ASP.NET Web Forms ASPX strane 234
ASP.NET MVC Razor strane 235
Kada treba koristiti Šablonski prikaz? 236
Kontroler Prikaza 238
ASP.NET Web Forms 238
Kada treba koristiti Kontroler prikaza? 240
MVP 241
Kada treba koristiti MVP šablon? 246
MVC 247
ASP.NET MVC 249
Kada treba koristiti MVC šablon? 250
MVVM 251
Xamarin 254
Razor model strane 256
Fluks 258
Kada treba koristiti MVVM šablon? 260

Zaključak 263

Indeks pojmova 267

Literatura 271



Uvod

Dizajn šabloni (engl. *Design Patterns*) ili Projektni uzorci su bitni za svakog softverskog inženjera. Oni predstavljaju opise standardnih problema na koje često nailazimo u softverskom dizajnu i preporučene načine na koje se oni mogu rešiti. Može vam se desiti da rešavate neki problem i shvatite da dodajete sve više i više koda, konstantno refaktorišete programski kôd koji ste napisali, a onda vidite primer kojim se to moglo mnogo jednostavnije i elegantnije rešiti uz minimalnu količinu koda. Ovakvi standardni primeri implementacija opisani su kao Dizajn šabloni i oni vam omogućavaju da nađete ideje, primere i trikove kojima će vaš kôd biti jednostavniji razumljiviji i elegantniji.

Refaktorisanje koda je važna operacija održavanja softverskih sistema kojom postojeći programski kôd menjate kako bi postao jednostavniji i čitljiviji. Refaktorisanjem programskog koda ćete u velikom broju slučajeva transformisati vaš programski kôd u neke dizajn šablone. Međutim, kada jednom shvatite dizajn šablone, količina koda koji ćete morati da refaktorišete se može drastično smanjiti. Dizajn šabloni vam pomažu da pišete optimalan kôd čim počnete implementaciju pošto vas uče šta je optimalan način rada i razmišljanja. Na ovaj način ćete uštedeti veliku količinu vremena pošto će vam programski kôd od početka implementacije biti mnogo kvalitetniji i optimalniji. Dizajn šabloni nisu teorija koju treba naučiti samo kako biste dobili na ceni, nego znanje koje će vam pomoći da ne pravite iste greške kao oni koji su pre vas rešavali slične probleme.

Pored toga, oni predstavljaju softverski žargon i način kojim iskusni softverski inženjeri komuniciraju. Kada ovladate dizajn šablonima, često ćete koristiti fraze kao „Mislim da bi to trebalo da se implementira kao **Prototip**…”, bez potrebe da tražite

na internetu ili u nekoj knjizi šta je to i da li je baš taj šablon pravi izbor. Znanje dizan šablona je jedno od merila sposobnosti, zrelosti i kvaliteta ljudi koji se bave softverom, i omogućava vam da efikasno komunicirate sa drugim inženjerima. To je i razlog zašto oni predstavljaju neizostavna pitanja na intervjuima za posao kojima se proverava koliko ste iskusni u softverskom inženjerstvu. Ako možete da odgovorite na pitanja kao što su „Šta je šablon Mosta?", „Kada treba koristiti Lanac odgovornosti?", „Koja je razlika između Šablonskog metoda i Strategije?", sagovorniku ćete uliti pouzdanje da ste iskusan i kvalitetan softverski inženjer.

Zašto Dizajn šabloni na C# jeziku i .NET tehnologiji? Zato što je ova knjiga namenjena softverskim inženjerima koji se bave ovim tehnologijama ili žele da se usavrše u njima. Erih Gama je napisao odličnu knjigu o dizajn šablonima koristeći generičke koncepte i jezičke konstrukcije koji su primenljivi na veliki broj objektno orijentisanih jezika. Međutim, svaki jezik ima svoje specifičnosti. Ako napravite šablone po originalnom GoF receptu, onda verovatno ne koristite koncepte kao što su događaji, delegati, lambda izrazi, niti, LINQ, Task i Action klase i druge specifične C#.NET jezičke konstrukcije koje vam omogućavaju da na efikasniji način dizajnirate aplikacije. Svaki jezik je specifičan i ima nešto što drugi nemaju i bitno je da znate trikove iz pojedinih programskih jezika kako biste bolje dizajnirali softverske aplikacije. Cilj ove knjige je upravo da vam predstavi dizajn šablone kao i neke specifične jezičke konstrukcije u C# jeziku koje će vam pomoći da napravite efektniji dizajn.

Zanimljiva stvar je da mnogi softverski inženjeri koriste dizajn šablone a da toga nisu ni svesni. U .NET bibliotekama i paketima, ili .NET frejmvorku može se naći veliki broj dizajn šablona koje svakodnevno koristimo, a da toga nismo ni svesni. Cilj ove knjige je da vas provede kroz biblioteke i klase koje već dobro znate i na osnovu njih objasni primenu dizajn šablona. Ono po čemu se ova knjiga razlikuje od sličnih knjiga o dizajn šablonima je to što se šabloni neće objašnjavati na osnovu edukativnih problema kao što su geometrijske figure ili fiktivnih aplikacija koje se savršeno uklapaju u neki šablon ali nemaju dodira sa onim što vi konkretno radite. Većina dizajn šablona biće objašnjena na primerima iz .NET frejmvorka, ASP.NET, Entity Framework, Xamarin i sličnih biblioteka koje stalno koristite. U primerima u knjizi ćete videti kako su softverski timovi u Microsoftu odlučili da upotrebe dizajn šablone kada se radi u nekom od ovih frejmvorka, što vam može pomoći da shvatite kako da ih i vi upotrebite u vašim aplikacijama.

Verovatno najpoznatiji šabloni su 22 dizajn šablona koje su opisali Erih Gama i njegov tim (poznatiji kao GoF tim – engl. *Gang of four*) i oni najčešće predstavljaju ono što većina ljudi smatra za dizajn šablone. Međutim postoji veliki broj korisnih dizajn šablona koje je definisao Martin Fauler. U ovoj knjizi će biti objašnjeni šabloni iz obe grupe za koje smatram da su najbitniji za moderni razvoj softverskih aplikacija.

Ova knjiga je podeljena na dva dela.

U prvom delu će biti objašnjena 22 GoF dizajn šablona podeljena u tri kategorije koje je koristio GoF tim, gde ćete se upoznati sa preporučenim šablonima za **kreiranje** objekata, za definisanje **strukture** podataka i implementaciju **ponašanja** objekata. Pored 22 GoF šablona biće objašnjen i Domenski Objekat koji predstavlja jedan bitan vid ponašanja koji je definisao Martin Fauler. Ovo su **taktički dizajn šabloni** koje ćete koristiti i uklapati prilikom implementacije različitih funkcionalnosti.

U drugom delu knjige će biti objašnjeni odabrani **arhitekturni dizajn šabloni** koje je uglavnom opisao Martin Fauler. Za razliku od GoF dizajn šablona koji se mogu primeniti na različite implementacije funkcionalnosti ili slučajeve korišćenja, Faulerovi šabloni objašnjavaju kako se rešavaju problemi pristupa podacima u bazama i fajlovima (npr. *Repository*, *Query*), i kako se podaci prikazuju na korisničkom interfejsu (npr. MVC, MVP, MVVM). Oni se mogu nazvati arhitekturni dizajn šabloni pošto se promenom nekih od ovih šablona u postojećoj aplikaciji, često menja i njena kompletna arhitektura.

Ova knjiga će vam pomoći da shvatite dizajn šablone, vidite u akciji klase gde su ovi šabloni primenjeni kao i da razumete kada treba ili ne treba koristiti određene šablone.

Potrebno predznanje

Ovo nije knjiga za apsolutne početnike. Ako tek učite C#, .NET i objektno orijentisano programiranje, verovatno ćete imati problema da shvatite neke koncepte i primere iz ove knjige. Međutim, to ne znači da ne treba da pročitate neke delove, ili čak da uporedno učite osnove programiranja sa dizajn šablonima. U ovoj knjizi možete da nađete neke zanimljive primere primene koncepata koje učite u drugim knjigama.

S obzirom na to da različiti ljudi različito shvataju čak i osnovne koncepte, u nastavku će biti objašnjeni osnovni koncepti potrebni da se shvati ova knjiga.

Objektno orijentisana analiza i dizajn

Objektno orijentisana analiza i dizajn je metodologija dizajna i projektovanja većine modernih aplikacija. Ovo može da bude veoma opširan skup tehnika dizajna koji se ne primenjuje u potpunosti u svim projektima, ali se osnovni principi skoro uvek primenjuju.

U objektno orijentisanoj analizi i dizajnu se programski kôd piše tako da se koncepti koji se identifikuju u domenu u kome se rešava problem predstavljaju u vidu klasa. Objekti klasa predstavljaju realne objekte iz domena za koji se rešava problem. Interakcije među klasama se predstavljaju metodama klasa koje se međusobno pozivaju dok se poruke i podaci koji se razmenjuju prilikom ovih interakcija predstavljaju parametrima koji se predaju tim metodama prilikom poziva.

UML

UML (engl. *Unified modeling language*) je vizuelni jezik kojim se grafički predstavlja i opisuje objektno orijentisani dizajn. UML je skup pravila i simbola kojim se vizuelno opisuju klase, njihovi podaci, relacije i interakcije sa drugim klasama.

Odlično poznavanje UML dijagrama i iskustvo sa alatima za modelovanje nije preduslov za čitanje ove knjige. Međutim, bilo bi dobro da znate osnove UML jezika pošto će svaki šablon biti objašnjen nekim UML dijagramom.

Principi kvalitetnog dizajna

Pisanje programskog koda u vidu klasa i crtanje UML dijagrama vam neće garantovati da je vaš dizajn dobar (iako odsustvo ovakvog načina rada u mnogim slučajevima može da predstavlja rizik da dizajn neće biti dobar).

Postoje neki principi kojih se treba pridržavati prilikom dizajna i programiranja kako bi se održao kvalitet koda. Jedan od poznatih principa je DRY (engl. *Don't repeat yourself*) Kenta Beka po kome treba izbeći dupliranje istog ili sličnog koda refaktorisanjem duplikata. Klase i UML dijagrami nikog neće sprečiti da napravi duplikate sličnih klasa i još ih i dokumentuje UML jezikom. Zbog toga uvek treba imati na umu da ova pravila postoje i primenjivati ih.

Postoje dve veće grupe principa koje treba primenjivati paralelno dizajnu a to su SOLID i GRASP principi, i oni će biti ukratko opisani u nastavku.

Dizajn šabloni vam u velikom broju slučajeva omogućavaju po automatizmu da primenite ovakve preporuke i principe, ali je bitno da ih razumete kako biste shvatili šta vam dizajn šabloni omogućavaju.

SOLID principi

SOLID je skup dizajn principa koji predstavlja akronim sledećih svojstava (navedenih u originalnom obliku na engleskom):

- **Single responsibility principle** – svaka klasa, metoda ili funkcija mora da radi tačno jednu stvar i da bude odgovorna isključivo za nju. Na ovaj način promena funkcionalnosti za koju je ona odgovorna vrši se samo u toj komponenti i ne zahteva promenu i drugih komponenti.
- **Open/closed principle** – svaka klasa i funkcija mora da bude otvorena za promene ponašanja, što znači da treba da eksterne komponente mogu da konfigurišu kako ona radi, ali s druge strane treba da ima zaokružene i generičke funkcionalnosti koje ne moraju da se menjaju kad god je potrebna neka varijacija funkcionalnosti.
- **Liskov substitution principle** govori da se ekvivalentne komponente (npr. one koje imaju isti interfejs) moraju dizajnirati tako da se jedna može zameniti drugom bez uticaja na ostatak softverskog sistema koji ih koristi.
- **Interface segregation** je princip koji preporučuje da interfejsi treba da imaju minimalni neophodni skup funkcija i da ne uključuju metode koje će možda koristiti nekom pozivaocu, ali u većini slučajeva neće biti korišćene.
- **Dependency Inversion** je princip koji govori da interfejsi i apstrakcije ne treba da zavise od svojih konkretnih implementacija niti da budu svesni njih nego da samo imaju reference na ostale interfejse i apstrakcije.

GRASP principi

Drugi skup principa kojim se definišu odgovornosti i organizacija komponenti u arhitekturi su takozvani GRASP (engl. *General responsibility assignment software patterns*) principi koji govore o dodeljivanju uloga i odgovornosti komponentama, kao o načinima kako komponente treba da budu povezane. Komponente u arhitekturi mogu da imaju različite uloge koje definišu odgovornosti komponenti, kao na primer:

Kontroler je tip komponenti koji je nadležan za interakciju sa korisnikom i korisničkim interfejsom i koji prihvata sistemske događaje koje izazove korisnik pomoću korisničkog interfejsa ili koje inicira eksterni sistem pozivom neke funkcije sistema.

Kreator je tip komponenti koji je zadužen za kreiranje drugih komponenti ili za lociranje komponenti na osnovu imena ili specifikacije.

Informacioni ekspert je tip klase koja sadrži sve potrebno znanje o nekom konceptu (npr. student, prijave) i ka kojoj se uvek ide kada je potrebno naći neke

funkcionalnosti o određenom tipu operacije zato što ona verovatno zna kako će se ta operacija izvršiti.

Postoje dva bitna GRASP principa koja govore kako treba organizovati komponente – visoka kohezija i slabo povezivanje.

Visoka kohezija je princip koji govori da funkcionalnosti, koje su na neki način neodvojive, treba da budu organizovane zajedno u okviru istih klasa, paketa, ili fizičkih komponenti kao što su baze podataka ili .dll fajlovi.

Slabo povezivanje je koncept kojim se preporučuje da se smanji zavisnost među komponentama za koje nije neophodno da znaju jedne za druge kako bi se izbeglo da promena na jednoj komponenti uzrokuje bespotreban lančani niz promena na svim komponentama koje je koriste.

Ovo su generalni principi dobrog dizajna o kojima možete naći više informacija u knjizi „Osnove softverskog inženjerstva".

Dizajn šabloni

Konačno, glavna tema ove knjige su dizajn šabloni tako da je potrebno razumeti čemu oni služe i kako nam mogu pomoći.

U praksi se često mogu naći primeri programskog koda koji postane previše obiman ili se iskomplikuje zato što se rešava neki složeni problem. U velikom broju slučajeva se ovo ne može izbeći zato što složeni problemi ne mogu da se jednostavno reše.

Međutim, često se dešava da uvidite da je složeni kôd ili algoritam mogao da se pojednostavi da se samo promenio način razmišljanja i primenio neki trik kojeg se niko u tom trenutku nije setio. Dizajn šabloni su upravo ovi „trikovi" koji vam omogućavaju da jednostavnije i elegantnije rešite neke specifične probleme koji se često ponavljaju ako ih prepoznate na vreme.

Na primer, programski kôd kojim se kreira konekcija ka bazi podataka se može staviti u svaku metodu kojom se pristupa bazi podataka. Međutim, na ovaj način se duplira isti programski kôd čime se narušava DRY princip. Ako bi se ovaj kôd inicijalno stavio u posebnu CreateConnection metodu čime bi se primenio dizajn šablon Metode Fabrike (engl. *Factory Method*), DRY princip uopšte ne bi bio narušen.

Poznavanje ovakvih šablona vam često može pomoći da napišete programski kôd koji neće zahtevati refaktorisanje.



1

Šabloni za kreiranje

Prva stvar na koju morate da mislite kada počnete da dizajnirate objektno orijentisani programski kôd je kako da napravite objekte koji su vam potrebni. Svi objektno orijentisani jezici (uključujući i C#) obezbeđuju konstruktore kojima se prave novi objekti klasa:

```
var konekcija = new SqlConnection();
```

Ovo je verovatno najčešći način koji ćete da koristite kako biste kreirali objekte, međutim to ne znači i da je uvek najbolji.

Ako koristite konstruktore, onda morate da razmislite kako će se kreirani objekti uništiti kada više ne budu bili potrebni. U slučaju da se ne unište, oni uzalud troše memoriju iako se ne koriste. .NET okruženje obezbeđuje tzv. skupljača smeća (engl. *GC – Garbage collector*) - proces koji u pozadini ispituje objekte koji se više ne koriste i uništava ih. Iako ovo deluje kao veoma dobra stvar, GC ne garantuje kojom brzinom će očistiti objekte ni koliko efikasno će ih pronaći. U nekim slučajevima se može desiti da GC ne stigne da dovoljno brzo očisti objekte tako da će vaša aplikacija trošiti veliku količinu memorije. Zbog toga je dobro eksplicitno uništiti objekat korišćenjem destruktora, ali samo u slučaju da ste sigurni kada se objekat može uništiti.

Prerano uništavanje objekata koje možda neko još uvek koristi jedan je od najčešćih uzroka problema u jezicima kao što je C++.

Problem postaje još veći kada objekat sadrži u sebi neke resurse (npr. Konekcije ka bazi podataka ili fajlovima koje treba zatvoriti) koje treba eksplicitno uništiti što je moguće ranije kako bi drugi objekti mogli da ih koriste.

Korisna konstrukcija u C# jeziku je tzv. `using` blok, prikazan u sledećem primeru:

```
using ( var konekcija = new SqlConnection() ) {

    // objekat konekcija postoji samo u ovom bloku

}
```

U ovom bloku se objekat kreira na početku bloka, može se koristiti u njemu gde je i vidljiv, a kada se izađe iz bloka, objekat se automatski uništava. Ovakav blok u velikom broju slučajeva garantuje da će svi resursi biti vraćeni.

Drugi problem koji je potrebno rešiti je kako kontrolisati tzv. životni ciklus objekata kojim se kontroliše koliko dugo objekti postoje. Na osnovu trajanja životnog ciklusa, objekti se mogu podeliti na sledeće kategorije:

- **Tranzijentni objekti** su objekti koji se brzo uništavaju i kreiraju. Po potrebi oni se mogu i ponovo iskoristiti ako nemaju stanje koje ne sme biti preneto između više poziva (engl. *Stateless objects*).
- **Objekti u okviru sesije** (engl. *Session scoped*) imaju određeni kontekst u kome žive i sve klase i objekti unutar tog konteksta koriste taj objekat. Na primer, u ASP.NET web aplikacijama, kontekst može da bude sesija koja objedinjuje više nezavisnih zahteva koji dolaze od istog klijenta.
- **Statički objekti** su objekti koji žive dok živi aplikacija i kad god nekom treba referenca na objekat dobijaju referencu na isti objekat.

Običnim konstruktorima ili `using` blokovima bi bilo teško kontrolisati ove životne cikluse objekata.

Pre nego što se bolje upoznamo sa načinima na koje se kreiraju objekti, potrebno je prvo videti kako klase kreiraju druge objekte. Jedan od čestih načina kojim se zamenjuju obični konstruktori su posebne klase sa metodama (npr. `GetXXX()` ili `CreateXXX()`) koje kreiraju objekte.

```
public static class ServiceLocator
{
    public static ILogger GetLogger() { ... }
```

```
    public static IConnection GetConnection() { ... }
}
```

Ovakve klase se nekada nazivaju Lokatori servisa ili Fabrike (mada nemaju veze sa šablonima Apstraktne Fabrike i Fabričke Metode koji će biti objašnjeni u ovom poglavlju) i u sebi sadrže logiku kreiranja objekata. Eksterni programski kôd vidi samo metode koje treba pozvati i eventualno prosleđuje neke parametre, kao što je prikazano u sledećem primeru:

```
public class HomeController : Controller
{
    private readonly ILogger logger;
    private readonly IConnection connection,

    public HomeController()
    {
        this.logger = ServiceLocator.GetLogger();
        this.connection = ServiceLocator.GetConnection();
    }
}
```

Prednost ovakvog pristupa leži u činjenici da se centralizuje programski kôd koji kreira objekte gde metode koje kreiraju objekte (servise) skrivaju od pozivaoca implementaciju i kompleksnost algoritama kreiranja objekata koje pozivalac ionako ne mora da zna. U primeru, metoda `GetConnection` kreira konekciju ka bazi podataka i može da čita ime servera, korisničko ime i lozinku iz fajla ili neke druge lokacije. Ako se promeni način implementacije, kao na primer ako konekcija više ne koristi korisničko ime i lozinku radi autentifikacije ili je promenila tip baze podataka, promena će se uraditi samo na jednom mestu i svi pozivaoci neće biti svesni promene niti će postojati potreba da se menjaju zato što je promenjena konekcija. U ovom slučaju šabloni za kreiranje mogu da pomognu da se dizajnira programski kôd unutar ovih metoda. Mana ovakvog pristupa je činjenica da se može pojaviti veliki broj metoda u sistemima sa velikim brojem servisa.

Kao alternativa, može se napraviti klasa sa jednom generičkom metodom kojoj se prosleđuje tip objekta koji treba kreirati:

```
public static class ServiceLocator
{
    public static T Get<T>() { ... }
}
```

U ovom slučaju bi pozivalac samo prosledio interfejs koji mu je potreban i dobio kreirani objekat, kao na primer:

```
this.logger = ServiceLocator.Get<ILogger>();
this.connection = ServiceLocator.Get<IConnection>();
```

Ovaj način zahteva manje programskog koda ali metoda `Get<T>` može da sadrži kompleksnu logiku određivanja tipa na osnovu interfejsa.

Drugi često korišćeni način kreiranja objekata je tzv. **inverzija kontrole** (engl. *IoC – Inversion of control*) u kojoj se ne očekuje od klijenta da eksplicitno pozove neku klasu koja bi mu obezbedila potrebne objekte. U ovoj tehnici postoji jedna komponenta koja se naziva **IoC kontejner** (engl. *IoC container*) i koja predstavlja registar svih objekata, klasa i servisa, koji će se dostaviti drugim klasama, dok klase kojima trebaju objekti samo deklarišu neophodne interfejse u svojim konstruktorima i postave ih lokalno. U sledećem primeru je prikazano kako se ova tehnika koristi u ASP.NET tehnologiji:

```
public class HomeController : Controller
{
    private readonly ILogger logger;
    private readonly IConnection connection,

    public HomeController(IConnection connection, ILogger logger)
    {
        this.connection = connection;
        this.logger = logger;
    }
}
```

IoC Kontejner je eksterna komponenta koja kontroliše kako se prave kontroleri i na osnovu analize argumenata zna kako da kreira objekte koji implementiraju tražene interfejse i da ih prosledi kao argumente konstruktorima. Kao što bi šabloni kreiranja pomogli da se u slučaju `ServiceLocator` klase dizajnira programski kôd unutar metoda koje kreiraju objekte, u ovom slučaju šabloni za kreiranje nam omogućavaju da dizajniramo kako će IoC kontejner praviti objekte koji će biti prosleđeni konstruktorima.

Ono što je zajedničko za oba načina kreiranja objekata je potreba da se definišu neke standardizovane tehnike kojima će lokatori servisa ili kontejneri inverzije kontrole praviti objekte pre nego što proslede reference na njih.

Kada se sagledaju ovi problemi, može se naslutiti da je potrebno definisati neke recepte koji će se koristiti prilikom kreiranja objekata i ovo su upravo dizajn šabloni za kreiranje objekata. U ovom delu knjige biće opisani sledeći šabloni koji opisuju kako se objekti mogu kreirati:

- **Singlton** (engl. *Singleton*) ili Unikat koji obezbeđuje da u aplikaciji postoji samo jedna instanca klase.
- **Prototip** (engl. *Prototype*) kojim se može napraviti jedan objekat na osnovu koga će se ostali kreirati.
- **Deljeni objekti** (engl. *Pooled object*) koji predstavljaju skup predefinisanih objekata koje ostale klase u sistemu mogu da koriste umesto da stalno kreiraju nove objekte.
- **Graditelj** (engl. *Builder*) koji omogućava pozivaocu da postepeno napravi objekat dodajući mu svojstva.
- **Apstraktna fabrika** (engl. *Abstract Factory*) koja omogućava da se u sistemu kreiraju samo objekti iz disjunktnih grupa koji ne mogu da se međusobno mešaju.
- **Metoda fabrike** (engl. *Factory Method*) koja predstavlja metodu koja će se pozvati kako bi se kreirali objekti potrebni u metodama klase.

Singlton

U mnogim scenarijima moramo da obezbedimo da postoji samo jedan objekat u sistemu. Najbolji primer je referentni časovnik koji jedini ima informaciju o tačnom vremenu. Iako svako ima časovnik na svom telefonu ili ruci, mora da postoji **jedan tačan časovnik** sa kojim će se ostali sinhronizovati. Drugi primer može da bude objekat preko koga se kopiraju tekstovi i drugi objekti (engl. *Clipboard*). Ovaj objekat u operativnom sistemu može da pozove bilo koja aplikacija ili klasa kako bi kopirala neki tekst (engl. *Copy*) ili ga izvukla kako bi se kopirao na drugo mesto (engl. *Paste*). Očigledno klase koje pozivaju ovaj objekat ne mogu da uvek prave novi i potrebno je osigurati da svaka komponenta referencira samo jedan objekat u sistemu preko koga se kopira sadržaj.

Ovde se ne radi samo o uštedi memorije, nego problem koji se rešava zahteva da postoji samo jedan objekat u sistemu.

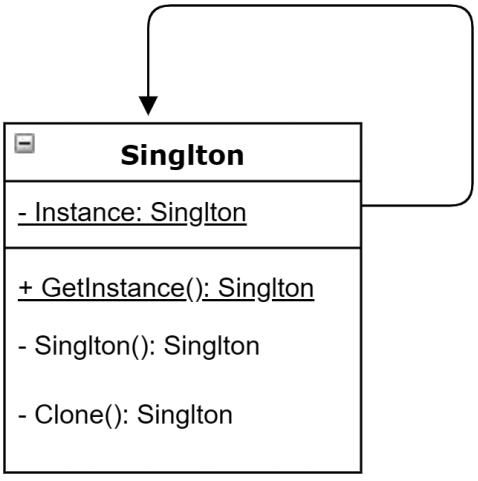
U mobilnim aplikacijama često postoji jedinstvena glavna nit kontrole koja može da izvršava kôd koji prikazuje interfejs korisniku i koja obrađuje događaje. Jedinstvena nit kontrole olakšava sinhronizaciju i obezbeđuje da se lakše obrade događaji koji se dese u aplikaciji, pogotovu ako ne postoji potreba da se paralelno izvršava

neko procesiranje podataka na klijentu. Xamarin frejmvork ima glavnu nit (engl. *MainThread*) koja mora da bude jedinstveni objekat. Ostali primeri kada bi mogao da se koristi ovaj šablon su:

- Keširanje – može da postoji samo jedan keš gde se čuvaju kopije podataka koji se sporo kreiraju ili dohvataju sa udaljenih lokacija (npr. baza podataka ili veb servisa).
- Aplikacioni parametri i konfiguracije – može da postoji samo jedna klasa kojoj se svi obraćaju da pročitaju neko podešavanje i putem nje se menjaju konfiguracioni parametri za sve ostale komponente.
- Globalno stanje – klase koje upravljaju globalnim ili statičkim stanjem u aplikaciji npr. sadrže liste servisa (`ServiceLocator` u prethodnoj sekciji), skup konekcija ka bazi mora da bude jedinstven u sistemu kako se različite eksterne komponente ne bi obraćale različitim skupovima globalnih promenljivih.

Singlton (ili Unikat) je GoF dizajn šablon koji se koristi kako bi se obezbedilo da u sistemu postoji samo jedan objekat određene klase.

UML dijagram koji opisuje strukturu ovog šablona prikazan je na sledećoj slici:



Slika 1.1. Singlton šablon.

Šablon je veoma jednostavan – referenca na jedini objekat ove klase postoji kao statičko polje klase kome se može pristupiti nekom metodom (npr. `GetInstance`). Pravljenje novih objekata je onemogućeno privatnim konstruktorom i privatnom metodom za kloniranje koje ne mogu da se koriste van klase.

Primer implementacije jedinstvenog objekta preko koga se kopiraju podaci pomoću šablona Singltona prikazan je u sledećem kodu:

```
public class Clipboard {

    private static Clipboard instance = null;
    // ili new Clipboard();
    public static Clipboard GetClipboard() {
        if (instance == null) {
            instance = new Clipboard();
        }
        return instance;
    }

    private Clipboard () {}
    ~Clipboard() { throw new NotImplementedException(); }
    protected new object MemberwiseClone() {
    throw new NotImplementedException(); }

    private object content;

    public void Copy(object content) => this.content = content;

    public object Paste() => this.content;

}
```

Ovo je klasa u koju se može upisati sadržaj (polje `content`) koji se kopira pomoću metode `Copy()` i koji se može pročitati i vratiti pozivaocu metodom `Paste()`. Postoji jedno statičko polje koje predstavlja referencu na jedini objekat u aplikaciji i metoda pomoću koje eksterne klase mogu da dobiju referencu na taj jedini objekat. Metoda kojom se vraća referenca na objekat u ovom slučaju proverava da li postoji objekat i ako ne postoji, ona ga kreira. Kada je jednom kreiran i referenca na njega postavljena u statičko polje, u svakom narednom pozivu se vraća referenca na taj jedini objekat.

Potrebno je napraviti još neke izmene kojima se garantuje da neće biti drugih objekata, kao na primer:

- Potrebno je blokirati sve konstruktore klase kako bi se obezbedilo da neko slučajno ne napravi objekat tako što izbegne `GetClipboard()` metodu.
- Potrebno je blokirati pristup metodama koje mogu da kopiraju jedini objekat i naprave duplikate.
- Potrebno je blokirati destruktore i eventualno `Dispose()` metodu kako bi sprečili eksterne klase da unište objekat.

Ovi preduslovi su karakteristični za C# jezik i neophodno ih je implementirati prilikom dizajna ovog šablona. Ovo je jedan od razloga zašto je potrebno učiti dizajn šablone na jeziku koji svakodnevno koristite pošto specifičnosti jezika kojih niste svesni ako primenjujete originalni šablon mogu da unesu probleme u implementaciji.

Klasa koja predstavlja jedinstveni objekat ima nekoliko strategija kojima se može napraviti ovaj objekat:

- **Rana inicijalizacija** je pristup kojim se statičko polje inicijalizuje konstruktorom objekta umesto vrednošću `null` (taj deo je pod komentarom u gornjem primeru).
- **Odložena inicijalizacija** gde statičko polje nije inicijalizovano, a metoda kojom se vraća referenca na taj objekat prvo proverava da li je inicijalizovan i ako nije, inicijalizuje se kao u primeru gore. Ovakva implementacija može da ubrza pokretanje aplikacija posebno u slučajevima kada bi se inicijalizovao veliki broj statičkih objekata, međutim ovakva implementacija može da ima problema sa sinhronizacijom.

Pod inicijalizacijom se ne podrazumeva samo poziv konstruktora kojim se kreira jedinstveni objekat. Često inicijalizacija uključuje i postavljanje nekih polja i svojstava koja inicijalno treba da ima objekat. U slučaju rane inicijalizacije može da se pozove neka posebna statička metoda u kojoj će se napraviti objekat ili se taj kôd može staviti unutar `if` bloka u odloženoj inicijalizaciji.

Sinhronizacija

Singlton je deljen među nitima koje rade sa objektima tako da često moraju da se primene neke tehnike sinhronizacije i zaključavanja[1]. Kôd za vraćanje objekata u prethodnim primerima proverava da li je jedini objekat napravljen, i ako jeste vraća ga pozivaocu, a ako nije, pravi novu instancu koja će biti vraćena i korišćena u novim pozivima. Problem sa ovakvim kodom leži u činjenici da se ipak u nekim slučajevima može desiti da nastanu dve ili više instanci objekta.

```
if (instance == null) {
  /* šta će se desiti ako se ovde procesor preda drugoj niti? */
            instance = new Clipboard();
}
```

[1] Tehnike zaključavanja će biti objašnjene u narednom poglavlju o šablonima za ponašanje.

Zamislimo da dve različite niti konkurentno pozivaju ovu metodu uz pretpostavku da objekat još uvek nije napravljen. Prva nit izvršava uslov `if (instance == null)`, zaključuje da treba da napravi novi objekat i prelazi na liniju koda u kojoj bi trebalo da ga napravi. Međutim, u tom trenutku gubi procesor i počinje da se izvršava druga nit koja isto prolazi kroz ovaj deo koda, proverava isti uslov i pravi novi časovnik pošto i ona vidi da instanca nije inicijalizovana. Posle nekog vremena prva nit ponovo dobija procesor, ali pošto je ona već prošla uslov provere i ne zna da je u međuvremenu neko već napravio časovnik i ona pravi novu instancu. U ovom stanju će se napraviti i koristiti dva „jedinstvena" objekta usled neočekivanog redosleda izvršavanja koda (engl. *Race conditions*).

> *Problemi koji nastaju usled preticanja i sinhronizacije se retko dešavaju, a to ih čini izuzetno teškim za otkrivanje. Zamislite da naiđete na problem u kome Copy/Paste funkcionalnost ne radi zato što su usled preticanja u toku kontrole napravljena dva Clipboard objekta, a onda Copy() ubacuje sadržaj u jedan, a Paste() čita iz drugog koji je prazan. Koliko bi vam bilo teško da pronađete ovaj problem i da uopšte pomislite da možda vaš jedinstveni objekat nije baš uvek jedinstven?*

Rešenje za ovaj problem je jednostavno – sve što treba uraditi je da se spreči nit koja dobije kontrolu da napravi novi objekat. U C# jeziku se može koristiti `lock` blok u ove svrhe kao što je prikazano u sledećem primeru:

```
static object obj = new object();

...

lock(obj) {
  if (instance == null) {
            instance = new Clipboard();
 }
}
```

Naredba `lock` će „zaključati" sekciju naredbi koju okružuje dok bar jedna nit izvršava naredbe u njoj. U slučaju da trenutna nit koja je izvršila proveru izgubi procesor, a druga pokuša da izvrši ovu proveru, druga nit će biti blokirana dok prva ponovo ne dobije kontrolu i ne izađe iz sekcije koju okružuje `lock` naredba. Ovo je još jedan primer specifičnosti C# jezika koja vam može pomoći da ispravno implementirate dizajn šablon.

Eksterno definisanje objekta

Klasični Singlton šablon je klasa koja pravi jedan objekat pozivajući neki od svojih konstruktora. Ovo može da bude nepraktično u nekim slučajevima, kao na primer:

- Nekada ova klasa ne može da zna kako da napravi objekat. Parametri koje treba predati konstruktoru mogu da se nalaze u drugim klasama ili moraju da se učitaju iz eksternih fajlova ili baza podataka. Dodavanje koda koji vrši ovo učitavanje moglo bi da poveća složenost ove klase.
- Nekada je potrebno koristiti drugu izvedenu klasu za Singlton objekat. U primeru sa `Clipboard` objektom se ubacuje (engl. *Copy*) sadržaj koji će biti u memoriji i odatle se izvlači (engl. *Paste*). Sledeća akcija ubacivanja će prepisati postojeći sadržaj novim. Drugačija implementacija bi mogla da snimi sadržaj u neki fajl tako da bude na raspolaganju čak i ako se računar ugasi. Neka druga implementacija bi mogla da čuva više sadržaja i da pamti listu vrednosti koje se mogu izvući. Ovakve funkcionalnosti bi se implementirale u klasama izvedenim iz klase `Clipboard`, ali bazna klasa ne zna za klase izvedene uz nje.
- U slučaju da je potrebno testirati objekte koji referenciraju Singlton objekat teško je zameniti Singlton njegovom simulacijom (engl. *Mock*). U ovom slučaju je potrebno obezbediti da klasa koja se testira dobije simulaciju Singltona kada se pozove `Singleton.GetInstance` metoda.

U ovim slučajevima je potrebno obezbediti da eksterni kôd može da inicijalizuje Singlton objekat.

Jedna mogućnost je da se implementira posebna metoda koja će postaviti instancu npr. `SetInstance(Clipboard)`. U ovom slučaju se ne može koristiti klasični privatni konstruktor pošto eksterne klase moraju da imaju mehanizam kojim bi napravile taj prvi objekat, ali treba obezbediti da se ne može napraviti drugi kada se jednom postavi jedini objekt.

Druga mogućnost je da se Singlton klasi prosledi funkcija koja će napraviti jedinstveni objekat. Pretpostavimo da je ovakva funkcija definisana kao statičko polje klase pod nazivom `CreateInstance`:

```
public class Clipboard {

    private static Clipboard instance = null;
    // ili new Clipboard();
```

```
    public static Function<Clipboard> CreateInstance =
                                              () => new Clipboard();
    public static Clipboard GetClipboard() {
        if (instance == null) {
            instance = CreateInstance();
        }
        return instance;
    }
}
```

U slučaju da se ništa ne promeni u klasi, kada se prvi put zatraži instanca, biće pozvana podrazumevana metoda koja pravi objekat klase `Clipboard`. Međutim, u slučaju da je potrebno da jedinstveni objekat bude instanca neke druge klase (npr. `FileClipboard` koja snima sadržaj i u fajl), eksterno se može postaviti metoda za inicijalno kreiranje objekata[2]:

```
Clipboard.CreateInstance = () => new FileClipboard();
```

Na ovaj način se ne menja originalno ponašanje, ali se daje mogućnost da eksterni kôd postavi specijalnu klasu po potrebi. Ovo je posebno bitno u slučaju da se testiraju klase koje koriste `Clipboard` zato što test kôd može postaviti simulaciju ove klase pre nego što počnu testovi.

Ovakav dizajn unosi potencijalni rizik u konkurentnom izvršavanju koda pošto neka klasa može da zatraži instancu pre nego što druga nit postavi klasu objekta koji treba da bude vraćen. U tom slučaju treba promeniti kôd delegata `CreateInstance` da ili inicijalno baca izuzetak ako nije postavljena nova funkcija, ili da se prilikom postavljanja funkcije baci izuzetak ako Singlton objekat već postoji i neko ga koristi. Zbog jednostavnosti, ove varijacije nisu prikazane u primeru.

ASP.NET inverzija kontrole

U ASP.NET tehnologiji se objekti koji će se koristiti inicijalizuju prilikom pokretanja web aplikacije unutar posebne metode:

[2] U ovom primeru se koristi sintaksa Lambda izraza za definiciju funkcija. `()` `=>` `new` `Clipboard()` je u stvari funkcija koja nema parametre, a u telu kreira objekat tipa `Clipboard` i vraća ga kao rezultat.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(
                                        new Operation(Guid.Empty));

}
```

Parametar `services` predstavlja registar servisa koji će distribuirati potrebne objekte klasama kojima trebaju. Ovo nije klasičan Singlton šablon pošto same klase ne obezbeđuju da postoji jedan objekat, nego neki IoC kontejner registruje objekte ovih klasa i garantuje da će se isti objekat vratiti kad god ga neka klasa zatraži na osnovu tipa interfejsa. Jedna moguća implementacija registrovanja Singlton objekata prikazana je u sledećem primeru.

```
public class ServiceCollection<I, T>: IServiceCollection
                                      where T : new {

    private Map<string, T> services = Map<string, T>;

    public AddSingleton<I, T> () {
        services.Add(nameof(I), new T());
    }

    public AddSingleton<I, T> (T service) {
        services.Add(nameof(I), service);
    }

    public Get<I> () {
        services.Get(nameof(I));
    }
}
```

Ovo nisu objekti napravljeni po GoF šablonu pošto se samo jedine instance objekata stavljaju u statičku mapu. S jedne strane efekat je isti i može postojati samo jedan objekat klase bez potrebe da se zna i implementira šablon, međutim ovo nije ekvivalentno pravom GoF šablonu i vi morate da se postarate da neko slučajno ne klonira objekat ili da ga kreira direktno pošto eksterni kôd ima pristup klasama koje su prosleđene kao Singlton objekti. Trenutno ASP.NET dokumentacija preporučuje da se ne kombinuju Singlton objekti napravljeni klasičnim GoF šablonom i Singlton objekti registrovani u servisima.

Kada treba koristiti Singlton?

Singlton je šablon koji bi trebalo koristiti u slučaju da u sistemu postoji jedinstveno statičko stanje ili skup podataka koji se ne mogu predstaviti statičkim klasama i kada je potrebno obezbediti da se ne može kreirati drugi objekat klase. Singlton predstavlja jedinstveno globalno stanje i kao takav je ekvivalentan statičkim klasama. Prednost Singltona u odnosu na statičke klase leži u činjenici da se ovakvi objekti mogu lakše konvertovati u druge dizajn šablone ili zameniti ekvivalentnim objektima nadklase ili izvedenih klasa bez promena ostatka koda. Ako programski kôd ne zadovoljava ove uslove, primena šablona može doneti više problema nego koristi.

Singlton je jedan od šablona koje je najlakše shvatiti i zbog toga se veoma često koristi čak i u scenarijima kada nije potreban. Zbog toga se može naći veliki broj primera u kojima se Singlton naziva antišablonom pošto donosi više problema nego koristi. Međutim problem nije u samom dizajnu pošto ovaj šablon daje odlično rešenje za scenario za koji je napravljen. Anti-patern koji stvara negativnu sliku o ovom šablonu je zasnovan na činjenici da se nekada koristi u scenarijima za koje nije predviđen samo zato što ga je lako implementirati.

Klase koje imaju samo metode (bez polja) nekada se dizajniraju kao Singlton objekti kako bi se uštedela memorija ili izbeglo nepotrebno sakupljanje objekata koji se više ne koriste. Iako je ovo korisno, ovakav dizajn uvodi rizik da će neko dodati polja ili stanje u neke od ovih klasa ne znajući da se koriste kao Singlton objekti.

Singlton ne treba da se koristi u svakoj situaciji u kojoj ga je moguće primeniti kako bi se uštedela memorija. Singlton je statički objekat i preteranim i nepotrebnim korišćenjem ovog šablona uvodite mnoge probleme koje statičke promenljive donose (zaključavanje, sinhronizacija niti, statička inicijalizacija) a pri tome ih maskirate „objektno-orijentisanim dizajnom".

Preterano veliki broj Singltona nekada može da znači da je vaš dizajn suviše proceduralan i u tom slučaju bi možda trebalo da revidirate arhitekturu ako težite pravom objektno orijentisanom dizajnu. Imajte u vidu da i u .NET frejmvorku postoji veliki broj statičkih klasa kao što su `Convert`, `Math` i njima slične koje nisu dizajnirane kao Singltoni. S obzirom na to da je Singlton globalni statički objekat, on nosi sve rizike i probleme koje statičke promenljive mogu da donesu. Svaki put kada se klasa dizajnira kao Singlton, treba imati na umu da je ovo statički objekat i da u slučaju da nema potrebe da se obezbedi jedinstveno stanje, on donosi sve probleme koje statičke promenljive donose.

Prototip

Prototip je način kreiranja objekta po uzoru na jedan postojeći objekat koji je usvojen kao standard. Kao primer se mogu posmatrati etaloni mernih jedinica kao što je masa. U Parizu se nalazi jedan cilindar napravljen od legure iridijuma i platine, koji je od 1889. služio kao međunarodni etalon za kilogram. Ostale institucije koje moraju da znaju koliko tačno iznosi jedan kilogram moraju da naprave svoje tegove kao kopije ovog etalona koje će biti identične mase. Kada se prave novi tipovi automobila ili aviona, često se prvo napravi jedan objekat koji se testira a onda se ostali prave serijski kao njegova kopija. Recepti za hranu i piće su još jedan vid pravljenja objekata kopiranjem. Kada se napravi jedan recept koji zadovoljava zahteve, hrana i piće se prave po tom receptu. DNK je primer prototipa šablona koji definiše karakteristike ćelija živih bića i čijim se kloniranjem prave nove ćelije. U svim ovim slučajevima, prvi objekat koji zadovoljava tražene karakteristike i na osnovu koga se prave kopije naziva se Prototip.

Za razliku od realnog života gde se ne može fizički klonirati prototip, u softveru se novi proizvodi mogu lako praviti kopiranjem. Ova karakteristika nam omogućava da se objekti kreiraju tako što se počinje od određenog stanja ili konfiguracije umesto da se oni prave od početka. Objekti često moraju da se inicijalizuju tako da imaju određene početne vrednosti ili reference ka drugim objektima čim se kreiraju.

U C# jeziku se to može lako postići postavljanjem inicijalnih vrednosti polja objekata prilikom deklaracije klase. To je sasvim prihvatljivo rešenje ako je lako napraviti objekte koje treba referencirati i izračunati vrednosti koje je potrebno postaviti. Međutim, u nekim slučajevima postavljanje početnih vrednosti nije laka ili brza informacija. Zamislimo da se početne vrednosti nekog objekta čitaju iz nekog fajla ili čak i udaljenog servisa, odnosno baze podataka. U ovom slučaju bi se kreiranje objekata značajno usporilo pošto to više ne predstavlja samo dodeljivanje memorije novom objektu, nego i čitanje vrednosti koje sporo traje.

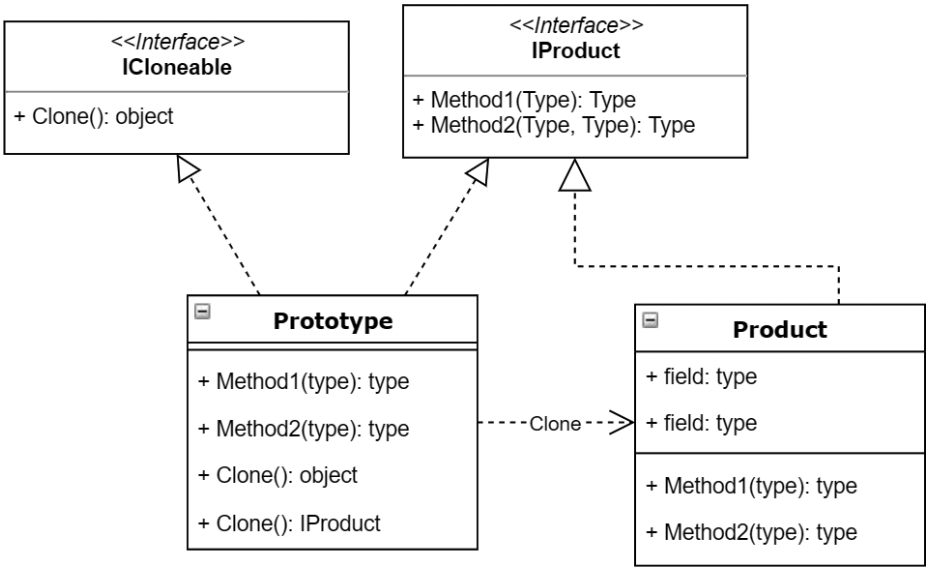
Jedan od najboljih primera su slike. Zamislimo da postoji slika koju treba prikazivati u nekoliko dimenzija. Promena veličine slike je operacija koja troši mnogo resursa i može da traje. Ovaj proces bi se ubrzao ako bi se umesto promene dimenzija originalne slike napravilo nekoliko slika traženih dimenzija a onda se svaki put kada neko traži sliku određene dimenzije samo davala kopija. U ovom slučaju su slike sa zadatim dimenzijama u stvari prototipovi slika koje će biti vraćene korisniku ili korišćene radi iscrtavanja složenih objekata.

U slučaju da se zna koje vrednosti treba pročitati, dovoljno je za svaki podatak koji se čita napraviti po jednu statičku promenljivu ili klasu u koje će biti učitane

inicijalne vrednosti is eksternih sistema. Ove promenljive služe da keširaju vrednosti iz eksternih fajlova, kako se ne bi čitale svaki put prilikom pravljenja novog objekta. Međutim da li su u ovom slučaju uopšte potrebne te statičke promenljive i njihovo kopiranje? Zašto se umesto statičkih promenljivih koje sadrže informacije samo radi inicijalizacije i kopiranja u objekat ne bi one stavile u jedan od objekata i onda kopirao ceo objekat kada je potrebna nova instanca klase?

Prototip je šablon u kome se jednom napravi objekat (tj. prototip objekat) kome se direktno upišu polja prilikom inicijalizacije. Potom se novi objekti kreiraju tako što se objekat prototipa klonira i kopija tog inicijalnog objekta je novokreirani objekat.

UML dijagram šablona prototipa je prikazan na sledećoj slici:



Slika 1.2. UML dijagram šablona prototipa.

Pozivalac želi da dobije novi objekat nekog proizvoda ili servisa i obraća se prototipu tog proizvoda kad god mu je potrebna nova instanca. Prototip klonira objekat koji predstavlja inicijalno stanje objekta i predaje referencu na njega pozivaocu.

Obično postoji samo jedan prototip za sve objekte klase tako da ovaj šablon liči na Singlton. Međutim, osnovna razlika između Prototipa i Singltona leži u činjenici da Singlton uvek vraća referencu na taj jedinstveni objekat, dok prototip pravi kopiju jedinstvenog objekta i uvek vraća kopiju.

Kao primer, pretpostavimo da `Clipboard` objekat prikazan u prethodnoj sekciji ne mora da bude jedinstven nego da je zahtev da više različitih grupa klasa ima svoj

deljeni `Clipboard` objekat preko koga će deliti sadržaj. U ovom slučaju je potrebno inicijalizovati jedan `Clipboard` objekat i napraviti kopiju ovog objekta kad god je neko zatraži.

Implementacija Prototip šablona prikazana je u sledećem primeru:

```
public class Clipboard {

    private static Clipboard instance = null;
    // ili new Clipboard();
    public static Clipboard GetClipboard() {
        if (instance == null) {
            instance = new Clipboard();
        }
        return instance.Clone();
    }

    private Clipboard () {}
    ~Clipboard() { throw new NotImplementedException(); }
    protected new object MemberwiseClone() {
    throw new NotImplementedException(); }

    private object content;

    public void Copy(object content) => this.content = content;

    public object Paste() => this.content;

}
```

Kao što možete primetiti, primer programskog koda je skoro identičan Singltonu, uz jednu bitnu razliku:

Kada se napravi novi objekat, ne vraća se referenca na prototip, nego se prototip objekat klonira i vraća referenca na kopiju tog objekta.

MemberwiseClone

.NET frejmvork obezbeđuje podršku za efikasno kloniranje objekata pomoću metode `MemberwiseClone()` klase `object` iz koje su izvedene sve klase u .NET aplikacijama. Ovo znači da u nekim slučajevima možete da implementirate prototip korišćenjem postojećih funkcionalnosti .NET objekata.

Primer jednostavne implementacije prototipa koji kreira nove objekte pomoću metode `MemberwiseClone()` prikazan je u sledećem listingu:

```
public class Product {

    private static prototype = null;

    public static void SetPrototype(Product obj) {
        prototype = obj;
    }

    public static Product GetProduct() {
        if (prototype == null) {
            throw new  = InvalidOperationException();
        }
        return (Product) prototype.MemberwiseClone();
    }
}
```

Prototip objekta je implementiran u vidu Singlton šablona zato što postoji samo jedan objekat na osnovu koga će se praviti ostali. Kad god je potrebno da se napravi novi objekat, prototip se klonira i vraća se referenca na klon. Prototip objekata se može interno inicijalizovati ili se kao u ovom slučaju obezbeđuje metoda `SetProduct()` kojom eksterni kôd može da postavi inicijalizovanu i unapred konfigurisanu instancu objekata koji će se koristiti kao prototip svih budućih instanci. Ovakav dizajn je dobar izbor u slučaju da se prave prototipovi jednostavnih objekata bez referenci na druge objekte koje treba klonirati zajedno sa objektom ili objekata bez stanja (engl. *Stateless*) koji imaju samo metode.

Metoda `MemberwiseClone()` pravi jednostavnu kopiju (engl. *Shallow copy*) objekta kloniranjem njegovog binarnog sadržaja. U slučaju sa želite da imate punu kontrolu nad načinom kojim se kreiraju objekti i da sami odlučite da li objekti koje referencira objekat koji se klonira moraju isto da budu klonirani ili je dovoljno kopirati reference na njih, potrebno je da implementirate interfejs `ICloneable` i u svojoj metodi definišete eksplicitno kako ga treba klonirati.

Teški objekti

Jedan od najčešćih scenarija gde Prototip može da pomogne je ubrzavanje procesa kreiranja teških objekata. Teški objekti su objekti koji se sporo prave i često pozivanje njihovih konstruktora bi usporilo aplikaciju.

Regularni izrazi su odličan primer objekata za koje je potrebno neko vreme da se naprave. Regularni izrazi se koriste da pronađu određene formate u tekstu kao što su telefoni, imejl adrese, tekst između nekih graničnika i slično. Regularni izrazi se jednostavno zadaju – dovoljno je da se prosledi opis regularnog izraza koji opisuje pravila u nekom tekstu i zatraži od njega da pronađe tekst koji odgovara regularnom izrazu. Primer programskog koda koji pronalazi imejl adresu u tekstu prikazan je u sledećem listingu:

```
Regex regex = new Regex(@"[a-zA-Z]+@([a-zA-Z][.])+[a-zA-Z]+");
Match match = regex.Match("Email: petar@acme.co.rs");
```

Međutim, da bi se stvarno procesirao tekst, potrebno je konvertovati definiciju regularnog izraza u nekada veoma složenu mašinu stanja (vidite šablon Stanja u poglavlju o šablonima ponašanja), što može usporiti aplikaciju posebno ako se istovremeno procesira mnogo tekstova u različitim nitima. Često je preporuka da se regularni izrazi kompajliraju kako bi se poboljšale performanse analize teksta. Umesto kreiranja novih instanci objekata regularnih izraza na osnovu istog izraza koje treba da procesiraju, optimalnije rešenje bi bilo da se napravi jedan kompajlirani regularni izraz koji će se klonirati kada je potrebno ponovo da se procesira. Ovde se polazi od pretpostavke da će kloniranje mašine stanja u vidu kopiranja nizova, funkcija, i ostalih struktura koje čine mašinu stanja biti brže od njihovog kreiranja.

> *Ovo pravilo uglavnom ne važi sa standardne parsere za XML i JSON (npr. JSON.Net) pošto su oni jednom već kompajlirani i u svojim konstruktorima imaju sve strukture koje treba inicijalizovati kada se parser napravi. Međutim, u slučaju da se dinamički pravi parser sa nestandardnom gramatikom, često je potrebno koristiti prvu instancu kao prototip za nove umesto ponovnog kompajliranja.*

Sličan primer je dinamičko kompajliranje programskog koda. Roslin (engl. *Roslyn*) je C# kompajler kome se može proslediti neki C# programski kôd i koji će dinamički generisati tzv. sintaksno stablo programskog koda (vidite Kompozitni šablon) ili napraviti izvršni međukod (engl. IL – *intermediate language*) koji se može izvršiti. U slučaju da postoje neke klase koje moraju da koriste dinamički konstruisani kôd koji se prevodi i izvršava, bolje je zapamtiti instance kompajliranih objekata i klonirati ih svaki put kada se zatraži prevođenje istog programskog koda nego svaki put ponovo prolaziti kroz proces prevođenja.

Eksterno konfigurisanje objekata

U nekim slučajevima programski kôd nema sve potrebne informacije za kreiranje i konfigurisanje objekata. Na primer, ASP.NET frejmvork koristi objekte `context` i `config` kako bi konfigurisao objekte koji se koriste prilikom procesiranja HTTP zahteva. Međutim, inicijalno ovi objekti nemaju neke korisne informacije tako da se moraju inicijalizovati. Na primer, `config` objektu treba definisati odakle bi trebalo da čita konfiguracione parametre (npr. iz fajlova). ASP.NET frejmvork omogućava da se ovi objekti eksterno konfigurišu tako što se u jednom trenutku pozove metoda kojoj se kao parametri proslede ovi objekti i omogućava se da se u toj metodi postave svojstva ovih objekata. U sledećem primeru prikazano je kako se prilikom inicijalizacije `WebHostBuilder` pomoću `ConfigureAppConfiguration` definiše funkcija (u vidu lambda izraza u ovom slučaju) kojoj se predaju objekti koje treba konfigurisati i u čijem telu će se postaviti svojstva ovih objekata. Jedini preduslov je da funkcija koja se prosleđuje ima dva parametra preko kojih će biti prosleđene reference na objekte `context` i `config` koje treba konfigurisati:

```
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        ...
        .ConfigureAppConfiguration((context, config) =>
        {
           var env = context.HostingEnvironment;
           config.AddJsonFile("appsettings.json",
                           optional: true, reloadOnChange: true)
             .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                      optional: true, reloadOnChange: true);
           config.AddEnvironmentVariables();
        })
        ...
        .Build();

}
```

Potrebno je doneti odluku šta se radi sa ovim objektima kada se konfigurišu. U slučaju da se ovi objekti koriste prilikom pokretanja aplikacija ili su ovo Singlton objekti koji se jednom konfigurišu, više ne menjaju i dele među ostalim objektima, ovo je prihvatljiv dizajn.