



TEHNIČKI FAKULTET "MIHAJLO
PUPIN" ZRENJANIN
UNIVERZITET U NOVOM SADU



SOFTVERSKO INŽENJERSTVO 2

dr Zdravko Ivanković, dr Dejan Lacmanović

Sadržaj

1.	Modelovanje softvera.....	7
1.1.	Dijagrami slučajeva korišćenja	8
1.2.	Dijagrami aktivnosti.....	10
1.3.	Dijagram klasa	11
1.3.1.	Asocijacija.....	12
1.3.2.	Kompozicija i agregacija	13
1.3.3.	Generalizacija/specijalizacija.....	13
1.4.	Modelovanje dinamičke interakcije	14
2.	Osnovni koncepti objektno-orjentisanog programiranja	17
2.1.	Klase i elementi klasa	17
2.1.1.	Atributi	18
2.1.2.	Metode	18
2.1.3.	Modifikatori pristupa	18
2.1.4.	Properti.....	19
2.1.5.	Konstruktor	20
2.1.6.	Statička klasa i statički članovi	20
2.1.7.	Sealed klase.....	21
2.1.8.	Partial klase.....	21
2.2.	Nasleđivanje.....	22
2.3.	Polimorfizam.....	24
2.4.	Abstraktne klase.....	26
2.5.	Interfejsi	28
2.6.	Generičke klase.....	30
3.	Arhitektura sistema	33
3.1.	Ciljevi arhitekture	33
3.2.	Principi softverske arhitekture	34
3.3.	Modeli u razvoju softvera	35
3.3.1.	Tradicionalna metodologija	35
3.3.2.	Agilna metodologija.....	36
4.	Dizajn principi i paterni	38
4.1.	Uobičajeni dizajn principi.....	39
4.2.	S.O.L.I.D. dizajn principi.....	39
4.2.1.	Single responsibility princip	40
4.2.2.	Open-close princip	41
4.2.3.	Liskov substitution princip.....	42
4.2.4.	Interface segregation princip.....	44
4.2.5.	Dependency inversion princip	45
5.	Višeslojna arhitektura	49
5.1.	Antipattern – Smart UI.....	49
5.2.	Slojevita aplikacija.....	49
5.1.	Zašto koristiti slojeve.....	50
5.2.	Kreiranje slojeva u praksi	50
5.2.1.	Oblast #1: Formatiranje podataka	51
5.2.2.	Oblast #2: CRUD operacije	51

5.2.3.	Oblast #3: Ugrađene procedure.....	52
6.	Prezentacioni sloj	53
6.1.	Korisnički interfejs i prezentaciona logika	53
6.2.	Odgovornost prezentacionog sloja.....	53
6.2.1.	Nezavisnost od korisničkog interfejsa	53
6.2.2.	Podrška za testiranje.....	54
6.2.3.	Nezavisnost od modela podataka	54
6.3.	Odgovornost korisničkog interfejsa	55
6.3.1.	Koristan prikaz podataka	55
6.3.2.	Komforan unos podataka	55
6.3.3.	Opšti prikaz	55
6.4.	MVC patern	56
6.4.1.	ASP.NET implementacija MVC-a.....	56
7.	Sloj servisa	59
7.1.	Namena sloja servisa.....	59
7.2.	Prednosti servisnog sloja.....	61
7.3.	Paterni u okviru sloja servisa	61
7.4.	Primer aplikacije sa slojem servisa	61
7.4.1.	Kreiranje poslovnog sloja	62
7.4.2.	Kreiranje sloja za pristup podacima.....	64
7.4.3.	Kreiranje sloja servisa	65
7.4.4.	Sloj korisničkog interfejsa	68
7.5.	Idempotent patern	70
8.	Poslovni sloj.....	73
8.1.	Razlaganje poslovnog sloja.....	73
8.1.1.	Objektni model domena	73
8.1.2.	Poslovna pravila.....	74
8.2.	Paterni za kreiranje poslovnog sloja	74
8.2.1.	Proceduralni paterni	75
8.2.2.	Objektno bazirani paterni	75
8.3.	Grupe dizajn paterna	76
8.3.1.	Creational dizajn paterni	76
8.3.2.	Structural dizajn paterni	76
8.3.3.	Behavioral dizajn paterni	76
8.4.	Kako odabrati i primeniti dizajn patern	77
9.	Strukturni dizajn paterni	79
9.1.	Adapter patern.....	79
9.1.1.	UML dijagram Adapter paterna	79
9.1.2.	Demo primer	80
9.1.3.	Realni primer	81
9.2.	Composite patern	85
9.2.1.	UML dijagram Composite paterna	85
9.2.2.	Demo primer	86

9.2.3.	Realni primer	88
9.3.	Decorator pattern	93
9.3.1.	UML diagram Decorator paterna.....	94
9.3.2.	Demo primer	95
9.3.3.	Realni primer	97
9.4.	Facade pattern	99
9.4.1.	UML diagram za Facade patern.....	99
9.4.2.	Demo primer	100
9.4.3.	Realni primer	102
9.5.	Proxy pattern.....	105
9.5.1.	UML diagram Proxy paterna	106
9.5.2.	Demo primer	106
9.5.3.	Realni primer	108
9.6.	Bridge pattern.....	112
9.6.1.	UML diagram Bridge paterna	112
9.6.2.	Demo primer	113
9.6.3.	Realni primer	114
10.	Dizajn paterni za kreiranje objekata	119
10.1.	Prototype pattern	119
10.1.1.	UML diagram za Prototype patern.....	119
10.1.2.	Demo primer	120
10.1.3.	Realni primer	123
10.2.	Factory method pattern	128
10.2.1.	UML diagram za Factory method patern.....	128
10.2.2.	Demo primer	129
10.3.	Singleton pattern	131
10.3.1.	UML diagram za Singleton patern.....	131
10.3.2.	Demo primer	132
10.3.3.	Realni primer	133
10.4.	Abstract Factory pattern.....	135
10.4.1.	UML diagram za Abstract factory patern	135
10.4.2.	Primer paterna.....	136
11.	Dizajn paterni ponašanja.....	139
11.1.	Strategy pattern	139
11.1.1.	UML diagram za Strategy patern.....	139
11.1.2.	Demo primer	139
11.1.3.	Realni primer	141
11.2.	State pattern	143
11.2.1.	UML diagram za State patern	143
11.2.2.	Demo primer	144
11.2.3.	Realni primer	146
11.3.	Template method pattern	150
11.3.1.	UML diagram za Template method	150

11.3.2.	Demo primer	151
11.3.3.	Realni primer	152
11.4.	Command pattern.....	154
11.4.1.	UML dijagram za Command patern	154
11.4.2.	Demo primer	155
11.4.3.	Realni primer	156
11.5.	Iterator pattern.....	159
11.5.1.	UML dijagram za Iterator patern	159
11.5.2.	Demo primer	160
11.5.3.	Realni primer	161
12.	Sloj za pristup podacima.....	165
12.1.	Funkcionalni zahtevi	165
12.2.	Nadležnosti sloja za pristup podacima	166
12.3.	CRUD usluge	167
12.3.1.	Repository patern	167
12.4.	Upiti	168
12.4.1.	Query Object Patern.....	168
12.5.	Upravljanje transakcijama.....	176
12.5.1.	Unit of Work patern	177
12.6.	Obrada konkurentnosti.....	182
12.6.1.	Data Concurrency Control	182
12.6.2.	Identity Map.....	185
13.	LITERATURA	188
14.	Ispitna pitanja.....	189

1. MODELOVANJE SOFTVERA

Modelovanje se koristi u mnogim aspektima života. Prvi put se sreće u drevnim civilizacijama poput Egipta, Grčke i Rima gde se modelovanje koristilo u kreiranju maketa za potrebe umetnosti i arhitekture. Danas je modelovanje u širokoj upotrebi u nauci i inženjerstvu sa ciljem da pruži apstrakciju sistema sa određenim nivoom tačnosti i sa određenim stepenom detalja. Kreirani model se zatim analizira kako bi pružio bolje razumevanje sistema koji se kreira.

U softverskom dizajnu i razvoju baziranom na modelu, modelovanje se koristi kao osnovni deo procesa razvoja softvera. Modeli se kreiraju i analiziraju pre implementacije samog sistema, i služe da usmere implementaciju koja će uslediti.

Sa sve većim brojem notacija i metoda za objektno-orjentisanu analizu i dizajn softverskih aplikacija, stvorila se potreba za jedinstvenim jezikom za modelovanje. Kao rezultat, nastao je UML (Unified Modeling Language) kako bi ponudio standardizovani grafički jezik i notaciju za opis objektno-orjentisanih modela. UML je tokom godina evoluirao u standardni jezik za modelovanje i prikaz objektno-orjentisanog dizajna. Napor u standardizaciji je kulminirao izdavanjem inicijalnog predloga pod nazivom UML 1.0 od strane OMG grupe u januaru 1997. godine. Nakon nekoliko revizija, finalni predlog pod nazivom UML 1.1 je izdat kasnije iste godine, a prilagođen je i kao standard u objektnom modelovanju. Glavna revizija u prikazu je učinjena 2005. godine sa verzijom UML 2.0. Poslednja verzija je UML 2.5 koja je izdata u junu 2015. godine.

Po OMG-u "modelovanje predstavlja dizajn softverskih aplikacija koji se dešava pre kodiranja". OMG promoviše arhitekturu baziranu na modelu kao pristup u kom se UML modeli softverske arhitekture razvijaju pre implementacije. UML predstavlja notaciju za opis rezultata objektno-orjentisane analize i dizajna razvijenih pomoću bilo koje metodologije.

UML model može biti nezavistan od platforme PIM (Platform-Independent Model), ili model namenjen specifičnoj platformi PSM (Platform Specific Model). PIM predstavlja precizan model softverske arhitekture pre nego što se odabere platforma za razvoj. Razvoj PIM-a je posebno koristan jer se isti PIM može mapirati na različite platforme kao što su COM, CORBA, .NET, J2EE, Web servise ili druge platforme.

Bolje razumevanje sistema se može postići ako se on posmatra iz više različitih perspektiva (više različitih pogleda). Savremene metode za objektno-orjentisanu analizu i dizajn koriste kombinaciju modelovanja slučaja korišćenja, modelovanja aktivnosti, statičkog modelovanja, modelovanja stanja i modelovanja interakcije između objekata.

U modelovanju slučajeva korišćenja, funkcionalni zahtevi sistema se definišu u smislu načina upotrebe i učesnika koji koriste ili vrše interakciju sa datim sistemom. Statičko modelovanje daje strukturni pogled na sistem. Klase se definišu u skladu sa njihovim atributima i relacijama sa drugim klasama. Dinamičko modelovanje pruža pogled na sistem u odnosu na ponašanje. Slučajevi korišćenja se kreiraju kako bi prikazali interakciju između objekata koji u njemu učestvuju. Dijagrami interakcije se kreiraju kako bi prikazali kako objekti međusobno komuniciraju u cilju realizovanja slučajeva korišćenja.

Arhitektura softvera razdvaja sveukupnu strukturu sistema, u smislu komponenti i njihove međusobne povezanosti, pa sve do detalja unutrašnje realizacije pojedinih komponenti. Naglasak na komponentama i njihovim međusobnim vezama se ponekad naziva *programiranje-na-veliko*, a detaljan dizajn pojedinih komponenti se naziva *programiranje-u-malom*.

Softverska arhitektura se može opisati na različitim nivoima apstrakcije (sa različitim nivoom detalja). Na višim nivoima, ona može opisivati dekompoziciju sistema u podsisteme. Na nižem nivou, ona može opisivati dekompoziciju podsistema u module ili komponente. U oba slučaja, naglasak je na spoljašnjem pogledu na podsistem/komponentu – odnosno, interfejse koje pruža i zahteva, kao i njihovu međusobnu povezanost sa drugim podsistemima/komponentama.

Softverska arhitektura se ponekad posmatra kao dizajn visokog nivoa. Ona se može opisati korišćenjem različitih UML pogleda. Važno je osigurati da arhitektura ispunjava softverske zahteve, i

to kako funkcionalne (šta softver treba da radi) tako i nefunkcionalne (koliko dobro je potrebno da to radi). Ona predstavlja i startnu tačku za detaljni dizajn i implementaciju, posebno u slučajevima kada tim za razvoj postane dosta velik.

Modelovanje softvera predstavlja opisivanje softverskog dizajna grafički, tekstualno, ili i grafički i tekstualno. Npr. dijagram klasa predstavlja grafički prikaz dizajna, dok pseudo kod predstavlja tekstualni prikaz dizajna.

UML notacija je evoluirala od kad je prvi put prihvaćena kao standard 1997. godine, pa danas podržava veliki broj dijagrama. U razvoju aplikacija se koriste sledeći dijagrami:

- Dijagram slučajeva korišćenja
- Klasni dijagram
- Dijagram objekata
- Komunikacioni dijagram
- Sekvencijalni dijagram
- Dijagram stanja
- Dijagram aktivnosti
- Dijagram razmeštanja

1.1. DIJAGRAMI SLUČAJEVA KORIŠĆENJA

Modelovanje slučajeva korišćenja predstavlja pristup za opisivanje funkcionalnih zahteva sistema. Ulazne informacije i izlazi koje sistem daje, prvo se opisuju pomoću modela slučajeva korišćenja, a zatim se specificiraju pomoću statičkog modelovanja.

U pristupu modelovana pomoću slučajeva korišćenja, funkcionalni zahtevi se opisuju pomoću učesnika, koji predstavljaju korisnike sistema, i slučajeva korišćenja. Slučaj korišćenja definiše sekvencu interakcija između jednog ili više učesnika i sistema. Sistem je predstavljen kao crna kutija, odnosno govori se šta sistem daje kao odgovor na ulaze od strane korisnika, a ne kako to radi. Tokom naknadnih analiza i modelovanja se određuju objekti koji učestvuju u svakom od slučajeva korišćenja.

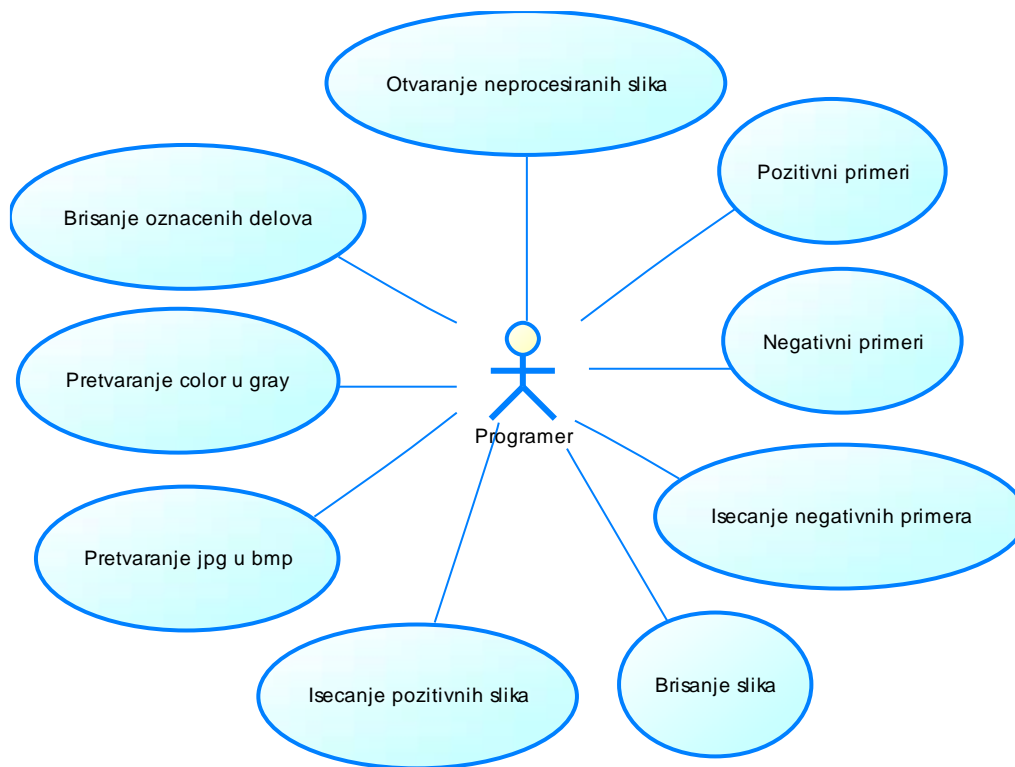
Slučaj korišćenja se obično sastoji od niza interakcija između učesnika i sistema. Svaka interakcija se sastoji od ulaza koji daje korisnik, nakon čega sledi odgovor dobijen od sistema. Dakle, učesnik daje ulaz sistemu, a sistem daje odgovor učesniku. Dok se jednostavni slučajevi korišćenja sastoje od samo jedne interakcije između učesnika i sistema, složeniji slučajevi se sastoje od nekoliko interakcija. Složeniji slučajevi korišćenja mogu uključiti više od jednog učesnika.

Učesnici predstavljaju spoljašnje korisnike koji vrše interakciju sa sistemom. U modelu slučajeva korišćenja, učesnici predstavljaju jedine spoljašnje entitete koji vrše interakciju sa sistemom. Učesnici su izvan sistema, odnosno, ne predstavljaju deo sistema koji se modeluje.

Učesnik predstavlja određenu ulogu koja se odigrava u domenu aplikacije, tipično od strane ljudskog korisnika. Korisnik predstavlja jednu osobu, dok učesnik predstavlja ulogu koju imaju svi korisnici istog tipa. Npr. sve osobe koje prilaze bankomatu da podignu novac se predstavljaju kao jedan učesnik.

Učesnik je veoma često čovek. Iz tog razloga se u UML-u predstavlja korišćenjem čovekolike figure. U nekim informacionim sistemima, ljudi nisu jedini učesnici. Moguće je da učesnik bude spoljašnji sistem koji pruža usluge posmatranom sistemu. U nekim aplikacijama, učesnik može biti i spoljašnji I/O uređaj ili tajmer. Spoljašnji I/O uređaji i tajmeri se veoma često sreću u embedded sistemima koji rade u realnom vremenu, a u kojima sistem vrši interakciju sa spoljašnjim okruženjem putem senzora i aktuatora.

Primer dijagrama slučajeva korišćenja je prikazan na slici 1.1.



Slika 1.1 - Primer dijagrama slučajeva korišćenja

Slučaj korišćenja je kreiran za potrebe alata kojim se kreiraju primeri za obuku AdaBoost algoritma za mašinsko učenje. Kreiranje skupa za obuku sadrži dodatne funkcionalnosti koje su prikazane na slici 1.1. Ove funkcionalnosti uključuju:

- Otvaranje neprocesuiranih slika – Otvaranje skupa slika sa kojih se označavaju objekti od interesa, ukoliko postoje, ili delovi slike koji ne sadrže objekte od interesa.
- Pozitivni primeri – Označavanje objekata od interesa na slici za koje se vrši obuka. Svaki objekat se označava pomoću četiri koordinate, i to x i y koordinate gornjeg levog ugla, i širine i visine objekta. Podaci o objektima se upisuju u poseban fajl kako bi se mogli koristiti tokom obuke.
- Negativni primeri - Ukoliko slika ne sadrži ni jedan objekat od interesa, ona se označava kao negativan primer i služi u procesu obuke. Tokom obuke se pozitivni primeri "lepe" na slike koje predstavljaju negativne primere.
- Isecanje negativnih primera – Ukoliko deo slike ne sadrži objekte od interesa, on se može iseći i označiti kao negativan primer.
- Brisanje slika – Ukoliko slika ne predstavlja dobar ni negativan ni pozitivan primer, ona može biti obrisana kako ne bi loše uticala na sam proces obuke.
- Isecanje pozitivnih slika – Ova funkcionalnost omogućava da se svi pozitivni primeri iseku iz celih slika kako bi se nad njima mogla izvršiti dodatna korekcija (pojačavanje kontrasta, izoštravanje, oduzimanje pozadine, ...).
- Pretvaranje jpg u bmp – Sam AdaBoost algoritam podržava i jpg i bmp format slika, ali kod jpg formata može doći do problema u obuci, pa je poželjno da sve slike budu u bmp formatu.
- Pretvaranje color u gray – AdaBoost algoritam radi nad crno-belim slikama pa se stoga slikama može oduzeti boja kako bi zauzimale manje mesta i kako bi sam algoritam bio brži jer ne mora sam da ih pretvara u crno bele.
- Brisanje označenih delova – Ako se prilikom označavanja pozitivnih primera slučajno označi deo slike koji ne predstavlja objekat od interesa, moguće je poništiti selekciju na posmatranoj slici i ponovo označiti objekte od interesa.

1.2. DIJAGRAMI AKTIVNOSTI

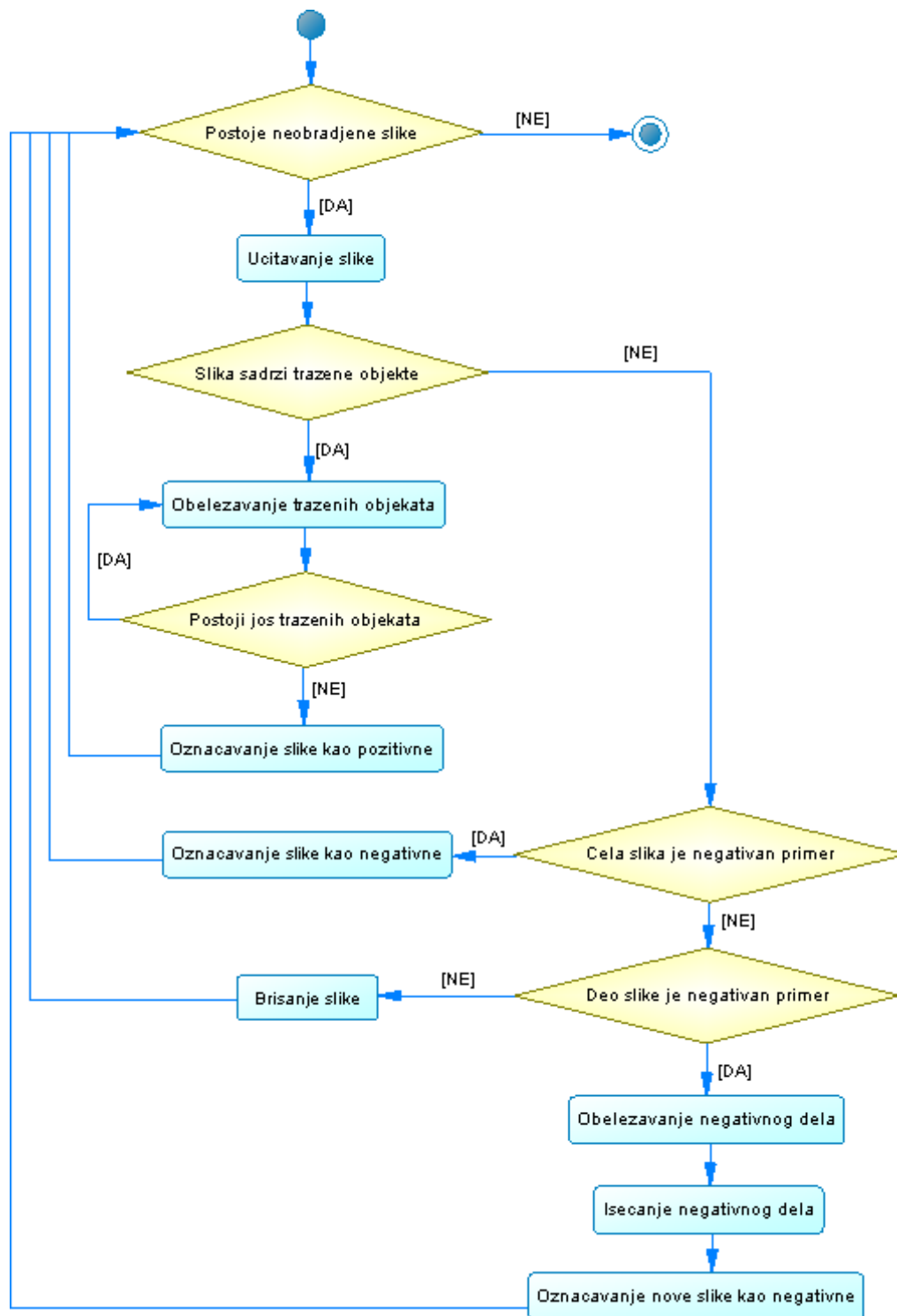
Dijagrami aktivnosti predstavljaju UML dijagrame koji prikazuju tokove kontrole i sekvence koje se dešavaju tokom softverske aktivnosti. Dijagram aktivnosti prikazuje sekvence aktivnosti, čvorove odluke, skokove, pa čak i konkurentne aktivnosti. Ovi dijagrami se dosta primenjuju u modelovanju tokova aplikacije, npr. kod servisno-orijentisanih aplikacija.

Model slučajeva korišćenja se može detaljnije opisati korišćenjem dijagrama aktivnosti. Dijagram aktivnosti se može koristiti da predstavi redosled koraka u slučaju korišćenja, uključujući glavnu sekvencu i sve alternativne sekvence. Drugim rečima, dijagram aktivnosti se može koristiti u cilju preciznijeg opisa slučaja korišćenja jer tačno prikazuje mesto i uslove u sekvenci koji su potrebni da bi se krenulo alternativnim putem izvršavanja. Čvor aktivnosti se može koristiti da prikaže jedan ili više koraka u slučaju korišćenja. Čvor aktivnost visokog nivoa se može koristiti da prikaže ceo slučaj korišćenja, pri čemu se on kasnije može dekomponovati u poseban dijagram aktivnosti.

U cilju prikazivanja slučajeva korišćenja, dijagrami aktivnosti koriste čvorove aktivnosti, čvorove odluke, lukove koji povezuju čvorove aktivnosti i skokove. Čvor aktivnosti se koristi da prikaže jedan ili više koraka potrebnih da bi se opisao slučaj korišćenja. Čvor odluke se koristi da prikaže situaciju u kojoj se na osnovu rezultata odluke može krenuti alternativnim putem izvršavanja. U zavisnosti od slučaja korišćenja, alternativna sekvenca se kasnije može spojiti sa glavnom sekvencom.

Čvorovi aktivnosti mogu biti agregisani čvorovi koji se hijerarhijski dekomponuju kako bi dali dijagram aktivnosti nižeg nivoa. Ovaj koncept se može koristiti kako bi prikazao slučajeve korišćenja sa uključenjem i proširenjem. Dakle, čvor aktivnosti u baznom slučaju korišćenja se može koristiti da predstavi vezu sa slučajem korišćenja koji predstavlja uključanje (ili proširenje), a koji se zatim prikazuje na posebnom dijagramu aktivnosti nižeg nivoa.

Slučaj korišćenja "Kreiranje skupa za obuku" se može prikazati dijagramom aktivnosti prikazanim na slici 1.2. To je proces kreranja skupa za obuku gde je cilj prepoznavanje igrača na košarkaškim utakmicama. Drugim rečima, skup predstavlja jasne slike košarkaša tokom utakmice. Na dijagramu se vidi da je prvi korak proveriti da li uopšte postoje slike za obuku. Ove slike se kreiraju tokom prikazivanja video sadržaja. Ukoliko slike ne postoje, prekida se izvršavanje aplikacije jer ne postoji skup slika nad kojima bi se pronalazili primeri za obuku. Ako slike postoje, one se jedna po jedna učitavaju i prikazuju u prozoru aplikacije. Slike dobijene iz video sadržaja mogu, ali i ne moraju sadržati objekte od interesa. Ako se posmatra košarkaška utakmica, postoje kadrovi koji ne sadrže košarkaše (reklame, najave događaja, snimci publike, zumiranje poznatih ličnosti,...) pa se na njima ne može vršiti obuka. Oni mogu biti korišćeni kao negativni primeri, odnosno primeri koji će u obuci služiti kako bi se proverilo da algoritam ne označava sve objekte kao tražene, odnosno da se provere performanse obuke. Postoje slike koje se ne mogu okarakterisati ni kao pozitivni ni kao negativni primeri. To su obično slike košarkaša koje su dosta nejasne, pa bi postojala bojazan da će odvesti obuku u pogrešnom smeru. Takve slike je najbolje u potpunosti isključiti iz procesa obuke, odnosno obrisati ih. Ponekad delovi slike ne sadrže ni jedan objekat od interesa pa se mogu iseći u posebnu sliku koja će biti označena kao negativan primer. Kod slika koje sadrže tražene objekte, potrebno ih je označiti, odnosno odrediti x i y koordinate gornjeg levog ugla pravougaonika koji okružuje tražene objekte, kao i njegovu visinu i širinu. Proces se ponavlja za sve objekte na slici koji mogu doprineti obuci. Objekte koji nisu korektno prikazani treba izostaviti (npr. veći deo objekta je zaklonjen od strane drugih objekata). Ovi podaci se upisuju u poseban fajl koji će biti korišćen u procesu obuke da se iz pozitivnih slika iseku samo traženi objekti dok se ostatak slike zanemaruje.



Slika 1.2 - Primer dijagrama aktivnosti

1.3. DIJAGRAM KLASA

Klasni dijagram se odnosi na statički strukturni pogled na problem, koji je nepromenljiv u odnosu na vreme. Preciznije, statički model definiše klase u sistemu, atribute klasa, relacije između klasa i operacije koje poseduje svaka klasa.

Objekat (naziva se i instanca objekta) predstavlja pojedinačnu "stvar", npr. Markov automobil ili Petrov bankovni račun. Klasa (naziva se i klasa objekata) predstavlja kolekciju objekata sa istim karakteristikama, npr. bankovni račun, automobil, klijent.

Atribut predstavlja vrednost podataka koju sadrži objekat klase. Svaki objekat ima sopstvene vrednosti atributa. Ime atributa je jedinstveno u okviru klase, ali različite klase mogu imati attribute sa istim imenom; npr. klase Klijent i Poslodavac mogu imati atribut sa imenom Adresa.

Operacija predstavlja specifikaciju funkcije koju izvršava objekat. Objekat poseduje nijednu, jednu ili više operacija. Operacije manipulišu vrednostima atributa koje su sadržane u objektu. Operacije mogu posedovati ulazne i izlazne parametre. Svi objekti koji pripadaju istoj klasi imaju i iste operacije. Npr. klasa Račun poseduje operacije očitavanje, kredit, otvaranje i zatvaranje.

U statičkom modelovanju se koriste tri tipa relacija: asocijacije, relacije celina/deo (kompozicija i agregacija) i relacije generalizacije/specijalizacije (nasleđivanje).

1.3.1. ASOCIJACIJA

Asocijacija definiše relaciju između dve ili više klasa i označava statičku, strukturalnu relaciju. Npr. *Zaposleni Radi u Odeljenju*, gde su *Zaposleni* i *Odeljenje* klase a *Radi u* predstavlja asocijaciju. Klase predstavljaju imenice, dok asocijacija predstavlja glagol.

Link predstavlja konekciju između instanci klasa (objekata) i predstavlja instancu asocijacije između klasa. Npr. *Petar radi u proizvodnji*, pri čemu je *Petar* instanca klase *Zaposleni* a *proizvodnja* predstavlja instancu klase *Odeljenje*. Link može postojati između objekata ako i samo ako postoji asocijacija između odgovarajućih klasa.

Asocijacije su po svojoj prirodi dvosmerne. Prethodni primer se može napisati i kao *U odeljenju rade zaposleni*. Asocijacije su najčešće binarne, odnosno, predstavljaju relacije između dve klase. Međutim one mogu biti i unarne (samo-asocijacija), ternarne, ili višeg reda.

U klasnim dijagramima, asocijacije se prikazuju kao puna linija koja povezuje dve klase, zajedno sa imenom asocijacije.

Višestrukost asocijacije navodi koliko instanci jedne klase može biti povezano sa instancom druge klase. Višestrukost asocijacije može biti sledeća:

- **Jedan-prema-jedan asocijacija.** U jedan-prema-jedan asocijaciji između dve klase, asocijacija je jedan-prema-jedan u oba smera. Stoga, objekat bilo koje od klasa poseduje link ka samo jednom objektu druge klase. Npr. u asocijaciji *Kompaniju vodi CEO*, posmatrana kompanija poseduje samo jednog CEO-a, a i CEO vodi samo jednu kompaniju
- **Jedan-prema-više asocijacija.** U jedan-prema-više asocijaciji, postoji jedan-prema-više asocijacija u jednom smeru između dve klase i jedan-prema-jedan asocijacija između njih u suprotnom smeru. Npr. u asocijaciji *Banka Administrira Račun*, posmatrana banka administrira veći broj računa, ali svaki od tih računa je administriran od strane samo jedne banke.
- **Numerički navedena asocijacija.** Numerički navedena asocijacija predstavlja asocijaciju koja se odnosi na određeni broj. Npr. u asocijaciji *Kola Imaju Vrata*, jedan automobil poseduje dvoje ili četvoro vrata, ali nikad jedna, troja ili petoro (ako u vrata ne računamo gepek). Asocijacija u suprotnom smeru je i dalje jedan-prema-jedan, odnosno, vrata pripadaju jednom automobilu. U prethodnom primeru, proizvođač određuje koliko vrata automobil može imati; drugi proizvođač može kreirati automobile sa drugačijim brojem vrata.
- **Opciona asocijacija.** U opcionalnoj asocijaciji, ne mora uvek postojati link od objekata u jednoj klasi do objekata u drugoj klasi. Npr. u asocijaciji *Klijent Posедуje Kreditnu karticu*, klijent može birati da li želi da poseduje kreditnu karticu ili ne. Moguće je imati nula, jednu ili više asocijacija. Asocijacija u suprotnom smeru je jedan-prema-jedan, tj. *Kreditnu karticu Posедуje jedan Klijent*).

- **Više-ka-više asocijacija.** Više-ka-više asocijacija predstavlja asocijaciju između dve klase u kojoj se jedan-ka-više asocijacija nalazi u oba smera. Npr. u asocijaciji *Kurs Pohađa Student*, *Student se Prijavio na Kurs*, postoji jedan-prema-više asocijacija između kursa i studenta koji ga pohađa, jer kurs pohađa veći broj studenata. Takođe postoji jedan-prema-više asocijacija u suprotnom smeru, jer se student može prijaviti na veći broj kurseva.

1.3.2. KOMPOZICIJA I AGREGACIJA

I kompozicija i agregacija se odnose na klase koje su sačinjene od drugih klasa. Kompozicija i agregacija predstavljaju specijalne forme relacija u kojima su klase povezane pomoću *celina/deo* veze. U oba slučaja, relacija između delova i celine je *Je deo od* relacija.

Kompozicija predstavlja jaču relaciju u odnosu na agregaciju, a agregacija predstavlja jaču relaciju u odnosu na asocijaciju. Kompozicija takođe predstavlja i relaciju između instanci klase. Stoga, objekti koji predstavljaju delove se kreiraju, žive, i umiru zajedno sa celinom. Delovi mogu pripadati samo jednoj celini.

Kombinovana klasa često uključuje fizičku relaciju između celine i delova. Npr. bankomat predstavlja kombinovanu klasu koja se sastoji od četiri dela: čitača kartica, dela za izdavanje novčanica, uređaja koji štampa izveštaj i displeja koji omogućava interakciju. Ova klasa poseduje jedan-prema-jedan asocijaciju sa svakim od delova.

Agregacija predstavlja slabiju formu celina/deo relacije. U agregaciji instance delova mogu biti dodavane i uklanjane od celine. Iz ovog razloga, agregacija se više koristi u modelovanju konceptualnih klasa u odnosu na fizičke klase. Pored toga, deo može pripadati ka više agregacija. Primer agregacije može biti fakultet čiji delovi su administracija, katedre i istraživački centri. Moguće je kreirati nove katedre, uklanjati katedre ili ih spajati.

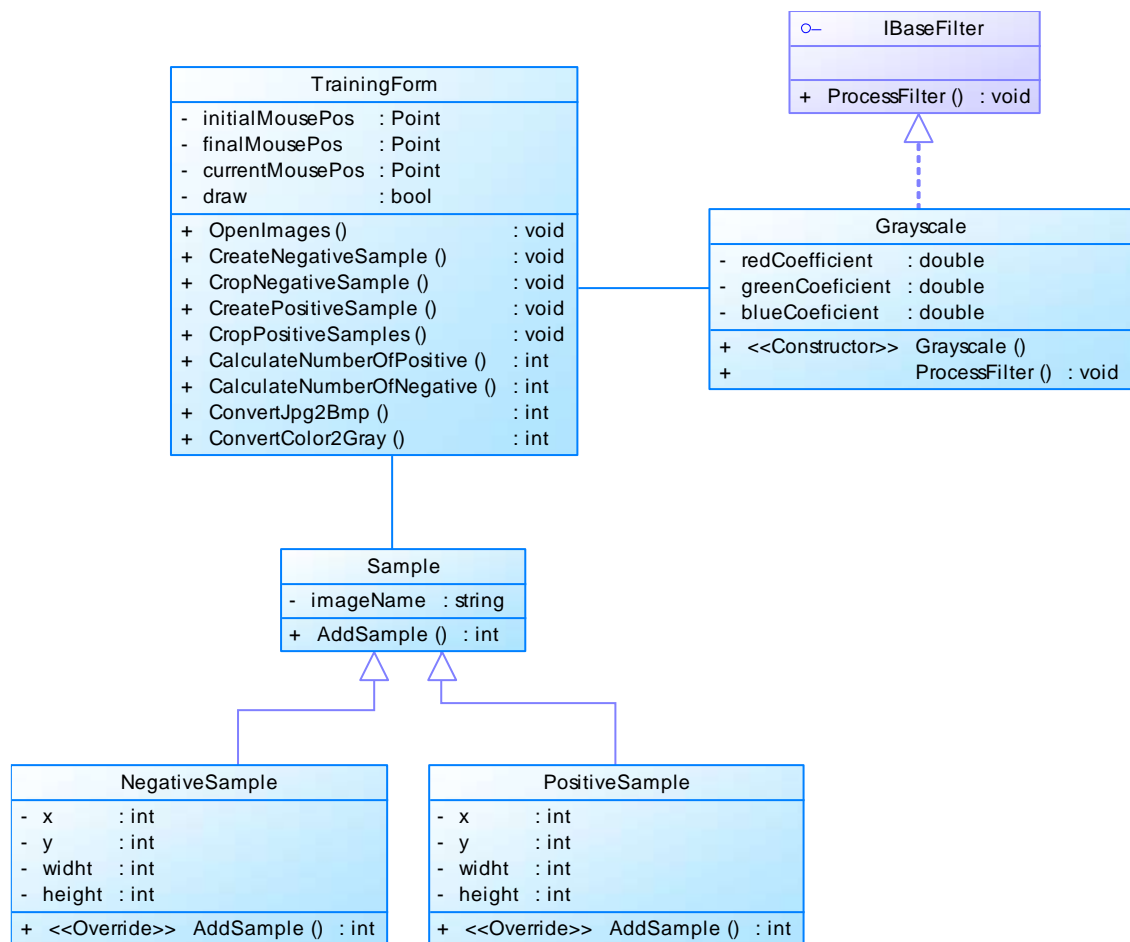
I u kompoziciji i u agregaciji atributi se propagiraju od celine ka delovima. Npr. svaki bankomat poseduje svoj Id koji takođe identifikuje čitač kartica, uređaj za štampu i displej, a koji predstavljaju delove tog bankomata.

1.3.3. GENERALIZACIJA/SPECIJALIZACIJA

Postoje klase koje su dosta slične, međutim nisu identične. One imaju neke zajedničke attribute, ali se neki atributi i razlikuju. U relaciji generalizacije/specijalizacije, zajednički atributi se izdvajaju u generalizovanu klasu, koja se naziva i *superklasa*. Različiti atributi predstavljaju osobine specijalizovanih klasa, koje se nazivaju *podklase*. Postoji *Je* relacija između podklase i superklase. Superklasa se naziva i klasa roditelj ili klasa predak. Podklasa se naziva klasa dete ili klasa potomak.

Svaka podklasa nasleđuje osobine od superklase, a zatim proširuje ove osobine na sopstveni način. Osobine klase su njeni atributi i operacije. Nasleđivanje dozvoljava prilagođavanje superklase kako bi se kreirale podklase. Podklasa nasleđuje attribute i operacije od superklase. Podklasa zatim može dodati attribute i operacije, ili definisane operacije na drugačiji način. Svaka podklasa može i sama biti superklasa drugim klasama.

Primer dijagrama klasa je prikazan na slici 1.3. Kada se video materijal pretvori u frejmove, iz njih je moguće vršiti obuku AdaBoost algoritma tako što se na frejmovima obeležavaju objekti od interese (pozitivni primeri), kao i slike koje ne sadrže objekte od interesa (negativni primeri). Slika prikazuje implementaciju dela aplikacije koji omogućava obeležavanje pozitivnih i negativnih primera, pretvaranje slika iz jpg u bmp format i pretvaranje kolor slika u crno bele.



Slika 1.3 - Primer dijagrama klasa

1.4. MODELOVANJE DINAMIČKE INTERAKCIJE

Modelovanje dinamičke interakcije je bazirano na realizaciji slučajeva korišćenja kreiranih tokom njihovog modelovanja. Za svaki slučaj korišćenja neophodno je odrediti kako objekti koji učestvuju u njemu međusobno vrše interakciju. Primjenjuje se kriterijum za strukturiranje objekata kako bi se odredili objekti koji učestvuju u bilo kom od slučajeva korišćenja. Objekti koji realizuju slučaj korišćenja dinamički sarađuju jedan sa drugim što se prikazuje UML komunikacionim dijagramima ili UML dijagramima sekvence.

Komunikacioni dijagrami predstavljaju UML dijagrame interakcije koji prikazuju dinamički pogled na grupu objekata koji vrše interakciju jedni sa drugima tako što prikazuju sekvence poruka koje se razmenjuju između njih. Tokom analitičkog modelovanja, komunikacioni dijagrami se kreiraju za svaki slučaj korišćenja i tom prilikom se prikazuju samo oni objekti koji učestvuju u posmatranom slučaju korišćenja. Na komunikacionom dijagramu, redosled kojim objekti učestvuju u slučaju korišćenja se prikazuje pomoću brojeva koji označavaju redosled poruka. Redosled poruka prikazan na komunikacionom dijagramu bi trebao da odgovara redosledu interakcija između učesnika i sistema koji je već prikazan na slučaju korišćenja.

Interakcija između objekata se može prikazati i pomoću dijagrama sekvence, koji prikazuje interakciju između objekata uređenu po vremenskom redosledu. Dijagram sekvenci pokazuje objekte koji učestvuju u interakciji i redosled po kom se šalju poruke. Ovi dijagrami mogu prikazati i skokove i iteracije. Dijagrami sekvenci i komunikacioni dijagrami prikazuju slične (mada ne obavezno i identične) informacije, ali na različite načine. Obično se koristi ili sekvencijalni dijagram ili komunikacioni dijagram kako bi se dao dinamički pogled na sistem, ali se mogu koristiti i oba. Pošto

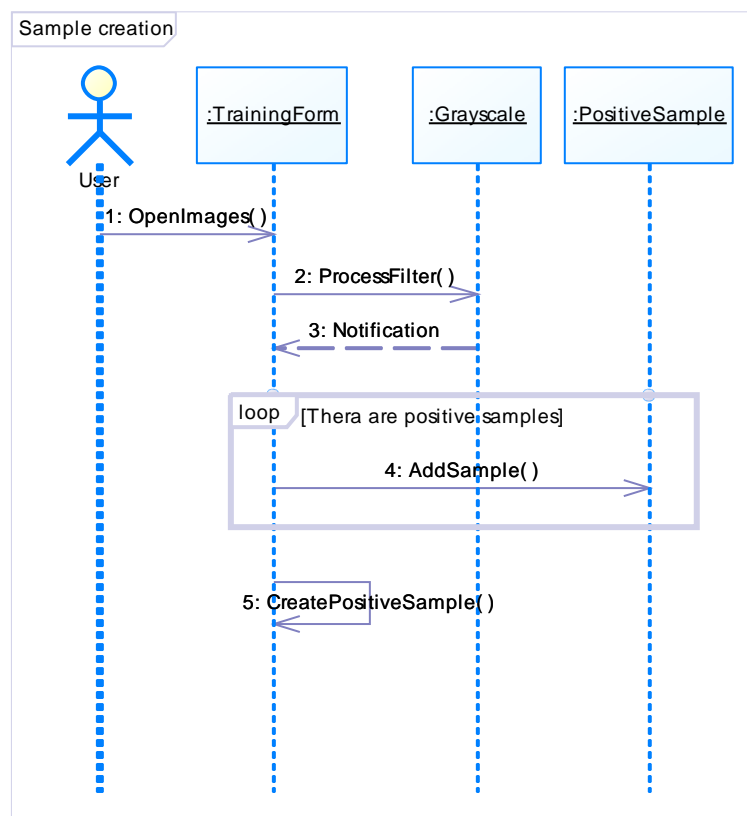
sekvencijalni dijagram prikazuje redosled poruka u redosledu od vrha ka dnu dijagrama, numerisanje poruka nije neophodno.

Prilikom analize, poruke predstavljaju informacije koje se prosleđuju između objekata. Dijagrami interakcije (komunikacioni dijagrami i dijagrami sekvence) pomažu da se odrede operacije izvršene od strane objekata jer dolazak poruke do nekog objekta obično povlači sa sobom izvršavanje operacije. U fazi analize, sve poruke koje se prenose između objekata se prikazuju kao proste poruke. Tip poruka koje se prenose između objekata, sinhrono ili asinhrono, predstavlja odluku koju treba ostaviti za fazu dizajna.

I komunikacioni dijagram i dijagram sekvence se može koristiti u cilju prikazivanja interakcije između objekata i redosleda poruka koje se prosleđuju između objekata. Sekvencijalni dijagrami jasno prikazuju redosled u kom se poruke šalju između objekata, međutim, u njemu je teže uočiti kako su objekti povezani jedan sa drugim. Korišćenje iteracija (kao što je do-while) i naredbi grananja (if-then-else) može sadržati nedovoljno informacija kada je u pitanju redosled interakcija između objekata.

Komunikacioni dijagram prikazuje raspored objekata, tačnije, kako su objekti međusobno povezani. Redosled poruka se prikazuje na oba dijagrama. Pošto je redosled poruka koji se prikazuje na komunikacionim dijagramima manje vidljiv u odnosu na dijagrame sekvence, redosled poruka se numerišu. Međutim, čak i sa numerisanim porukama na komunikacionom dijagramu, ponekad je potrebno više vremena da bi se uočio redosled. Sa druge strane, ako interakcija uključuje veliki broj objekata, dijagram sekvenci može postati težak za tumačenje.

Scenario se definiše kao jedna putanja kroz slučaj korišćenja. Stoga, redosled poruka koji se prikazuje na dijagramu interakcije ustvari prikazuje scenario a ne slučaj korišćenja. Da bi se prikazale sve moguće putanje kroz slučaj korišćenja često je potrebno kreiranje više dijagrama interakcije.



Slika 1.4 - Primer dijagrama sekvence

Dijagram kreiranja pozitivnih primera je prikazan na slici 1.4. Kreiranje pozitivnih primera kreće sa forme koja je predstavljana klasom TrainingForm. Sledeći korak je pretvaranje slike u crno belu pomoću klase Grayscale i metode ProcessFilter. Nakon toga se vrši označavanje pozitivnih primera na slici sve dok ima pozitivnih primera, što je i prikazano fragmentom interakcije na slici 1.4. Kada su svi

pozitivni primeri obeleženi, njihove koordinate se upisuju u tekstualni fajl metodom `CreatePositiveSample`.

2. OSNOVNI KONCEPTI OBJEKTNO-ORJENTISANOG PROGRAMIRANJA

Pre objektno orjentisanog pristupa, svaki program je predstavljao interakciju modula i podprograma. Programiranje je bilo proceduralno, što znači da je postojao glavni tok koda koji je određivao različite korake koje je trebalo izvršiti.

Objektno orjentisani dizajn predstavlja prekretnicu u dizajniranju softvera. On omogućava da se program tretira kao skup objekata koji vrše interakciju pri čemu svaki od njih sadrži svoje sopstvene podatke i ponašanje.

Opšte prihvaćena definicija OOD-a (Object Oriented Design) se može naći u knjizi "Design Patterns: Elements of Reusable Object-Oriented Software" čiji su autori Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides. Celokupna suština OOD je sadržana u sledećoj rečenici: "Moraju se pronaći odgovarajući objekti koji će biti izdvojeni po odgovarajućoj granularnosti, definisati interfejsi klase i hijerarhija nasleđivanja, i ustanoviti osnovne relacije između njih". U istoj knjizi se takođe navodi da "dizajn mora biti specifičan za dati problem ali u isto vreme i dovoljno opšt, da bi odgovorio na buduće probleme i zahteve".

U objektno-orjentisanom programiranju, u sklopu opisa problema, potrebno je uočiti entitete (jedinice posmatranja) koji se nalaze u svetu (domen) u kojem se nalazi i problem koji se rešava. Potrebno je uočiti koji se entiteti nalaze u svetu, opisati njihove osobine i navesti operacije nad njima a kojima se problem rešava. U objektno-orjentisanoj terminologiji, entiteti su opisani klasama, osobine su atributi, a operacije su metode.

2.1. KLASA I ELEMENTI KLASA

Klasa predstavlja kolekciju entiteta. Model entiteta u okviru datog domena problema se kreira pomoću odabranih relevantnih osobina entiteta. Entitet poseduje sve osobine, a njegov model samo odabrane. U C# programskom jeziku sve je klasa. U sledećem primeru je data definicija klase *Pravougaonik*.

```
class Pravougaonik
{
    double stranicaA;
    double stranicaB;

    public double GetO()
    {
        return 2 * stranicaA + 2 * stranicaB;
    }

    public double GetP()
    {
        return stranicaA * stranicaB;
    }
}
```

Listing 2.1 - Kod u okviru klase *Pravougaonik*

U sledećem kodu, mi koristimo klasu *Pravougaonik* kako bi smo deklarirali promenljivu sa imenom *p*:

```
Pravougaonik p = new Pravougaonik();
```

Listing 2.2 - Instanciranje klase *Pravougaonik*

U prethodnom primeru, *p* je promenljiva tipa *Pravougaonik* koji predstavlja klasu. U objektno-orjentisanom programiranju *p* zovemo **objekat** ili **instanca** klase *Pravougaonik* (ne koristimo termin promenljiva).

Definicija klase predstavlja plan koji određuje šta određeni tip može da uradi. Objekat je u suštini memorijski blok koji je alocirani i konfigurisan prema tom planu. Program može kreirati veći broj objekata jedne klase.

Svaka instanca određene klase može pristupiti svojstvima i metodama te klase. Na primer, objekat *p* može pristupiti sledećoj metodi:

```
p.Get0();
```

Listing 2.3 - Poziv metode nad instancom klase

2.1.1. ATRIBUTI

Atribut predstavlja objekat nekog tipa unutar klase. Da bi se dodao atribut, potrebno je navesti njegov tip i ime. U sledećem primeru je kreiran atribut *stranicaA*:

```
double stranicaA;
```

Listing 2.3 - Kreiranje atributa *stranicaA*

Atributi klasa imaju podrazumevane vrednosti. Reference kao atributi klasa imaju podrazumevanu vrednost *null* što može izazvati *NullPointerException* grešku u radu aplikacije.

Na steku su uvek samo reference na objekat, pri čemu su objekti zapravo na heap-u. U C# programskom jeziku je sve referenca (nema pokazivača). Objekat kao atribut može imati referencu na drugi objekat koji je na heap-u.

2.1.2. METODE

Metode oslikavaju ponašanje klasa, odnosno šta objekat određene klase može da uradi. Na primer, klasa *Automobil* može da ima metodu *Ubrzaj*, klasa *Pravougaonik* metode *GetO* i *GetP*:

```
public double Get0()  
{  
    return 2 * stranicaA + 2 * stranicaB;  
}
```

Listing 2.4 - Kreiranje metode *GetO*

U klasi može da postoji više metoda sa istim imenom. Onda se one razlikuju po parametrima i to po broju parametara i/ili po tipovima parametara. Metode se nikad ne razlikuju samo po povratnoj vrednosti. Metode mogu da vrate rezultat nakon izvršavanja (kao što se vidi u prethodnom primeru) ili da ne vrate ništa (tip *void*).

2.1.3. MODIFIKATORI PRISTUPA

Članovima klase (atributima i metodama) se može pristupiti iz instance te klase u zavisnosti od prava pristupa koji im je dodeljen. Dodeljivanje prava pristupa se vrši kroz modifikatore pristupa. Ukoliko se ne navede modifikator pristupa, smatra se da su članovi **private**. U C# programskom jeziku se koriste sledeći modifikatori:

- **private** – ovi članovi su dostupni samo unutar klase u kojoj su deklarirani. Njima se ne može pristupiti iz drugih klasa, kao ni iz klasa koje su izvedene iz te klase
- **public** – ovi članovi su dostupni bilo kom kodu koji ima pristup klasi u kojoj je član deklarisan
- **protected** – ovi članovi su dostupni klasi u kojoj su deklarirani i klasama koje su izvedene iz date klase
- **internal** – ovi članovi su dostupni samo klasama koje se nalaze u istom assembly-ju. Assembly predstavlja kolekciju tipova i resursa koji čine jednu funkcionalnu logičku jedinicu. U .NET-u oni mogu imati .dll ili .exe ekstenziju.

- **protected internal** – ovi članovi su dostupni klasama koje se nalaze u istom assembly-ju ili klasama koje su izvedene iz date klase

U sledećem primeru se kreira klasa *Pravougaonik* koja ima javne metode *GetO* i *GetP*.

```
class Pravougaonik
{
    double stranicaA;
    double stranicaB;

    public double GetO()
    {
        return 2 * stranicaA + 2 * stranicaB;
    }

    public double GetP()
    {
        return stranicaA * stranicaB;
    }
}
```

Listing 2.5 - Kod u okviru klase *Pravougaonik*

Sada se iz main metode za objekat *p* mogu pozivati metode *GetO* i *GetP*:

```
class Program
{
    static void Main(string[] args)
    {
        Pravougaonik p = new Pravougaonik();
        p.GetO();
        p.GetP();
    }
}
```

Listing 2.6 - Kod za *main* metodu

2.1.4. PROPERTI

Properti služe za kontrolisan pristup atributima klase. Properti je u suštini metod koji nam služi da kreiramo logiku kada se uzima vrednost atributa ili kada se ona postavlja:

```
double stranicaA;

public double StranicaA
{
    get { return stranicaA; }
    set
    {
        if (value < 0)
            value = 0;
        stranicaA = value;
    }
}
```

Listing 2.7 - Properti za atribut *stranicaA*

Kao što se može videti iz prethodnog primera, mi kontrolišemo vrednost koja se postavlja za atribut *stranicaA*. Ukoliko je vrednost manja od nule, postavljamo da je vrednost stranice jednaka nuli. Properti pod imenom *StranicaA* je zapravo metoda koja poseduje dva dela: **get** i **set**. Deo **get** se izvršava svaki put kada želimo da pristupimo vrednosti sadržanoj u atributu *stranicaA*, dok se **set** deo izvršava kada prosleđujemo vrednost koju želimo da smestimo u atribut *stranicaA*. Tip podataka atributa i proprietija moraju da se podudaraju.

Iz svojstva se mogu izostaviti **get** ili **set** deo, ali ne mogu oba. Ukoliko se izostavi **set** deo, svojstvo je read-only. Sa druge strane, ako se izostavi **get** deo, svojstvo je write-only.

2.1.5. KONSTRUKTOR

Konstruktor predstavlja metod unutar klase sa posebnim karakteristikama. Ime konstruktora je isto sa imenom klase u kojoj je deklarisan. Konstruktor predstavlja metod koji se poziva svaki put kada koristimo ključnu reč **new** da bi kreirali novu instancu klase. Klasa može da nema ni jedan, jedan ili više konstruktora. Ukoliko se u klasi ne deklarise konstruktor, C# automatski dodaje podrazumevani konstruktor bez parametara. Konstruktor služi da bi se postavile vrednosti atributa i zauzeo neophodan memorijski prostor za objekat:

```
class Pravougaonik
{
    public Pravougaonik()
    {
        StranicaA = 2;
        StranicaB = 3;
    }

    public Pravougaonik(double strA, double strB)
    {
        StranicaA = strA;
        StranicaB = strB;
    }

    ...
}
```

Listing 2.8 - Konstruktori za klasu *Pravougaonik*

U prethodnom primeru se mogu videti dva konstruktora. Prvi konstruktor nema parametre i služi da se postave podrazumevane vrednosti za stranice pravougaonika. Drugi konstruktor ima dva ulazna parametra: *strA* i *strB*. Ovo su vrednosti koje se prosleđuju prilikom kreiranja nove instance klase *Pravougaonik* i koje će predstavljati nove vrednosti za njegove stranice.

2.1.6. STATIČKA KLASA I STATIČKI ČLANOVI

Statičke klase predstavljaju vrstu klase koje se ne mogu instancirati. To znači da se ne može kreirati objekat tipa statičke klase upotrebom ključne reči **new**. Statička klasa može imati samo statičke članove. Njoj se pristupa putem imena klase i navođenjem njenog statičkog člana:

```
static class Math
{
    public static double Add(double x, double y)
    {
        return x + y;
    }
}
```

Listing 2.9 - Kod u okviru statičke klase *Math*

U prethodnom primeru je deklarirana statička klasa *Math* koja ima statičku metodu *Add*. Ovoj metodi se može pristupiti na sledeći način:

```
class Program
{
    static void Main(string[] args)
    {
        double result = Math.Add(1.2, 1.4);
        Console.WriteLine(result);
        Console.ReadKey();
    }
}
```

```
}
```

Listing 2.10 - Pristup statičkoj metodi

Statičke metode se mogu koristiti kao veoma zgodan kontejner za skupove metoda koje koriste samo ulazne parametre i ne postavljaju niti uzimaju vrednosti atributa nekih instanci. U prethodnom primeru, metoda *Add* koristi dva parametra i vraća njihov zbir. Pošto ne koristi vrednosti drugih instanci, ova metoda je kreirana kao statička da bi joj se lakše pristupalo.

Osnovne karakteristike statičkih klasa su:

- sadrže samo statičke članove
- ne mogu biti instancirane
- ove klase su **sealed** (ne mogu se nasleđivati)
- ne sadrže konstruktore

Kreiranje statičkih klasa je u suštini isto kao i kreiranje klasa koje sadrže samo statičke članove i privatni konstruktor. Privatni konstruktor sprečava mogućnost kreiranja instanci. Prednost korišćenja statičkih klasa je u tome da kompajler osigurava da ni jedna instanca date klase nije kreirana.

Ne-statička klasa može sadržati statičke metode, attribute ili događaje. Statički član klase se može pozivati i kada ni jedna instanca klase nije kreirana. Njima se pristupa preko imena klase a ne preko imena instance. U memoriji postoji samo jedna kopija statičkog člana, bez obzira koliko instanci te klase postoji. Statičke metode i propertiji ne mogu pristupati ne-statičkim atributima i događajima u okviru tipa u kom su definisani. One takođe ne mogu pristupiti ni jednom objektu bilo koje klase, ukoliko on nije eksplicitno prosleđen kao parametar u njenom pozivu.

Češće se deklarišu ne-statičke klase koje poseduju određene statičke članove, nego što se deklarišu cele klase kao statičke. Dve najčešće upotrebe statičkih atributa je da se čuva broj objekata koji su instancirani nad određenom klasom, ili da se čuva vrednost koja je zajednička za sve instance klase.

2.1.7. SEALED KLASA

Sealed klasa je klasa koja ne može biti nasleđena. Kada se klasa označi kao sealed, bilo koji pokušaj da se iz nje izvede druga klasa će rezultirati greškom. U ovakve klase se stavljaju poslednji čvorovi u hiherarhiji za koje ne želimo da budu prošireni drugim klasama.

```
public sealed class SealedClass
{
    int f1;

    public int F1
    {
        get { return f1; }
        set { f1 = value; }
    }

    public void SomeMethod()
    {
        Console.WriteLine("Some method");
    }
}
```

Listing 2.11 - Kod u okviru klase *SealedClass*

2.1.8. PARTIAL KLASA

Partial klase su klase koje mogu biti implementirane u više fajlova. Na ovaj način mogu biti implementirane i strukture, interfejsi ili metode. Svaki fajl sadrži deo definicije, a svi delovi se

kombiniju u trenutku kompajliranja aplikacije. Postoji nekoliko situacija kada je podela definicije u više fajlova poželjna:

- kada se radi na velikim projektima, razdvajanje klase u posebne fajlove omogućava da više programera radi istovremeno na istoj klasi
- kada se radi sa automatski generisanim kodom, dati kod se može proširiti u novom fajlu, bez potrebe da se menja automatski generisan kod. Visual Studio koristi ovaj pristup kada se radi sa Windows formama, Entity framework pristupom, itd.
- kada se kreiraju klase sa velikim brojem svojstava i metoda. Držanje svih metoda i svojstava u jednom fajlu može učiniti da on bude težak za održavanje i promenu.

Da bi se kreirale ovakve klase, koristi se ključna reč **partial**:

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

Listing 2.12 - Kod u okviru partial klase *Employee*

2.2. NASLEĐIVANJE

Nasleđivanje je veza između klasa koja podrazumeva preuzimanje sadržaja nekih klasa, odnosno klasa-predaka i na taj način, uz mogućnost modifikacije preuzetog sadržaja i dodavanje novog, dobija se klasa-potomak.

U C# programskom jeziku postoji samo jednostruko nasleđivanje. Jedna klasa može naslediti samo jednu klasu, ali više klasa može biti izvedeno iz jedne klase. Ako ništa ne napišemo, klasa nasleđuje klasu *Object*.

Klasa koja nasleđuje drugu klasu ima sve metode i atribute klase koju je nasledila, pri čemu može dodavati nove atribute i metode, ali može i redefinisati postojeće metode.

Zavisno od modifikatora pristupa, metode i atributi klase pretka su:

- **public** - vidljivi unutar metoda klase naslednica i mogu se pozivati nad objektima klase naslednica
- **protected** - vidljivi unutar metoda klase naslednica i ne mogu se pozivati nad objektima klase naslednica
- **private** - nisu vidljivi unutar metoda klase naslednica i ne mogu se pozivati nad objektima klase naslednica

U sledećem primeru se kreira klasa *Pravougaonik*. Ova klasa poseduje konstruktore, atribute, svojstva i javne metode za računanje obima i površine. Iz ove klase je primenom nasleđivanja izvedena klasa *Kvadrat*. Prilikom nasleđivanja se ne nasleđuju konstruktori, pa klasa *Kvadrat* mora da implementira svoje sopstvene konstruktore. Ostali elementi se nasleđuju. Na kraju se iz *main* metode kreiraju objekti za kreirane klase i nad njima se pozivaju metode kako bi se dobio željeni ispis.

```
class Pravougaonik
{
    public Pravougaonik()
```

```

    {
        StranicaA = 2;
        StranicaB = 3;
    }

    public Pravougaonik(double strA, double strB)
    {
        StranicaA = strA;
        StranicaB = strB;
    }

    double stranicaA;
    public double StranicaA
    {
        get { return stranicaA; }
        set
        {
            if (value < 0)
                value = 0;
            stranicaA = value;
        }
    }

    double stranicaB;
    public double StranicaB
    {
        get { return stranicaB; }
        set
        {
            if (value < 0)
                value = 0;
            stranicaB = value;
        }
    }

    public double GetO()
    {
        return 2 * stranicaA + 2 * stranicaB;
    }

    public double GetP()
    {
        return stranicaA * stranicaB;
    }
}

class Kvadrat : Pravougaonik
{
    public Kvadrat() : base(2, 2) { }
    public Kvadrat(double aa) : base(aa, aa) { }
}

class Program
{
    static void Main(string[] args)
    {
        Pravougaonik p = new Pravougaonik(3, 4);
        Console.WriteLine("Obim: {0}; Povrs: {1}", p.GetO(), p.GetP());

        Kvadrat k = new Kvadrat();
        Console.WriteLine("Obim: {0}; Povrs: {1}", k.GetO(), k.GetP());
    }
}

```

```

        Console.ReadKey();
    }
}

```

Listing 2.13 - Primer za nasleđivanje klasa

Nasleđivanje nam omogućava da ne moramo da kopiramo kod koji se već nalazi u jednoj klasi. Izvođenjem će sav kod koji se nalazio u *public* i *protected* delu preći u izvedenu klasu.

2.3. POLIMORFIZAM

Polimorfizam se često navodi kao treći osnovni gradivni element objektno-orjentisanog programiranja, posle enkapsulacije i nasleđivanja. Polimorfizam je grčka reč koja znači "više oblika". Polimorfizam ima dva osnovna aspekta:

- u vreme izvršavanja, objekti izvedene klase se mogu tretirati kao objekti osnovne klase na mestima kao što su parametri u pozivu metoda, kolekcije i nizovi. Kada se to desi, deklarirani tip objekta više nije identičan sa njegovim tipom u vreme izvršavanja

```

class Pravougaonik
{
    public Pravougaonik()
    {
        StranicaA = 2;
        StranicaB = 3;
    }

    public Pravougaonik(double strA, double strB)
    {
        StranicaA = strA;
        StranicaB = strB;
    }

    double stranicaA;
    public double StranicaA
    {
        get { return stranicaA; }
        set
        {
            if (value < 0)
                value = 0;
            stranicaA = value;
        }
    }

    double stranicaB;
    public double StranicaB
    {
        get { return stranicaB; }
        set
        {
            if (value < 0)
                value = 0;
            stranicaB = value;
        }
    }

    public double GetO()
    {
        return 2 * stranicaA + 2 * stranicaB;
    }

    public double GetP()

```



```

    {
        return stranicaA * stranicaB;
    }
}

class Kvadrat : Pravougaonik
{
    public Kvadrat() : base(2, 2) { }
    public Kvadrat(double aa) : base(aa, aa) { }
}

class Program
{
    static void Prikazi(Pravougaonik pp)
    {
        Console.WriteLine("Obim: {0}; Povrs: {1}", pp.GetO(), pp.GetP());
    }

    static void Main(string[] args)
    {
        Pravougaonik p = new Pravougaonik(3, 4);
        Prikazi(p);

        Kvadrat k = new Kvadrat(2);
        Prikazi(k);

        Console.ReadKey();
    }
}

```

Listing 2.14 - Primer polimorfizma

- osnovne klase mogu da definišu i implementiraju virtuelne metode, a izvedene klase mogu da ih redefinišu, što znači da one daju sopstvenu definiciju i implementaciju. U vreme izvršavanja, kada klijentski kod poziva metodu, CLR traži tip objekta u vreme izvršavanja i pokreće redefinisanu virtuelnu metodu. Zbog toga je moguće u kodu pozvati metodu nad osnovnom klasom, a izazvati izvršavanje metode u izvedenoj klasi.

```

public class Osoba
{
    protected string _ime;
    protected string _prezime;

    public Osoba(string ime, string prezime)
    {
        _ime = ime;
        _prezime = prezime;
    }

    public virtual void predstaviSe()
    {
        Console.WriteLine("Ja sam {0} {1}", _ime, _prezime);
    }
}

public class Student : Osoba
{
    protected int _brojIndeksa;

    public Student(string ime, string prezime, int brojIndeksa) :
        base(ime, prezime)
    {
        _brojIndeksa = brojIndeksa;
    }
}

```

```

    }

    public override void predstaviSe()
    {
        base.predstaviSe();
        Console.WriteLine("Ja sam student, moj indeks je:{0}", _brojIndeksa);
    }
}

class Program
{
    static void Predstavljanje(Osoba o)
    {
        o.predstaviSe();
    }

    static void Main(string[] args)
    {
        Osoba o = new Osoba("Petar", "Petrovic");
        Student s = new Student("Jovan", "Jovanovic", 1234);
        Predstavljanje(o);
        Predstavljanje(s);

        Console.ReadKey();
    }
}

```

Listing 2.15 - Primer polimorfizma sa virtuelnim metodama

2.4. ABSTRAKTNE KLASKE

Abstraktna klasa predstavlja specijalni tip klase koji se ne može instancirati. One se mogu naslediti i tu se nalazi njihova uloga u objektno-orjentisanom programiranju. One sadrže članove koje će klase, koje budu izvedene iz nje, implementirati. Abstraktne klase mogu sadržati abstraktne članove.

```

public abstract class Figura
{
    public abstract double GetO();
    public abstract double GetP();
}

```

Listing 2.16 - Kod u okviru abstraktne klase *Figura*

Metode *GetO* i *GetP* su označene kao abstraktne i moraju biti implementirane u klasama koje su izvedene iz klase *Figura*. Treba primetiti da metode *GetO* i *GetP* nemaju telo. Telo će biti kreirano u izvedenim klasama.

```

public class Pravougaonik : Figura
{
    public Pravougaonik()
    {
        StranicaA = 2;
        StranicaB = 3;
    }

    public Pravougaonik(double strA, double strB)
    {
        StranicaA = strA;
        StranicaB = strB;
    }

    double stranicaA;
    public double StranicaA
    {

```

```

        get { return stranicaA; }
        set { stranicaA = value; }
    }

    double stranicaB;
    public double StranicaB
    {
        get { return stranicaB; }
        set { stranicaB = value; }
    }

    public override double GetO()
    {
        return 2 * stranicaA + 2 * stranicaB;
    }

    public override double GetP()
    {
        return stranicaA * stranicaB;
    }
}

public class Kvadrat : Pravougaonik
{
    public Kvadrat() : base(1, 1) { }
    public Kvadrat(double aa) : base(aa, aa) { }
}

public class Krug : Figura
{
    public Krug()
    {
        Poluprecnik = 2;
    }

    public Krug(double p)
    {
        Poluprecnik = p;
    }

    double poluprecnik;
    public double Poluprecnik
    {
        get { return poluprecnik; }
        set { poluprecnik = value; }
    }

    public override double GetO()
    {
        return 2 * Poluprecnik * Math.PI;
    }

    public override double GetP()
    {
        return Poluprecnik * Poluprecnik * Math.PI;
    }
}

```

Listing 2.17 - Klase izvedene iz klase *Figura*

```

class Program
{
    static void Prikazi(Figura f)

```

```

    {
        Console.WriteLine("Obim: {0} ; Povrsina: {1}", f.GetO(), f.GetP());
    }

    static void Main(string[] args)
    {
        Pravougaonik p = new Pravougaonik(3, 4);
        Prikazi(p);

        Kvadrat k = new Kvadrat(2);
        Prikazi(k);

        Krug r = new Krug(3);
        Prikazi(r);

        Console.ReadKey();
    }
}

```

Listing 2.18 - Poziv klasa koje su izvedene iz klase *Figura* primenom polimorfizma

2.5. INTERFEJSI

Interfejsi sadrže samo prototipove metoda, svojstva i događaja. Interfejsi su kao ugovori. Ako klasa treba da ispuni određene mogućnosti, onda bi ona trebala da implementira odgovarajuće interfejse. Klasa koja implementira interfejs mora implementirati sve članove interfejsa koji su navedeni u njegovoj definiciji.

Druga upotreba interfejsa je da omoguće višestruko nasleđivanje, koje nije moguće u C# programskom jeziku. To znači da je moguće klasu izvesti iz jedne i samo jedne klase, ali je moguće implementirati veći broj interfejsa.

```

interface IVozilo
{
    double Brzina(double rastojanje, int sati);
}

interface ISTaza
{
    double Rastojanje(double brzina, int sati);
}

class Vozilo : IVozilo, ISTaza
{
    public double Brzina(double rastojanje, int sati)
    {
        double brzina = 0.0d;
        brzina = rastojanje / sati;
        return brzina;
    }

    public double Rastojanje(double brzina, int sati)
    {
        double rastojanje = 0.0d;
        rastojanje = brzina * sati;
        return rastojanje;
    }
}

class Program
{
    static void Main(string[] args)

```

```

{
    double rastojanje = 500;
    int sati = 2;
    double brzina = 120;
    Vozilo objVozilo = new Vozilo();
    Console.WriteLine("Brzina: {0}", objVozilo.Brzina(rastojanje, sati));
    Console.WriteLine("Rastojanje: {0}", objVozilo.Rastojanje(brzina, sati));
    Console.ReadKey();
}
}

```

Listing 2.19 - Primer upotrebe interfejsa

Preko interfejsa je moguće implementirati i polimorfizam:

```

interface IVozilo
{
    double Brzina();
}

class Automobil : IVozilo
{
    double _rastojanje;
    int _sati;

    public Automobil(double rastojanje, int sati)
    {
        _rastojanje = rastojanje;
        _sati = sati;
    }

    public double Brzina()
    {
        double brzina = 0.0d;
        brzina = _rastojanje / _sati;
        return brzina;
    }
}

class Kamion : IVozilo
{
    double _rastojanje;
    int _sati;

    public Kamion(double rastojanje, int sati)
    {
        _rastojanje = rastojanje;
        _sati = sati;
    }

    public double Brzina()
    {
        double brzina = 0.0d;
        // Jedan sat kamion pravi pauzu
        brzina = _rastojanje / (_sati - 1);
        return brzina;
    }
}

class Program
{
    static void Ispisi(IVozilo vozilo)
    {
        Console.WriteLine("Brzina: {0}", vozilo.Brzina());
    }
}

```

```

    }

    static void Main(string[] args)
    {
        Automobil automobil = new Automobil(300, 4);
        Kamion kamion = new Kamion(300, 7);

        Ispisi(automobil);
        Ispisi(kamion);

        Console.ReadKey();
    }
}

```

Listing 2.20 - Primer upotrebe interfejsa sa polimorfizmom

Važne razlike između interfejsa i abstraktnih klasa:

Interfejs	Abstraktna klasa
Interfejsi podržavaju višestruko nasleđivanje	Abstraktne klase ne podržavaju višestruko nasleđivanje
Interfejsi ne sadrže attribute	Abstraktne klase mogu da sadrže attribute
Interfejsi sadrže samo nepotpune članove (prototipove metoda)	Abstraktne klase sadrže i nepotpune (abstraktne) članove i potpune članove (implementirane metode)
Kod interfejsa se ne koriste modifikatori pristupa. Sve je public	Abstraktne klase mogu da sadrže modifikatore pristupa za attribute, metode
Član interfejsa ne može da bude static	Samo potpuni članovi abstraktnih klasa mogu da budu static
Kod interfejsa moraju da se implementiraju sve metode	Kod abstraktnih klasa moraju da se implementiraju samo nepotpune metode

Tabela 2.1 - Razlike između abstraktnih klasa i interfejsa

2.6. GENERIČKE KLASKE

Generičke klase su dodate u verziju 2.0 C# programskog jezika. One omogućavaju kreiranje klasa i metoda koje se razlikuju u specifikaciji jednog ili više tipova dok se klasa ili metoda ne deklariraju i instanciraju u okviru klijentskog koda. Pomoću njih opisujemo opšti slučaj bez upotrebe konkretnih tipova.

U sledećem primeru kreiramo klasu `MyGenericArray` koja predstavlja generički niz. Ovaj niz može da sadrži elemente bilo kog tipa. U okviru klase dodajemo konstruktor i metode `getItem` i `setItem`. Metoda `main` će tokom izvršavanja kreirati dva niza pri čemu jedan sadrži celobrojne vrednosti, a drugi karaktere.

```

public class MyGenericArray<T>
{
    T[] array;

    public MyGenericArray(int size)
    {
        array = new T[size + 1];
    }

    public T getItem(int index)
    {
        return array[index];
    }
}

```

```

    }

    public void SetItem(int index, T value)
    {
        array[index] = value;
    }
}

class Program
{
    static void Main(string[] args)
    {
        MyGenericArray<int> intArray = new MyGenericArray<int>(5);
        for (int i = 0; i < 5; i++)
        {
            intArray.SetItem(i, i * 5);
        }
        for (int i = 0; i < 5; i++)
        {
            Console.Write(intArray.GetItem(i) + " ");
        }

        Console.WriteLine();

        MyGenericArray<char> charArray = new MyGenericArray<char>(5);
        for (int i = 0; i < 5; i++)
        {
            charArray.SetItem(i, (char)(i + 97));
        }
        for (int i = 0; i < 5; i++)
        {
            Console.Write(charArray.GetItem(i) + " ");
        }

        Console.ReadKey();
    }
}

```

Listing 2.21 - Primer generičke klase *MyGenericArray*

Primer *Stack* generičke klase je dat u nastavku. To je kolekcija kod koje se metodom *Push* stavlja jedan element bilo kog tipa na vrh steka, a metodom *Pop* se sa vrha steka uzima jedan element. U okviru *main* metode je kreiran stek celobrojnih vrednosti.

```

public class Stack<T>
{
    int _size;
    int _stackPointer = 0;
    T[] _items;

    public Stack(int size)
    {
        _size = size;
        _items = new T[_size];
    }

    public void Push(T item)
    {
        if (_stackPointer >= _size)
            throw new StackOverflowException();
        _items[_stackPointer] = item;
        _stackPointer++;
    }
}

```

```

public T Pop()
{
    _stackPointer--;
    if (_stackPointer >= 0)
        return _items[_stackPointer];
    else
    {
        _stackPointer = 0;
        throw new InvalidOperationException(
            "Cannot pop an empty stack");
    }
}
}

class Program
{
    static void Main(string[] args)
    {
        Stack<int> intStack = new Stack<int>(10);
        for (int i = 0; i < 10; i++)
        {
            intStack.Push(i * 5);
        }
        for (int i = 0; i < 10; i++)
        {
            Console.Write(intStack.Pop() + " ");
        }
        Console.ReadKey();
    }
}

```

Listing 2.22 - Primer generičke klase *Stack*

3. ARHITEKTURA SISTEMA

Definisanje arhitekture softverske aplikacije predstavlja proces kreiranja softvera koji zadovoljava sve tehničke i funkcionalne zahteve, dok u isto vreme optimizuje kvalitativne osobine kao što su performanse, bezbednost i mogućnost jednostavnog održavanja. Softverska arhitektura uključuje niz odluka zasnovanih na mnogobrojnim faktorima, a svaka od ovih odluka može imati značajan uticaj na kvalitet, performanse, održavanje i opšti uspeh cele aplikacije.

Kao i svaka druga kompleksna struktura, softver mora biti kreiran na jakim osnovama. Kreiranje aplikacije može biti neuspešno ukoliko se ne uzmu u obzir ključni scenariji upotrebe i osnovni zahtevi koji se pred nju postavljaju. Moderni alati i platforme pojednostavljaju kreiranje složenih softverskih sistema, ali i dalje ne umanjuju potrebu za kreiranjem dobre softverske arhitekture. Rizici koji se pojavljuju kao posledica loše arhitekture uključuju softver koji je nestabilan, nije u mogućnosti da odgovori trenutnim ili budućim poslovnim zahtevima, ili je težak za postavljanje ili održavanje.

Arhitektura sistema predstavlja skelet datog sistema. Za arhitekturu sistema su odgovorne arhitektae. Njihov posao je da prikupe zahteve, dizajniraju ceo sistem, osiguraju da implementacija odgovara očekivanju i na kraju omoguće da korisnici dobiju ono što im je potrebno (što ne mora uvek da bude ono što su na početku tražili). Arhitektae takođe vrše komunikaciju sa timovima koji su zaduženi za razvoj sistema. Ova komunikacija se uglavnom vrši putem UML dijagrama. Primenom opštih principa softverskog inženjerstva, kao i principa objektno orijentisanog programiranja, arhitektae imaju za cilj da sistem podele u što je moguće manje celine.

Arhitektura softvera ima određene preduslove - principi u dizajniranju sistema, i određene postuslove - implementirani sistem koji daje očekivane rezultate. Ne postoji način da se proveri arhitektura sistema i da se odredi da li je ona odgovarajuća za postavljene zahteve. Da bi se ocenio dizajn sistema, sistem se mora testirati na različite načine i na različitim nivoima. Potrebno je primeniti testiranje komponenti kako bi se proverile pojedinačne funkcionalnosti, integraciono testiranje da bi se proverilo kako sistem funkcioniše u sprezi sa drugim sistemima i aplikacijama. Na kraju se vrši testiranje prihvatljivosti da bi se proverilo šta korisnici misle o aplikaciji i da li aplikacija pruža usluge za koje je kreirana.

Softverska arhitektura se fokusira na to kako se koriste najvažniji elementi i komponente u okviru aplikacije, ili kako oni komuniciraju i sarađuju sa ostalim elementima i komponentama u okviru aplikacije. Odabir struktura podataka i algoritama kao i načina implementacije pojedinih komponenti predstavlja oblast softverskog dizajna. Softverska arhitektura i softverski dizajn se veoma često preklapaju. Ove dve oblasti se zajedno kombinuju u cilju kreiranja što kvalitetnijeg softvera. Dizajn se koristi kako bi se realizovala softverska arhitektura.

Prilikom kreiranja softverske arhitekture, treba uzeti u obzir sledeće koncepte:

- Kako će korisnici koristiti aplikaciju?
- Kako će aplikacija biti postavljena na krajnje okruženje na kom će biti korišćena?
- Koji su zahtevi sa stanovišta sigurnosti, performansi, konkurentnosti, internacionalizacije i konfiguracije?
- Kako dizajnirati aplikaciju da bude fleksibilna i laka za održavanje tokom njenog životnog veka?

3.1. CILJEVI ARHITEKTURE

Softverska arhitektura ima za cilj da kreira vezu između poslovnih zahteva i tehničkih zahteva razumevanjem slučajeva korišćenja, kao i pronalaženjem načina da se ti slučajevi korišćenja implementiraju u softver. Cilj arhitekture je da identifikuje zahteve koji utiču na strukturu aplikacije. Dobra arhitektura smanjuje poslovne rizike povezane sa kreiranjem tehničkog rešenja. Dobar dizajn je dovoljno fleksibilan da izdrži prirodne promene u softverskoj i hardverskoj tehnologiji, kao i u

korisničkim zahtevima, koji će se dešavati tokom životnog veka aplikacije. Arhitektura mora uzeti u obzir ukupne efekte odluka o softverskom dizajnu, kompromis između kvalitativnih osobina (npr. performanse i sigurnost), kao i kompromise kako bi se zadovoljili zahtevi korisnika, sistema i poslovnih zahteva.

Softverska arhitektura bi trebala da:

- Prikaže strukturu sistema ali da sakrije detalje implementacije
- Realizuje sve slučajeve korišćenja i scenarije
- Odgovori i na funkcionalne i na kvalitativne zahteve

Važno je razumeti korisničke zahteve, kao i poslovne zahteve za bržim odzivom, boljom podrškom u cilju izmene tokova rada, kao i lakšim izmenama. Ovo su faktori koji utiču na softversku arhitekturu, i koji će je oblikovati u budućnosti tokom životnog veka softvera.

Potrebno je razumeti sledeće zahteve:

- Zadovoljstvo korisnika - dizajn koji je u skladu sa zadovoljstvom korisnika mora da bude fleksibilan i da se može prilagoditi potrebama i željama korisnika. Aplikaciju treba kreirati tako da je korisnik može prilagoditi sebi. Treba mu omogućiti da on sam definiše kako želi da vrši interakciju sa sistemom, umesto da mu se to nameće. Ovde treba voditi računa da se ne pretera sa nepotrebnim opcijama i podešavanjima koja mogu dovesti do konfuzije. Treba dizajnirati aplikaciju tako da budu jednostavne i da je lako pronaći informaciju i koristiti aplikaciju.
- Treba iskorisiti već postojeće platforme i tehnologije. Koristiti framework-e visokog nivoa, gde to ima smisla, kako bi se mogli fokusirati na funkcionalne zahteve aplikacije, a ne da se ponovo kreira nešto što već postoji. Treba koristiti paterne koji predstavljaju izvor proverenih rešenja za uobičajene probleme.
- Fleksibilan dizajn - koristi prednosti slabog povezivanja kako bi se isti kod mogao koristiti na više mesta i kako bi se olakšalo održavanje. Mogu se koristiti prednosti servisno orjentisanih tehnologija kao što je SOA kako bi se obezbedila saradnja sa drugim sistemima.
- Kada se kreira aplikacija, treba razmišljati o budućim trendovima koji se mogu očekivati u dizajnu nakon postavljanja aplikacije. Npr. treba uzeti u obzir mogućnost korišćenja bogatijih UI alta, povećanje mrežnog protoka i dostupnosti, moguću upotrebu mobilnih uređaja, korišćenje jačeg hardvera, prelazak na cloud, itd.

3.2. PRINCIPI SOFTVERSKJE ARHITEKTURE

Prilikom kreiranja arhitekture moramo pretpostaviti da će dizajn evoluirati tokom vremena i da ne možemo znati sve što je potrebno unapred kako bi u potpunosti kreirali arhitekturu sistema. Dizajn u opštem slučaju mora da se menja tokom implementiranja aplikacije, kako se aplikacija testira u odnosu na stvarne zahteve. Treba kreirati arhitekturu sa tim na umu, kako bi mogla da se prilagodi zahtevima koji nisu u potpunosti poznati na početku procesa dizajniranja.

Treba razmatrati sledeća pitanja kada se kreira arhitektura aplikacije:

- Koji su osnovni delovi arhitekture koji predstavljaju najveći rizik ukoliko se ne predvide dovoljno dobro?
- Koji su delovi arhitekture koji će se najverovatnije menjati, ili čiji se dizajn može odložiti a da to nema velike posledice na celokupan proces izrade aplikacije?
- Koje su pretpostavke i kako ih testirati i proveriti?
- Koji uslovi mogu dovesti do izmene dizajna?

Ne treba pokušavati da se predvidi bukvalno sve što uključuje arhitektura novog softvera, kao što i ne treba praviti pretpostavke koje se ne mogu proveriti. Umesto toga, treba kreirati sistem koji je otvoren za nove promene. Postoje delovi u okviru dizajna koji se moraju ispraviti u ranim fazama, jer njihov redizajn sa sobom povlači visoku cenu. Ove oblasti treba dobro istražiti i njima posvetiti dodatno vreme kako bi se valjano kreirale.

Treba razmotriti sledeće ključne principe kada se kreira arhitektura aplikacije:

- **Kreiraj da menjaš umesto da kreiraš da traje** - treba razmotriti kako će se aplikacija vremenom menjati kako bi se što bezbolnije moglo odgovoriti na zahtevane promene.
- **Modeluj kako bi analizirao i smanjio rizik** - treba koristiti alate za modelovanje kao što je UML (Unified Modeling Language), kako bi se bolje sagledali zahtevi, arhitektura i dizajn, i kako bi se analizirao njihov uticaj. Međutim, ne treba modelovati do nivoa koji smanjuje mogućnost da se dizajn lako prilagodi novim zahtevima.
- **Koristi modele i vizuelizaciju kao alate za komunikaciju i saradnju** - efikasna komunikacija, kreiranje odluka i predviđanje izmena predstavljaju ključne faktore u cilju kreiranja dobre arhitekture. Treba koristiti modele, poglede i druga sredstva vizuelizacije kako bi se efikasno komuniciralo sa svim činiocima koji učestvuju u kreiranju softvera.
- **Identifikovanje ključnih inženjerskih odluka** - treba koristiti sve dostupne informacije i znanja kako bi se razumele najčešće inženjerske odluke i uočile oblasti gde se greške najčešće prave. Treba investirati kako bi se na samom početku donele ispravne odluke i kako bi dizajn bio više fleksibilan i otporan na izmene.

Treba koristiti inkrementalan i iterativan pristup kako bi se arhitektura činila boljom. Treba krenuti sa opštom arhitekturom kako bi se kreirala opšta slika, a zatim razvijati samu arhitekturu kroz testiranje i poboljšavanje u skladu sa zahtevima. Ne treba pokušavati da se sve uradi ispravno iz prvog pokušaja. Treba kreirati dizajn taman toliko koliko je neophodno da bi se on mogu testirati u odnosu na zahteve i pretpostavke. Iterativno treba dodavati detalje u dizajn kroz više prolazaka kako bi se obezbedili da smo velike odluke doneli ispravno, a nakon toga se možemo fokusirati na detalje. Uobičajena greška je da se fokusiramo na detalje suviše brzo i da kreiramo opštu sliku pogrešno, koristeći se netačnim i neproverenim pretpostavkama. Kada testiramo arhitekturu, treba razmotriti sledeća pitanja:

- Koje pretpostavke su učinjene u okviru arhitekture?
- Koje eksplicitne ili izvedene zahteve ova arhitektura zadovoljava?
- Koji su ključni rizici u okviru kreirane arhitekture i korišćenog pristupa?
- Koje kontramere su primenjene kako bi se ublažili ključni rizici?
- Na koje načine je nova arhitektura poboljšanje u odnosu na prethodnu arhitekturu?

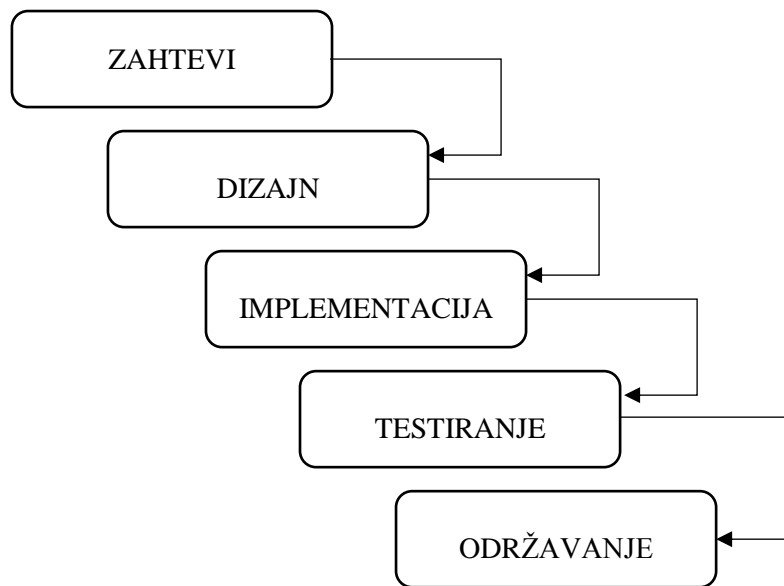
3.3. MODELI U RAZVOJU SOFTVERA

Pre početka rada na softverskom projektu, potrebno je odabrati metodologiju koja odgovara datom projektu i koja je kompatibilna sa znanjem i sposobnostima ljudi uključenih na projektu. Metodologija je skup preporučene prakse koja se primenjuje u procesu razvoja softvera.

Postoje dva osnovna modela razvoja softvera: tradicionalna metodologija i agilna metodologija.

3.3.1. TRADICIONALNA METODOLOGIJA

Najpoznatija i najstarija metodologija je model vodopada. To je model u kom razvoj softvera produžava iz jedne faze u drugu u čisto sekvencijalnom smislu. U korak N+1 se premeštamo tek kada je korak N 100% uspešno završen. Slika 3.1 prikazuje šemu modela vodopada.



Slika 3.1 - Model vodopada

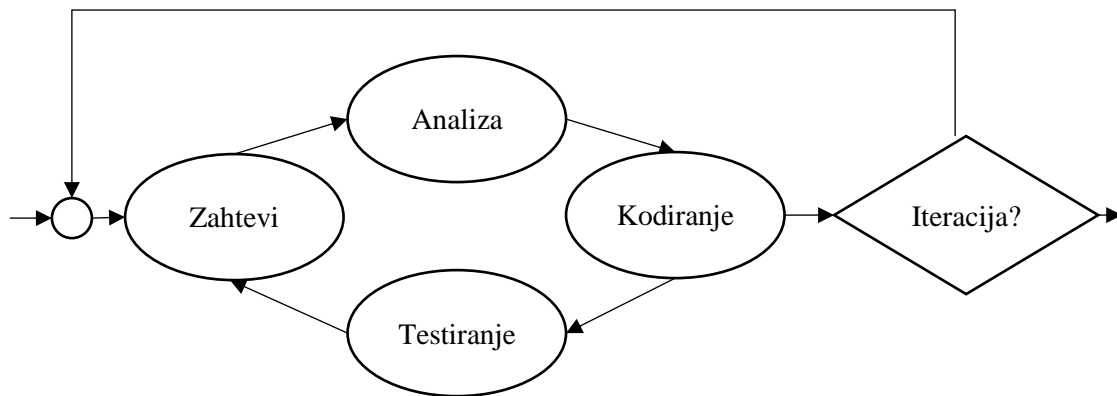
Nakon što je tim završio sa analizom zahteva, on kreće sa kreiranjem arhitekture sistema. Nakon toga se kreće sa kodiranjem, zatim testiranjem i na kraju održavanjem.

Model vodopada je jednostavan i veoma disciplinovan model, ali se može koristiti samo u jednostavnim projektima. Ovo je zbog toga što svi zahtevi skoro nikad nisu poznati pre nego što krenemo sa dizajniranjem sistema. Zbog toga moramo krenuti na sledeći korak dok prethodni još uvek nije kompletiran.

Iz ovog razloga su razvijene različite varijacije modela vodopada u kojima se faze dizajna i implementacije u izvesnoj meri preklapaju. Sve ove metodologije dele nekoliko zajedničkih atributa: broj faza kroz koje prolaze, broj iteracija koji je potreban da bi se kreirao softver, kao i tipično trajanje jedne iteracije. Sve faze se izvršavaju sekvencijalno, i uvek postoji bar jedna iteracija koja završava sa kreiranjem softvera. Razlike između metodologija su u redosledu po kojem se ulazi u određene faze, broju zahtevanih iteracija i trajanju svake iteracije.

3.3.2. AGILNA METODOLOGIJA

Iterativni razvoj je ciklični proces koji je razvijen kao odgovor na metod vodopada, a koji naglašava inkrementalno kreiranje softvera. Nakon početnih aktivnosti, projekat prolazi kroz niz iteracija koje uključuju analizu, kodiranje i testiranje. Svaka iteracija daje isporučivu, ali ipak nekompletnu verziju sistema. U svakoj iteraciji, tim unosi promene u dizajnu i dodaje nove funkcionalnosti sve dok se ne ispune svi zahtevi. Slika 3.2 daje grafički prikaz iterativnog modela.



Slika 3.2 - Iterativni model

Iterativni razvoj predstavlja osnovu za agilne metodologije. Agilne metodologije postavljaju pojedince u centar dešavanja. Njihov fokus je usresređen na ljude koji zajednički rade na kreiranju softvera i na njihovoj komunikaciji, a ne na kreiranju softvera. Promene i drugačije formulisanje su ključni u agilnim metodologijama. Povratne informacije od korisnika se vrednuju više od planiranja. Jedan od osnovnih principa agilnih metodologija glasi: "Softver koji radi je osnovna mera napretka".

Da bi uvideli razliku koju donosi agilna metodologija možemo posmatrati sledeći primer. Projekat kreće sa realizacijom pri čemu je poznato samo nekoliko zahteva. Sa sigurnošću se može reći da će još dosta zahteva stići od datog momenta pa do kraja realizacije. Sa agilnim načinom razmišljanja, to ne predstavlja problem. Uzima se podskup postojećih zahteva koji se mogu implementirati u jednoj iteraciji. U sledećem koraku se ulazi u prvu iteraciju. Tokom date iteracije se fokusira na jedan po jedan zahtev koji se zatim implementira. Na kraju iteracije je kreiran softver koji radi. On možda nije kompletan, ali je važno da radi.

Nakon ovoga se ulazi u sledeću iteraciju koja se fokusira na drugi skup zahteva. Ukoliko se u međuvremenu nešto promenilo ili se sa sigurnošću može reći da nešto nije korektno, vrše se izmene. Proces se nastavlja sve dok se ne dođe do stadijuma da više nema šta da se doda.

U ovakvom pristupu klijenti i programeri svakodnevno rade zajedno. Povratne informacije se redovno skupljaju pomoću klijenata, a i promene u kodu koje urade programeri su odmah vidljive klijentima. Arhitektae su uključene u razvoj sistema kao programeri, a ceo tim je visoko obučan i motivisan. Dužina trajanja iteracije se meri u nedeljama. Drugim rečima, agilne metodologije su spremne da brzo reaguju na promene.

Termin "agilne" metodologije je opšti termin. Postoji više metodologija koje spadaju pod ovaj pojam, a najpopularnija je ekstremno programiranje (Extreme Programming - XP). U ekstremnom programiranju faze se izvode u veoma kratkim intervalima koje ne traju duže od dve nedelje. Kodiranje i dizajn se izvršavaju paralelno jedno sa drugim.

4. DIZAJN PRINCIPI I PATERNI

Jedna stvar je napisati kod koji radi. Nešto sasvim drugo je napisati *dobar* kod koji radi. Usvajanje stanovišta "pisanje dobrog koda koji radi" proističe iz sposobnosti da se sistem posmatra iz široke perspektive. Na kraju, vrhunski sistem ne nastaje samo kao proizvod pisanja instrukcija i naredbi koje mu omogućavaju da se izvršava. Tu postoji mnogo više aspekata koji se na kraju, direktno ili indirektno, svode na dizajn.

Stav "pisanje dobrog koda koji radi" vodi ka vrednovanju mogućnosti lakog održavanja koda. Ovo se može usvojiti ne zato što su drugi aspekti manje važni u odnosu na održavanje, već zbog toga što je održavanje dosta skupo i može biti veoma frustrirajuće za programere koji su uključeni u njega. Osnovni kod koji se može lako pretraživati u cilju pronalaženja grešaka, i kome ispravljanje grešaka ne predstavlja problem, pogodan je za bilo kakve vrste poboljšanja uključujući i proširivost. Zbog toga, mogućnost lakog održavanja predstavlja karakteristiku kvaliteta kojoj je potrebno dati najviši prioritet u procesu dizajniranja sistema.

Održavanje postaje skupo kada kreirani softver ne zadovoljava zahteve, kada nije dovoljno testiran, ili kada oba uslova nisu ispunjena. Strukturni pristup dizajnu predstavlja atribut koji omogućava da se softver lakše održava i razvija. On se primenjuje kroz odgovarajuće tehnike kodovanja. Čitljivost koda je još jedna fundamentalna osobina koja se najbolje postiže u situacijama kada se kod kombinuje sa odgovarajućom količinom dokumentacije i sistemom koji automatski prati promene.

Nezadovoljavajući softver je uglavnom posledica lošeg dizajna. Loš dizajn obično nastaje kao posledica dva uzroka koja nisu međusobno isključiva: nedovoljno znanje arhitekta i neprecizni ili kontradiktorni zahtevi. Kontradiktorni zahtevi uglavnom nastaju kao posledica loše komunikacije.

Ispravljanje problema u komunikaciji vodi direktno ka agilnim metodologijama. Glavna prednost ovih metodologija dolazi iz konstantne komunikacije koju metodologije promovišu, a koje se dešavaju između ljudi u timu kao i između tima i klijenata. Šta god da je pogrešno shvaćeno ili protumačeno u prvoj iteraciji, biće ispravljeno u sledećoj iteraciji (ili nekoj od sledećih iteracija). Razlog ovome je komunikacija koja je neophodna da bi se išlo ka rešenju. Ispravljanje pogrešno protumačenih zahteva se dešava u ranoj fazi kreiranja procesa.

Posao softverskih inženjera je pre svega rešavanje softverskih problema. To su problemi koje su drugi inženjeri verovatno već rešavali veliki broj puta u različitim oblicima. Tokom kreiranja softvera primenom objektno-orijentisanog programiranja, veliki broj šablona (pattern), principa i slučajeva najbolje prakse je otkriven, imenovan i unet u katalog. Poznavanjem ovih šablona i uobičajenih rešenja, inženjeri mogu da „razbijaju“ složene probleme i razviju aplikacije na jedinstven način sa oporbnim i pouzdanim rešenjima.

Dizajn šablona (design patterns) predstavljaju apstraktne primere rešenja na visokom nivou. Njih treba posmatrati kao plan u rešavanju problema, a ne kao samo rešenje. Gotovo je nemoguće pronaći okvir (framework) koji će biti primenjen kako bi se kreirala cela aplikacija. Umesto toga, inženjeri vrše generalizaciju (uopštavanje) problema kako bi prepoznali paterne koje treba da primene. Dizajn paterni imaju za cilj ponovnu upotrebu postojećih rešenja. Iako nisu svi problemi jednaki, ako je moguće posmatrati problem „razbiti“ i naći sličnosti sa problemima koji su ranije bili rešavani, onda je moguće primeniti uniformno rešenje nad njim. Većina problema na koje se nailazi tokom programiranja je već rešena nebrojeno puta, pa verovatno postoji i patern koji može pomoći u implementaciji rešenja.

Paterni su nastali kao rezultat dobre prakse i iskustva programera. Skup najvažnijih i najčešće korišćenih paterni je skupljen i objavljen u knjizi Design Patterns: Elements of Reusable Object-Oriented Software koja je poznata i pod nazivom Design Patterns Bible. Knjigu su napisali Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides koji su u literaturi i uopšte na satovima sa informatičkom tematikom poznati kao Gang of Four.

Oni su skupili 23 dizajn paterni i organizovali ih u tri grupe:

- **Creational Patterns:** odnose se na kreiranje objekata i referenciranje
- **Structural Patterns:** odnose se na relacije između objekata i na to kako objekti međusobno deluju jedan na drugi u cilju kreiranja većih i kompleksnijih objekata
- **Behavioral Patterns:** odnose se na komunikaciju između objekata, naročito u smislu odgovornosti i algoritama

Paterni su ključni za dizajn i razvoj softvera. Oni omogućavaju izražavanje namera kroz zajednički rečnik, kada se problemi rešavaju prilikom dizajna, tako i tokom kreiranja samog koda. Paterni promovišu upotrebu dobrog dizajna objektno-orjentisanog softvera. Oni predstavljaju efikasan način da se opiše rešenje kompleksnih problema. Sa poznavanjem dizajn paterna, moguća je brza komunikacija unutar tima bez obraćanja pažnje na detalje implementacije niskog nivoa. Njihova posebna vrednost se nalazi u činjenici da su to oprobana i testirana rešenja.

Paterni su nezavisni od korišćenog programskog jezika. Njihova primena je identična u svim objektno-orjentisanim programskim jezicima.

Međutim, ne zahtevaju svi problemi primenu dizajn paterna. Tačno je da dizajn paterni mogu učiniti da kompleksni problemi postanu jednostavni, ali oni takođe mogu jednostavne probleme učiniti kompleksnim. Nakon prvog upoznavanja sa dizajn paternima, mnogi inženjeri updaju u problem da pokušavaju da primene paterne na svaki deo koda. Ovim se postiže suprotan efekat od željenog, odnosno, sam softver se dodatno komplikuje. Bolji način upotrebe paterna je da se identifikuju osnovni problemi koje je potrebno rešiti i da se pronađu rešenja koja im odgovaraju.

4.1. UOBIČAJENI DIZAJN PRINCIPI

Postoji veliki broj uobičajenih dizajn principa koji, kao i dizajn paterni, predstavljaju najbolju praksu i omogućavaju osnovu na kojoj se mogu kreirati profesionalni softveri koji su laki za održavanje. Neki od najpoznatijih principa su:

- **Keep It Simple Stupid (KISS)** – veoma često softverska rešenja budu suviše komplikovana. Cilj KISS principa je da kod bude jednostavan i da se izbegne nepotrebna kompleksnost
- **Don't Repeat Yourself (DRY)** – ovaj princip nalaže da se izbegne bilo kakvo ponavljanje delova sistema. Ovo se postiže apstrakcijom onih delova koji su zajednički i njihovim smeštanjem na jednu lokaciju. Ovaj princip se ne bavi samo kodom, već bilo kojom logikom koja je duplirana u sistemu.
- **Tell, Don't Ask** – ovaj princip je blisko povezan sa enkapsulacijom i dodelom odgovornosti odgovarajućim klasama. On govori da je potrebno reći objektima koja akcija treba da se izvrši, umesto da se postavlja pitanje o stanju objekta i da se onda pravi odluka koja akcija treba da se izvrši. Ovo pomaže da se uredi odgovornost i da se izbegne jaka veza između klasa.
- **You Ain't Gonna Need It (YAGNI)** – ovaj princip govori da je potrebno uključiti samo funkcionalnosti koje su neophodne aplikaciji, a izbaciti sve ostaje funkcionalnosti za koje se misli da bi mogle zatrebati.
- **Separation of Concerns (SoC)** – predstavlja proces razdvajanja dela softvera na zasebne karakteristike koje sadrže jedinstveno ponašanje, kao i na podatke koje mogu koristiti i druge klase. Proces razdvajanja programa po diskretnim odgovornostima značajno pospešuje ponovnu upotrebu koda, održavanje i testiranje.

4.2. S.O.L.I.D. DIZAJN PRINCIPI

S.O.L.I.D. dizajn principi predstavljaju kolekciju najbolje prakse za objektno-orjentisan dizajn. Svi dizajn paterni koje su zapisali *Gang of Four* odgovaraju u određenoj formi ovim principima. Naziv S.O.L.I.D. dolazi iz prvih slova svakog principa:

- **Single Responsibility Principle (SRP)**

- Open-Close Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

4.2.1. SINGLE RESPONSIBILITY PRINCIP

Princip SRP (Single responsibility principle - princip o samo jednoj odgovornosti) je u bliskoj vezi sa SoC (Separation of Concerns - razdvajanje odgovornosti). On govori da svaki objekat treba da ima jedan fokus. Pridržavanjem ovog principa, izbegava se problem monolitnih klasa, što predstavlja softverski ekvivalent za švajcarski nož. Ako su objekti koncizni, povećava se čitljivost i omogućava lakše održavanje sistema.

Kao primer možemo posmatrati klasu *Book* koja obuhvata koncept knjige i pridruženih funkcionalnosti. Kod za ovu klasu je prikazan u listingu 4.1.

```
class Book
{
    public string getTitle()
    {
        return "A Great Book";
    }

    public string getAuthor()
    {
        return "John Doe";
    }

    public string turnPage()
    {
        // pointer to next page
    }

    public void printCurrentPage()
    {
        Console.WriteLine("current page content");
    }
}
```

Listing 4.1 - Kod u okviru klase *Book.cs*

Ova klasa se može učiniti kao sasvim ispravna. Imamo knjigu za koju možemo da odredimo naslov, autora i možemo da pređemo na sledeću stranicu. Poslednja metoda omogućava da se odštampa trenutna stranica i da se prikaže na ekranu. Međutim, ovde se može pojaviti problem. Mešanje logike sa prikazom je loša praksa. Zbog toga bi kod trebalo izmeniti kao što je prikazano u listingu 4.2.

```
class Book
{
    public string getTitle()
    {
        return "A Great Book";
    }

    public string getAuthor()
    {
        return "John Doe";
    }

    public string turnPage()
```



```

    {
        // pointer to next page
    }
}

interface Printer
{
    void printPage(string page);
}

class PlainTextPrinter : Printer
{
    public void printPage(string page)
    {
        Console.WriteLine("current page content: " + page);
    }
}

class HtmlPrinter : Printer
{
    public void printPage(string page)
    {
        Console.WriteLine("<div style='single-page'>" + page + "</div>");
    }
}

```

Listing 4.2 - Primer SRP principa

U okviru ovog jednostavnog primera, može se videti razdvajanje prezentacione logike od poslovne logike, što predstavlja SRP princip. Primenom ovog principa dobijamo veliku prednost u fleksibilnosti našeg softvera.

4.2.2. OPEN-CLOSE PRINCIP

Princip OCP (Open-close principle - otvoren-zatvoren princip) govori da klasa treba da bude otvorena za proširenja a zatvorena za izmene. Drugim rečima, treba omogućiti dodavanje novih karakteristika i proširenje klase bez promene unutrašnjeg ponašanja postojećih metoda. Princip teži da se izbegne menjanje postojeće klase i drugih klasa koje od nje zavise, jer će to dovesti do pojavljivanja velikog broja grešaka u samoj aplikaciji.

U okviru primera možemo posmatrati klasu *Customer* koja poseduje jednu metodu *getDiscount()*. Primer klase je prikazan u okviru listinga 4.3.

```

class Customer
{
    public virtual double getDiscount(double TotalSales)
    {
        return TotalSales;
    }
}

```

Listing 4.3 - Kod u okviru klase *Customer*

Ako se vremenom pored klijenata pojave i klijenti posebnog tipa (npr. silver i gold), onda se klasa *Customer* može promeniti kao što je prikazano u listingu 4.4.

```

class Customer
{
    private int _custType;

    public int CustType
    {
        get { return _custType; }
        set { _custType = value; }
    }
}

```

```

    }

    public double getDiscount(double TotalSales)
    {
        if (_custType == 1)
        {
            return TotalSales - 100;
        }
        else
        {
            return TotalSales - 50;
        }
    }
}

```

Listing 4.4 - Kod u okviru klase *Customer*

Problem je što, ako budemo imali još tipova klijenata, moraćemo dodavati još IF uslova u okviru *getDiscount* metode, što će dovesti do nove promene u okviru *Customer* klase. Kada se god promeni klasa, mora se osigurati da prethodni kod radi, jer izmene mogu dovesti do pojave grešaka u ostatku koda. Umesto toga, kod treba proširiti kako bi bili sigurni da postojeći kod i dalje radi. Prošireni kod je prikazan u okviru listinga 4.5.

```

class Customer
{
    public virtual double getDiscount(double TotalSales)
    {
        return TotalSales;
    }
}

class SilverCustomer : Customer
{
    public override double getDiscount(double TotalSales)
    {
        return base.getDiscount(TotalSales) - 50;
    }
}

class GoldCustomer : SilverCustomer
{
    public override double getDiscount(double TotalSales)
    {
        return base.getDiscount(TotalSales) - 100;
    }
}

```

Listing 4.5 - Proširenje funkcionalnosti klase *Customer*

Ovim kodom smo postigli da je klasa *Customer* zatvorena za bilo kakve izmene ali je otvorena za proširenja, što je suština OCP-a.

4.2.3. LISKOV SUBSTITUTION PRINCIP

Princip LSP (Liskov substitution principle - Liskov princip zamene) govori da bi trebalo omogućiti korišćenje bilo koje izvedene klase na mestu klase roditelja i da bi ta klasa trebala da se ponaša na isti način bez izmena. Ovaj princip je u skladu sa OCP principom jer osigurava da izvedena klasa ne utiče na ponašanje klase roditelja.

U okviru listinga 4.6 dat je primer LSP-a kod koga imamo tri klase: *Vehicle*, *Car* i *ElectricBus*. Klase *Car* i *ElectricBus* su nasleđene iz klase *Vehicle*.

```

class Vehicle
{

```

```

    public virtual void startEngine()
    {
        Console.WriteLine("Start engine");
    }

    public virtual void accelerate()
    {
        Console.WriteLine("Accelerate");
    }
}

class Car : Vehicle
{
    public override void startEngine()
    {
        engageIgnition();
        base.startEngine();
    }

    private void engageIgnition()
    {
        Console.WriteLine("Ignition procedure");
    }
}

class ElectricBus : Vehicle
{
    public override void accelerate()
    {
        increaseVoltage();
        connectIndividualEngines();
        base.accelerate();
    }

    private void increaseVoltage()
    {
        Console.WriteLine("Electric logic");
    }

    private void connectIndividualEngines()
    {
        Console.WriteLine("Connection logic");
    }
}

```

Listing 4.6 - Klase koje su kreirane u skladu sa LSP

Da bi demonstrirali upotrebu ovih klasa i da bi pokazali da se izvedene klase mogu upotrebiti na istom mestu gde i klasa roditelj, možemo kreirati klasu *Program* sa *main* metodom, kao što je prikazano u listingu 4.7.

```

class Program
{
    static void go(Vehicle v)
    {
        v.startEngine();
        v.accelerate();
    }
}

```

```

static void Main(string[] args)
{
    Vehicle v = new Vehicle();
    Vehicle c = new Car();
    Vehicle e = new ElectricBus();

    go(v);
    go(c);
    go(e);
}
}

```

Listing 4.7 - Kod u okviru klase *Program.cs*

4.2.4. INTERFACE SEGREGATION PRINCIP

Cilj ovog principa je dodeljivanje novog interfejsa grupama metoda koje imaju isti fokus kako bi se izbeglo da klijent mora da implementira jedan veliki interfejs i veliki broj metoda koje mu nisu potrebne. Prednost ovog principa se ogleda u tome da klase koje žele da koriste iste interfejse, treba da implementiraju samo određen skup metoda.

Da bi se demonstrirao Interface segregation princip, biće kreirana mala aplikacija čiji je domen katalog proizvoda. Katalog proizvoda čine filmovi u obliku DVD ili Blu-Ray diskova. Za svaki podtip proizvoda postoji odgovarajuća klasa. Obe klase implementiraju *IProduct* interfejs kao što se može videti u listingu 4.8.

```

public interface IProduct
{
    decimal Price { get; set; }
    int WeightInKg { get; set; }
    int RunningTime { get; set; }
}

public class DVD : IProduct
{
    public decimal Price { get; set; }
    public int WeightInKg { get; set; }
    public int RunningTime { get; set; }
}

public class BluRayDisc : IProduct
{
    public decimal Price { get; set; }
    public int WeightInKg { get; set; }
    public int RunningTime { get; set; }
}

```

Listing 4.8 - Klase *DVD* i *BluRayDisc* u okviru ISP

Sada ćemo u aplikaciju dodati novi tip proizvoda koji nije film. Dodajemo klasu *TShirt*. Pošto je i ona proizvod, mora da implementira *IProduct* interfejs. Problem sa tim što klasa *TShirt* implementira *IProduct* interfejs je u tome što properti *RunningTime* nema nikavo značenje za majicu i ne bi trebao tu da se nalazi. Rešenje je u uočavanju razlika između proizvoda kao što su filmovi i majice i prebacivanje tih razlika u specifične interfejse.

Dodaćemo novi interfejs *IMovie*. Izmenjeni kod aplikacije je prikazan u listingu 4.9.

```

public interface IMovie
{
    int RunningTime { get; set; }
}

public interface IProduct
{

```

```

    decimal Price { get; set; }
    int WeightInKg { get; set; }
}

public class DVD : IProduct, IMovie
{
    public decimal Price { get; set; }
    public int WeightInKg { get; set; }
    public int RunningTime { get; set; }
}

public class BluRayDisc : IProduct, IMovie
{
    public decimal Price { get; set; }
    public int WeightInKg { get; set; }
    public int RunningTime { get; set; }
}

public class TShirt : IProduct
{
    public decimal Price { get; set; }
    public int WeightInKg { get; set; }
}

```

Listing 4.9 - Izmenjeni kod koji demonstrira ISP

Ovo je suština Interface segregation principa. Razdvajanjem interfejsa, povećava se mogućnost ponovne upotrebe i razumevanja koda.

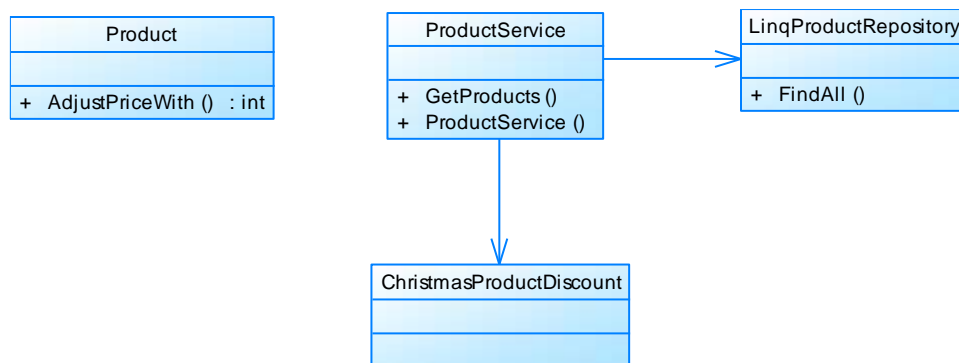
4.2.5. DEPENDENCY INVERSION PRINCIP

Princip DIP se bazira na izolovanju klasa od konkretne implementacije kako bi zavisile od apstraktnih klasa i interfejsa. On promoviše kodiranje bazirano na interfejsima, a ne na implementaciji. Ovo pospešuje fleksibilnost u okviru sistemam, osiguravajući da kod nije čvrsto vezan za jednu implementaciju.

DI predstavlja dobavljanje klase nižeg nivoa i zavisne klase kroz konstruktor, metodu ili properti. Zavisne klase se mogu zameniti interfejsima ili apstraktnim klasama koje će voditi ka slabo povezanim sistemima koji su vrlo dobri za testiranje i laki za izmenu.

Dependency Inversion princip (DIP) pomaže u razdvajanju koda tako što osigurava da unutar koda imamo zavisnosti od apstrakcija, a ne od konkretnih implementacija. Ovaj princip je najvažniji u cilju razumevanja dizajn paterna. Dependency Injection (DI) predstavlja implementaciju DIP. Ovi termini se često prepliću a cilj im je da se postigne razdvajanje koda.

U ovom primeru će se uraditi refaktorisanje koda kako bi se uveo DI princip i kako bi se u potpunosti razdvojila *ProductService* klasa od njenih zavisnih klasa. Primer je baziran na domenu kataloga proizvoda. Klasa *ProductService* zahteva od repository sloja da dobavi skup proizvoda i strategiju popusta, kako bi primenila popust nad svakim proizvodom pre nego što skup proizvoda vrati klijentu. Klase uključene u ovaj primer su prikazane na slici 4.1.



Slika 4.1 - Klase uključene u domen kataloga proizvoda

Klasa *Product* predstavlja proizvod koji se nalazi u katalogu i ima jednu metodu *AdjustPriceWith* koja prima kao parametar objekat *ChristmasProductDiscount*. Klasa *LinqProductRepository* predstavlja repozitori klasu čija je odgovornost dobavljanje podataka iz skladišta. *ChristmasProductDiscount* predstavlja tip popusta koji će biti primenjen nad proizvodima. Ova klasa ne sadrži kod i služi samo da bi se pokazali DI principi. Na kraju, klasa *ProductService* je odgovorna za dobavljanje kolekcije proizvoda iz repozitorija nad kojima će se zatim primeniti odgovarajući popust pre nego što se kolekcija prosledi kodu koji ju je tražio.

Primer koda pre primene DI principa je prikazan u listingu 4.10.

```

public class ChristmasProductDiscount
{
}

public class Product
{
    public void AdjustPriceWith(ChristmasProductDiscount discount)
    {
    }
}

public class LinqProductRepository
{
    public IEnumerable<Product> FindAll()
    {
        return new List<Product>();
    }
}

public class ProductService
{
    private LinqProductRepository _productRepository;
    private ChristmasProductDiscount _discountStrategy;

    public ProductService()
    {
        _productRepository = new LinqProductRepository();
        _discountStrategy = new ChristmasProductDiscount();
    }

    public IEnumerable<Product> GetProducts()
    {
        IEnumerable<Product> products = _productRepository.FindAll();
        foreach (Product p in products)
            p.AdjustPriceWith(_discountStrategy);
        return products;
    }
}
  
```

```
}
```

Listing 4.10 - Kod klasa iz domena pre primene DI principa

U projekat dodajemo klasu *ChristmasProductDiscount*. Nakon ovoga možemo dodati klasu *Product* koja će predstavljati proizvode. Ona sadrži jednu metodu koja nema kod, ali demonstrira interakciju između klase *Product* i klase *ChristmasProductDiscount*. U projekat dodajemo još jednu klasu *LinqProductRepository*. Zbog jednostavnosti, implementacija *FindAll* metode vraća praznu kolekciju proizvoda, kako ne bismo morali da kreiramo stvarnu bazu podataka.

Da bi kreirali sve klase iz domena, potrebno je da dodamo klasu *ProductService*. Iz prethodnog koda se može videti da su u okviru konstruktora kreirane dve zavisne klase. Jedina metoda u servisu jednostavno dobavlja kolekciju proizvoda iz repozitorija i primenjuje strategiju popusta nad svakim proizvodom, pre nego što ih prosledi kodu koji ih je tražio.

Problem sa klasom *ProductService* je taj što je čvrsto povezana sa konkretnim implementacijama za repository i za popust. Ovo ima negativan efekat i čini *ProductService* klasu teškom za održavanje jer ju je nemoguće testirati samostalno. Da bi se ona testirala potrebno je imati validne klase *ChristmasProductDiscount* i *LinqProductRepository*. Ukoliko se strategija popusta promeni, moraće se uraditi i promena u okviru klase servisa.

Kako bi razdvojili moduo visokog nivoa (*ProductService*) od modula niskog nivoa (*ChristmasProductDiscount* i *LinqProductRepository*), možemo refaktorisati kod u skladu sa DIP tako što ćemo uvesti dve forme DI.

Da bi se usaglasili sa DI paternom, mora se osigurati da se moduli niskog nivoa referenciraju kroz apstrakcije, a ne kroz konkretne tipove. Zbog toga moramo dodati interfejsa za klase *ChristmasProductDiscount* i *LinqProductRepository*.

Izmenjeni kod koji implementira DI princip je prikazan u listingu 4.11.

```
public interface IProductDiscountStrategy
{
}

public interface IProductRepository
{
    IEnumerable<Product> FindAll();
}

public class ChristmasProductDiscount : IProductDiscountStrategy
{
}

public class Product
{
    public void AdjustPriceWith(IProductDiscountStrategy discount)
    {
    }
}

public class LinqProductRepository : IProductRepository
{
    public IEnumerable<Product> FindAll()
    {
        return new List<Product>();
    }
}

public class ProductService
{
    private IProductRepository _productRepository;
    private IProductDiscountStrategy _discountStrategy;
```

```

public ProductService(
    IProductRepository productRepository,
    IProductDiscountStrategy discountStrategy)
{
    _productRepository = productRepository;
    _discountStrategy = discountStrategy;
}

public IEnumerable<Product> GetProducts()
{
    IEnumerable<Product> products = _productRepository.FindAll();
    foreach (Product p in products)
        p.AdjustPriceWith(_discountStrategy);
    return products;
}
}

```

Listing 4.11 - Kod klasa iz domena nakon primene DI principa

Da bi primenili DI uvodimo konstruktor injection. Umesto da se odgovornost za kreiranje instanci za *IProductRepository* i *IProductDiscountStrategy* ostavi klasi *ProductService*, ovo se može podići na viši nivo tako što će instanca biti prosleđena kao parametar konstruktora.

Ovo refaktorisanje je učinilo da klasa *ProductService* postane otvorena za promene dok je zatvorena za modifikacije, jer bilo koji popust koji implementira *IProductDiscountStrategy* interfejs može biti primenjen nad kolekcijom proizvoda, bez da se menja *ProductService* klasa.

5. VIŠESLOJNA ARHITEKTURA

Sredinom devedesetih godina, kada su aplikacije u preduzećima postajale sve složenije i izvršavale se na računarima nekoliko stotina ili hiljada krajnjih korisnika, klijentska strana u tradicionalnom dvoslojnom klijent-server modelu predstavljala je problem koji je sprečavao izmene i proširenja iz više razloga:

- zahtevana je znatna količina resursa na klijentskoj mašini kako bi se aplikacije uspešno izvršavale, uključujući procesorsku snagu, prostor na disku i RAM memoriju
- poslovna logika je bila distribuirana između servera i klijentskih mašina što je otežavalo ažuriranje aplikacije
- značajni administrativni naponi su bili potrebni na klijentskoj strani

Varijacija dvoslojnog modela koja je rešila problem skalabilnosti u velikim sistemima pojavila se 1995. godine. Nova arhitektura sastojala se iz tri sloja, od kojih se svaki mogao nalaziti na drugoj platformi. Ovi slojevi su:

- prezentacioni sloj koji je smešten na računaru krajnjeg korisnika - klijenta
- sloj poslovne logike i obrade podataka koji je smešten na serveru (aplikativni server)
- sloj za pristup podacima koji je smešten na serveru baze podataka.

Prednosti troslojne arhitekture su brojne:

- smanjenje troškova za hardver klijentskih mašina
- zbog izdvajanja poslovne logike, kojoj pristupa veliki broj korisnika, na poseban sloj u vidu aplikativnog servera, ažuriranje i održavanje aplikacije je centrirano. Ovim se eliminiše problem distribucije softvera koji je bio prisutan u dvoslojnom klijent-server modelu
- sa dobijenom modularnošću moguće je lako zameniti neki od slojeva bez uticaja na ostale
- balansiranje opterećenja je mnogo lakše usled razdvajanja poslovne logike od servisa baze podataka.

5.1. ANTIPATERN – SMART UI

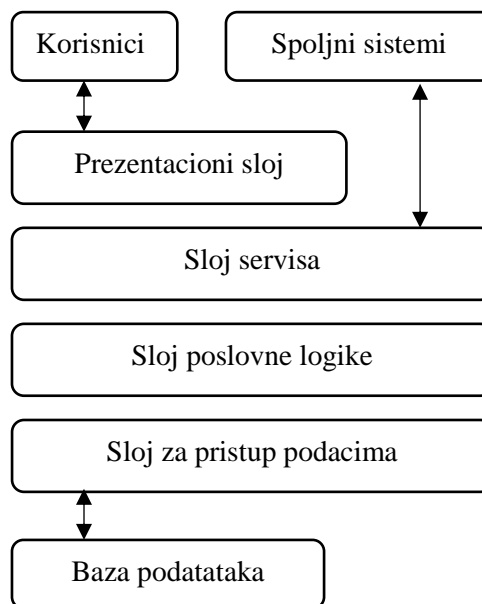
ASP.NET web forme i Visual Studio čine kreiranje aplikacija veoma lakim i jednostavnim prevlačenjem kontrola na HTML dizajner. Ovi fajlovi u sebi sadrže sve obrade događaja, pristup podacima i poslovnu logiku. Problem kod ovog pristupa je to što su svi koncepti pomešani. Ovo pravi probleme kod testiranja i rezultuje dupliranjem poslovne logike jer je teško više puta upotrebiti logiku koja je čvrsto povezana sa određenim pogledom (ASPX stranicom).

Smart UI aplikacije se, međutim, ne moraju izbegavati po svaku cenu. One su dobre za izradu prototipova i za aplikacije koje će biti kratko korišćene.

5.2. SLOJEVITA APLIKACIJA

Rešenje za Smart UI antipattern je razdvajanje aplikacije u slojeve. Razdvajanje se može postići pomoću imenskih prostora, foldera ili odvojenih projekata. Uobičajena arhitektura profesionalne ASP.NET aplikacije je prikazana na slici 5.1. Sa slike se može videti da korisnici pristupaju prezentacionom sloju. Taj sloj predstavlja izgled aplikacije i obično se implementira pomoću MVC paterna. Preko prezentacionog sloja se pristupa sloju servisa koji dalje obrađuje zahteve i komunicira sa ostatkom aplikacije.

Ukoliko neki spoljni sistem treba da koristi aplikaciju, on direktno komunicira sa slojem servisa koristeći sopstveni prezentacioni sloj.



Slika 5.1 – Arhitektura ASP.NET aplikacije

5.1. ZAŠTO KORISTITI SLOJEVE

Sloj se obično shvata kao neka vrsta crne kutije sa interfejsom koji definiše ulaz i izlaz i malo ili nimalo zavisnosti ka drugim slojevima i fizičkim serverima. Glavne prednosti koje se dobijaju upotrebom slojeva su organizacione i obuhvataju ponovnu upotrebu i razdvajanje nadležnosti. Ponovna upotreba sloja omogućava da se jednom napisani sloj upotrebljava u okviru više poslovnih aplikacija. Razdvajanje nadležnosti predstavlja ideju da slojevi budu zatvoreni, ili drugim rečima, sa visokim stepenom kohezije i niskim stepenom sprege.

Veći stepen ponovne upotrebljivosti zajedno sa odgovarajućom izolacijom funkcija, čini održavanje sistema boljim i lakšim. Logički nivoi značajno skraćuju vreme potrebno za kreiranje sistema, jer različiti razvojni timovi mogu raditi u paraleli.

Logička podela u slojeve je izrazito poželjna osobina u aplikacijama ili sistemima bez obzira na stvarnu kompleksnost.

Opšte je prihvaćeno stanovište da bi poslovni sloj trebao da poseduje svoj sopstveni modul. Međutim, u stvarnim sistemima se to ne dešava uvek. Veoma često se logika koja pripada poslovnom sloju nalazi isprepletana u drugim slojevima (prezentacioni sloj, sloj za pristup podacima). Skoro svi programeri će se složiti da to nije pravi pristup; međutim to se dešava i to veoma često.

5.2. KREIRANJE SLOJEVA U PRAKSI

Pretpostavimo da imamo klasičan sistem sa slojevima koji se sastoji od prezentacionog sloja, poslovnog sloja i sloja za pristup podacima. Koji procenat poslovne logike bi trebao da se nalazi raspoređen po slojevima? Pravilan odgovor bi bio 0-100-0.

Kada se posmatra aplikacija, ono što je zaista važno jeste da je sistem ispravan, da je lak za održavanje i da su korisnici zadovoljni. Arhitekta se mogu naći u situaciji da je jedini način da se ostvare ovi ciljevi, da se urade određeni kompromisi koji uključuju dupliranje logike i njeno prebacivanje u druge slojeve.

Zbog toga se može reći da nije neophodno staviti svu logiku u BLL (Business Logic Layer). Bilo koja količina logike koja se može pronaći u prezentacionom sloju i sloju za pristup podacima, se može tamo nalaziti zbog namernog dupliranja koje je urađeno kako bi se ostvarile neke dodatne performanse.

Dupliranje se može desiti i zbog nekih "sivih" oblasti u ukupnoj funkcionalnosti sistema u kojima nije u potpunosti jasno kom sloju one pripadaju. U nastavku su navedene neke tipične sive oblasti.

5.2.1. OBLAST #1: FORMATIRANJE PODATAKA

Jedan od najčešćih razloga zašto količina logike u prezentacionom sloju nije nula predstavlja primena poslovne logike u validaciji ulaza i formatiranju ispisa. Npr. postavlja se pitanje kako skladištiti i obraditi neke specifične podatke kao što su brojevi telefona ili vrednosti koje predstavljaju novac.

Korisnici opravdano zahtevaju da vide brojeve telefona ili novčane iznose prikazane na način kako je najlakše ljudima da ih pročitaju, odnosno formatirane na pravi način. Da li je ovo poslovna logika ili UI? Veliki broj autora bi odabrao prvu opciju, ali je ovo svakako oblast koja nije u potpunosti jasna.

Ukoliko bi formatiranje radili na UI sloju, on bi morao da zna dosta o sadržaju i logici sistema. Ako bi formatiranje radili u poslovnom sloju, morali bi ga opremiti sa odgovarajućim metodama koje bi vraćale sadržaj spreman za ispisivanje.

Postavlja se i pitanje kako čuvati sirove podatke (u ovom slučaju, brojeve). Sve u svemu postoje tri opcije:

- Skladištiti podatke dva puta (i u sirovom obliku i u obliku spremnom za prikaz)
- Skladištiti podatke u sirovom obliku, a formatiranje primeniti usput
- Skladištiti podatke u obliku spremnom za prikaz

Većina autora bi odmah izabrala opciju #2. Ovo je tip odluke koju arhitekta donose kao tip kompromisa.

Skladištenje podataka dva puta ne predstavlja dobro rešenje pre svega jer povećava kompleksnost koda i donosi dodatan posao prilikom ažuriranja. Drugim rečima, duplirani podaci rezultuju potrebom da se primeni više testova, dovode do rizika pojave grešaka, itd.

5.2.2. OBLAST #2: CRUD OPERACIJE

Kada postoji logika u sloju za pristup podacima, to obično znači da je određena logika kodirana u obliku ugrađenih procedura. Međutim ovo može dovesti do interesantnog ponašanja. Posmatraćemo sledeći problem: kako obrisati slog sa podacima o klijentu?

Verovatno bi se svi složili da je u BLL klijent poslovni objekat čiji su podaci mapirani na neki slog u bazi podataka. Neke odluke u poslovnoj logici moraju da se ispune kako bi "obrisali" klijenta. U suštini, potrebno je doneti odluke povodom sledećih pitanja: Da li klijent može da bude obrisani? Drugi slogovi, kao što su podaci o narudžbinama za datog klijenta, mogu onemogućiti brisanje klijenta. Koji još zavisni slogovi treba da budu obrisani? Da li brisati sve narudžbine koje je izvršio dati klijent, kao i da li brisati sve kontakte u kojima je on naznačen? Šta je još potrebno uraditi pre i posle brisanja? Da li je potrebno obavestiti druge procese, zatvoriti račune ili poslati obaveštenja nabavljačima?

Ovakva komplikovana logika se idealno uklapa u ponašanje poslovnog objekta. Kod koji implementira ponašanje poslovnih objekata zna kada objekat može biti obrisani kao i da li može biti obrisani, i šta to predstavlja za ostatak sistema.

Sloja za pristup podacima bi trebao da bude zadužen samo da prenese operacije ka bazi, prilikom čega ne bi trebao da bude svestan ni da postoji entitet pod nazivom "klijent". Sve što on treba da zna jeste kako da dođe do potrebnog sloga, da proveriti tipove podataka i da li postoje nedefinisani unosi, kao i da osigura integritet podataka i indeksiranje.

Drugi tipovi slogova možda nisu toliko složeni pa se sa njima može raditi kroz ugrađene procedure. Sve u svemu, to je pitanje odgovarajuće primene principa dizajna kao i razumevanje oblasti problema i konteksta.

5.2.3. OBLAST #3: UGRAĐENE PROCEDURE

Programeri se često susreću sa ugrađenim procedurama (SP - Stored Procedures). To se dešava bilo zbog toga što su sami programeri zaneseni njihovom upotrebom, ili zato što administratori baze podataka primoravaju programere da ih koriste. Kao rezultat toga, brisanje podataka pridruženih poslovnim objektu se obrađuje isključivo pomoću ugrađenih procedura. U ovom slučaju, logiku sadrže ugrađene procedure.

U suštini, ugrađene procedure bi trebalo posmatrati kao dodatni alat baze podataka, a ne kao skladište za pojedine delove poslovne logike. Ugrađene procedure treba koristiti kako bi se dobili ili ažurirali podaci. One bi trebale da se izvršavaju nad samo jednom tabelom, sa izuzetkom kada se tabele spajaju da bi se vratili podaci, ili kada se implementiraju ažuriranja koja prolaze kroz petlje nad podacima i koja bi zahtevala veliki broj naredbi kada bi se izvršavala kroz poslovnu logiku.

Veoma često poslovna logika živi u ugrađenim procedurama jer je takav način implementacije brz i lak. U žurbi, može se odlučiti da je (privremeno) bolje da se data funkcionalnost stavi u ugrađene procedure, sa namerom da se to kasnije ispravi kada bude više vremena.

Pored toga, izmeštanjem poslovne logike izvan ugrađenih procedura, logika postaje znatno lakša za ažuriranje, testiranje i debugiranje. Pored toga, povećava se prenosivost rešenja, jer ugrađene procedure koje su spremne za prenosivost moraju da budu veoma jednostavne. Pored toga, online procesiranje transakcija (OLTP - Online Transaction Processing) će takođe zahtevati jednostavne ugrađene procedure. Što više koda se smesti u ugrađene procedure, to će samim tim više vremena i trebati transakciji da se izvrši.

Argument koji stoji na strani primene ugrađenih procedura jeste to da njihova primena smanjuje saobraćaj ka bazi i od baze podataka.

6. PREZENTACIONI SLOJ

Ni jedna aplikacija ne bi bila korisna bez korisničkog interfejsa. Bez obzira na kvalitet koda koji se nalazi u središnjem sloju, on se ne može koristiti ukoliko ne postoji način da se on predstavi korisnicima.

Veliki broj arhitekta smatra prezentaciju manje važnim delom sistema, odnosno kao sitnicu o kojoj treba da se pobrinu kada završe sa poslovnim slojem i slojem za pristup podacima. Istina je da su korisnički interfejs (UI - User Interface), poslovna logika i kod za pristup podacima podjednako neophodni sistemu bilo kog nivoa kompleksnosti. Arhitekta i dizajneri mogu više pažnje posvetiti nekom od slojeva sve u zavisnosti od njihovih tendencija i znanja, ali na kraju rezultat mora biti takav da svaki sloj u potpunosti ispunjava očekivanja.

Prezentacioni sloj predstavlja sloj koji se dodaje na postojeći srednji sloj aplikacije. Mogućnost da se postojeći prezentacioni sloj ukloni sa postojeće aplikacije i da se zameni novim predstavlja glavni zahtev koji se pred njega postavlja.

6.1. KORISNIČKI INTERFEJS I PREZENTACIONA LOGIKA

Prezentacioni sloj se sastoji od dve glavne komponente: korisničkog interfejsa i prezentacione logike (koja se često naziva i UI logika). Korisnički interfejs daje korisniku alate pomoću kojih se program koristi. Svako ponašanje koje program obrađuje se prikazuje korisnicima kroz grafičke ili tekstualne elemente na korisničkom interfejsu. Ovi elementi pružaju informacije, predlažu akcije, i beleže aktivnosti korisnika preko tastature ili miša.

Bilo koja akcija koju korisnik izvrši na korisničkom interfejsu postaje ulaz za drugu komponentu prezentacionog sloja - prezentacionu logiku (PL - Presentation Logic). Prezentaciona logika se odnosi na sva procesiranja koja su potrebna da bi se prikazali podaci i da bi se transformisali korisnikovi unosi u komande za pozadinski deo sistema. Drugim rečima, PL je nadležan za prenos podataka od središnjeg sloja ka UI i od UI nazad ka središnjem sloju.

Prezentaciona logika je usko povezana sa prikazom podataka na ekranu. To je drugačiji tip logike kada se poredi sa logikom aplikacije - organizacijom odgovora na dati korisnikov zahtev - i poslovnom logikom - servisima i tokovima rada koji daju odgovor na dati poslovni zadatak.

Prezentacioni sloj predstavlja interfejs između korisnika i sistema. Sa jedne strane, on pruža alate koji su korisnicima aplikacije potrebni da bi je koristili. Sa druge strane, on sadrži logiku koja je neophodna kako bi se izvršila koordinacija akcija sistema i korisnika sa ciljem prikazivanja i unošenja podataka.

6.2. ODGOVORNOST PREZENTACIONOG SLOJA

U prezentacionom sloju postoji skup odgovornosti koji uključuje:

- nezavisnost od fizičkog korisničkog interfejsa
- mogućnost testiranja
- nezavisnost od modela podataka

6.2.1. NEZAVISNOST OD KORISNIČKOG INTERFEJSA

Grafički elementi sačinjavaju korisnički interfejs aplikacije. Korisnici vide ove komponente i vrše interakciju sa njima, na taj način prosleđujući komande i informacije pozadinskom delu sistema.

Isti logički sadržaj, npr. lista narudžbina, može biti prikazan na više različitih načina. U ASP.NET-u, npr. može se koristiti DataGrid kontrola ali i Repeater kontrola.

Prezentacioni sloj mora preživeti bilo kakvu promenu u grafičkom korisničkom interfejsu koja ne zahteva promenu u toku podataka u prezentacionoj logici. Sve dok su promene vezane čisto za grafičke stvari, prezentacioni sloj ih mora podržati na transparentan način.

Kao primer može se posmatrati blog. Kada se otvori stranica za konfiguraciju, ponuđeno je više tema i skinova koji se mogu primeniti. Nakon primene, stari sadržaj je prikazan u drugačijem okruženju. Blog nastavlja da radi na identičan način, samo sa novim izgledom.

Prezentacioni sloj mora biti nezavistan od primenjene UI tehnologije i platforme. Ovaj zahtev je dosta težak za ispunjavanje, a ponekad je i nemoguće postići potpunu nezavisnost. Zbog toga bi se ovaj zahtev mogao izraziti i kao "nezavisnost od UI tehnologije i platforme u meri koliko je to moguće (ili poželjno)".

Čest je slučaj da poslovni sistemi imaju više načina prezentovanja: Web, Windows forme, mobilne aplikacije. U datim situacijama, poželjno je da se ista prezentaciona logika koristi ispod očigledno različitih komponenti korisničkog interfejsa.

Dobro definisani prezentacioni sloj olakšava programerima da više puta iskoriste veliku količinu koda kada primenjuju različite UI tehnologije i platforme za isti logički korisnički interfejs.

6.2.2. PODRŠKA ZA TESTIRANJE

Prezentacioni sloj treba testirati do određene mere, kao i sve druge slojeve u aplikaciji. Međutim, testiranje prezentacionog sloja je teže nego što se na prvi pogled može učiniti. Potrebno je testirati da se kad korisnik izvrši određenu akciju, korisnički interfejs izmeni u skladu sa tim. Npr. podaci bi trebali da imaju korektan tok ka korisničkom interfejsu i iz njega. Pored toga, stanje korisničkog interfejsa bi trebalo da se ažurira - neki dugmići mogu biti onemogućeni, neki paneli mogu biti prikazani ili sakriveni, itd.

Komponente korisničkog interfejsa rade tako što pokreću događaje koje izvršno okruženje detektuje i mapira na odgovarajuće obrađivače. Nije lako simulirati događaj klika iz okruženja za testiranje. Zbog toga bi trebalo kod za testiranje izmestiti iz obrada događaja. Na ovaj način, svaki događaj se obrađuje pomoću metode koja se nalazi u nekoj odvojenoj klasi. Ova klasa se individualno testira.

Kako onda potvrditi da data akcija generiše očekivane rezultate? Ni u jednom alatu za testiranje programskih jedinica se ne može potvrditi da posmatrani metod daje očekivani vizuelni rezultat. Međutim, može se doći do apstraktnog opisa pogleda i jednostavno testirati da li su korektne informacije dostavljene objektu koji predstavlja pogled.

Apstrakcija je ključni alat za dodavanje mogućnosti testiranja u prezentacionom sloju. Ona se postiže razdvajanjem pogleda od prezentacione logike.

6.2.3. NEZAVISNOST OD MODELA PODATAKA

Poslovni sistem poseduje svoj sopstveni prikaz podataka. U poslovnom sloju aplikacije, modeli podataka mogu biti organizovani na različite načine pomoću odgovarajućih paterna. Bez obzira koji model da se odabere za podatke u središnjem sloju, to ne bi trebalo da ima uticaj na prezentacioni sloj.

Upotreba objekata za transfer podataka (DTO - Data Transfer Objects) za prenos ka i od prezentacionog sloja omogućava da ovaj sloj bude u potpunosti nezavistan od bilo kog modela koji je primenjen u nižim aplikacionim slojevima. Sa druge strane, prevelika upotreba DTO-a značajno produžava vreme razvoja. U teoriji, par ulaznih i izlaznih DTO je potreban za svaki poziv koji id od prezentacionog sloja ka aplikacionoj logici. Izraz "svaki poziv" se odnosi na svaki poziv bez obzira sa koje stranice prezentacionog sloja je upućen. Kao posledica, dolazi se do toga da je potrebno kreirati veliki broj klasa.

Iz datog razloga, DTO se sporadično koriste i to u onim oblastima sistema gde je veća verovatnoća da će doći do izmena. Ulazak objekata iz domena u prezentacioni sloj kreira zavisnost. Što se tiče

prezentacionog sloja, uvođenje zavisnosti u model podataka može biti prihvatljivo, međutim, praksa je da se takva zavisnost uvodi samo u slučajevima kada postoji jasna korist u smislu performansi, održavanja ili i jednog i drugog.

6.3. ODGOVORNOST KORISNIČKOG INTERFEJSA

Najveći deo vremena, arhitekta nije direktno uključen u definisanje grafičkih detalja koji se odnose na korisnički interfejs. Umesto toga, odgovornost arhitekta jeste da je aplikacija odgovori na zahteve sa stanovišta kvaliteta.

Najteži deo prezentacione logike jeste kreiranje apstrakcije pogleda, za svaki pogled u prezentacionom sloju. Apstrakcija pogleda definiše koji podaci su uključeni u ulaze i izlaze pogleda. Cilj prezentacione logike jeste da se osigura da podaci korektno ulaze i izlaze iz pogleda.

Nakon što su se sve zainteresovane strane dogovorile oko svih potrebnih pogleda, dva različita tima mogu krenuti da rade na projektu. Tim za razvoj može raditi na implementaciji i testiranju prezentacione logike i središnjeg sloja. Dizajn tim može kreirati jedan ili više grafičkih prikaza za svaki od podržanih korisničkih interfejsa. Rezultujući UI i prezentaciona logika mogu biti povezani u svakom momentu.

Svaki pogled koji čini korisnički interfejs ima sopstveni skup odgovornosti, koji se uglavnom odnosi na to da se ukupni prezentacioni sloj učini što je moguće lakšim.

6.3.1. KORISTAN PRIKAZ PODATAKA

Između ostalog, korisnički interfejs ima zadatak da prikaže podatke korisniku. Podaci uključuju opšte informacije, savete, i što je najvažnije, rezultate dobijene primenom operacija iz sistema.

Dobar korisnički interfejs uvek prikazuje korisniku korektno formatirane podatke na upotrebljiv način. Korisnički interfejs takođe sakriva efekte globalizacije i lokalizacije i koristi odgovarajuća podešavanja kad god je to potrebno.

U zavisnosti od paterna koji su primenjeni u prezentacionom sloju, mesto gde se podaci pripremaju za prikaz može varirati. Najčešće se to dešava u prezentacionoj logici, ali nije neobično da se neko formatiranje i logika prikaza primenjuju kroz osobine bogatih kontrola koje se nalaze u korisničkom interfejsu. U ovom slučaju postoji prednost da se prikaz menja na opisan način bez potrebe da se napiše i jedna linija koda.

6.3.2. KOMFORAN UNOS PODATAKA

Korisnički interfejs predstavlja mesto gde korisnici unose podatke u sistem. Koristan prezentacioni sloj omogućava komforan unos podataka omogućavajući korisnicima ulazne maske i upotrebu odgovarajućih kontrola za unos podataka. U skladu sa tim, bogate kontrole i prenesi-i-pusti principi predstavljaju osnovne funkcionalnosti.

Bez obzira na pozadinsku platformu (Windows, Web, Mobile), unos podataka predstavlja kritičan deo aplikacije. Zbog toga je neophodno vršiti brzu validaciju podataka što predstavlja još jednu veoma važnu osobinu korisničkog interfejsa. Validacija na korisničkom nivou nije dovoljna, ali predstavlja dobar prvi korak.

6.3.3. OPŠTI PRIKAZ

Izgled i osećaj aplikacije se odnosi na utisak koji korisnik ima kada koristi datu aplikaciju. Ovo predstavlja glavnu osobinu opšteg prikaza. Pronalaženje prave kombinacije kontrola, vizuelnih prikaza i stilova predstavlja proces koji može da oduzme značajnu količinu vremena. To predstavlja deo sistema o kom klijenti imaju najviše pitanja.

Obično aplikacije koje su namenjene javnoj upotrebi, kao što su to Web portali, imaju veoma lep korisnički interfejs jer je jedan od njihovih ciljeva da pridobiju korisničku pažnju i ostave dobar utisak. Sa druge strane, aplikacije koji imaju upotrebu unutar korporacije koriste interfejs kod kog je akcenat na upotrebljivosti a ne na vizuelnom zadovoljstvu.

Na kraju, to je stvar ispunjavanja klijentskih zahteva. UI arhitektura treba da bude urađena u skladu sa aktuelnim zahtevima.

6.4. MVC PATERN

Termin Model-View-Controller (MVC) je u upotrebi od 70-ih godina prošlog veka. Nastao je kao deo Smalltalk projekta u okviru kompanije Xerox PARC kao način organizacije u okviru ranih GUI aplikacija.

MVC zahteva razdvajanje odgovornosti (SoC) jer su model i logika kontrolera razdvojeni od korisničkog interfejsa. U okviru Web aplikacija, ovo znači da je HTML kod izdvojen od ostatka aplikacije, što čini održavanje i testiranje jednostavnijim i lakšim.

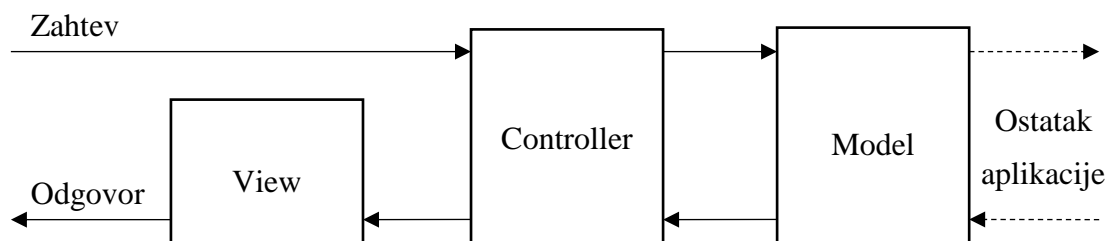
Posmatrajući sa visokog nivoa apstrakcije, MVC patern razdvaja aplikaciju u tri glavne komponente:

- Model - predstavlja poslovne podatke koje će pogled da prikaže ili da modifikuje
- Controller - vrši interakciju sa modelom na osnovu zahteva koje dobija preko browser-a , izvršava operacije nad modelom i bira odgovarajući view koji će biti prikazan
- View - je pasivan i ne poseduje znanje o kontroleru. On jednostavno prikazuje podatke iz modela koje je dobio od controller-a

6.4.1. ASP.NET IMPLEMENTACIJA MVC-A

U okviru MVC-a, kontroleri predstavljaju C# klase koje su izvedene iz klase *Controller* koja se nalazi u okviru paketa *System.Web.Mvc*. Svaka *public* metoda u klasi koja je izvedena iz klase *Controller* se naziva akciona metoda i može joj se pristupiti preko URL-a na način kako se odredi u okviru ASP.NET sistema za rutiranje. Kada se zahtev prosledi ka URL-u koji je pridružen nekoj akcionoj metodi, izrazi u okviru klase kontrolera se izvršavaju kako bi se izvele određene operacije nad domenom modela i kako bi se zatim odabrao odgovarajući pogled.

MVC patern je šematski prikazan na slici 6.1. Kao što se može videti, korisnik upućuje zahtev koji prvi prima i obrađuje controller. Controller vrši interakciju sa modelom na osnovu dobijenog zahteva i dobavlja podatke. On prikazuje odgovarajući pogled (view) i obezbeđuje ga potrebnim podacima koji treba da budu prikazani.



Slika 6.1 – Šematski prikaz MVC paternu

ASP.NET MVC Framework daje podršku za više metodologija za kreiranje pogleda. Ranije verzije su koristile standardan ASP.NET view engine koji je kreirao ASPX stranice na način veoma sličan Web Form sintaksi. MVC 3 je uveo upotrebu Razor view engine-a i koji koristi potpuno drugačiju sintaksu.

Ne postoje bilo kakva ograničenja kada je u pitanju implementacija domen modela. Možemo kreirati model pomoću standardnih C# objekata i implementirati skladištenje pomoću bilo koje baze

podataka, objektno-relacionog mapiranja, ili drugih alata koji su podržani od strane .NET-a. Visual Studio sam kreira folder *Models* kao deo šablona za MVC projekat. Ovo je odgovarajuće za jednostavne projekte, dok složenije aplikacija teže kreiranju domen modela u posebnom Visual Studio projektu.

Primer MVC aplikacije je dat u nastavku. Prvo kreiramo model. To je klasa *Product* koja sadrži propertyje za identifikator, ime, opis, cenu i kategoriju kojoj pripada:

```
namespace Razor.Models
{
    public class Product
    {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
}
```

Listing 6.1 - Kod u okviru klase modela *Product*

Nakon toga kreiramo kontroler *HomeController* koji je izveden iz klase *Controller*. Ova klasa sadrži jednu javnu metodu *Index* kojoj se može pristupiti preko URL-a. U okviru akcione metode se kreira nova instanca klase *Product* kojoj se dodeljuju vrednosti (u realnim sistemima ovde bi se pristupalo sloju servisa koji bi zatim dobavljaio podatke iz baze u skladu sa poslovnom logikom). Kreirani objekat se vraća pogledu.

```
namespace Razor.Controllers
{
    public class HomeController : Controller
    {
        // GET: Home
        public ActionResult Index()
        {
            Product myProduct = new Product
            {
                ProductID = 1,
                Name = "LG G4",
                Description = "New phone from LG",
                Category = "Mobile phones",
                Price = 475M
            };

            return View(myProduct);
        }
    }
}
```

Listing 6.2 - Kod u okviru kontrolera *HomeController*

Poslednji korak je dodavanje pogleda koji će prikazivati podatke. Pogled uključuje klasu *Product* kao svoj model, čime se govori da je pogled namenjen za prikazivanje podataka iz datog dela modela. Od podataka korisniku ispisujemo ime proizvoda.

```
@model Razor.Models.Product

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
```

```
</head>
<body>
  <div>
    @Model.Name
  </div>
</body>
</html>
```

Listing 6.3 - Kod u okviru pogleda *Index.cshtml*

Prethodni kod radi na sledeći način:

- Korisnik u internet pretraživaču unese adresu do sajta koji želi da pogleda. Ako je ta adresa `www.imesajta.com/Home/Index` to znači da će se izvršiti metoda *Index* koja se nalazi u okviru *HomeController* kontrolera
- Izvršava se metoda *Index* i u okviru nje se kreira nova instanca klase *Product* koja se vraća pogledu. Ovo znači da smo iz kontrolera pristupili modelu (klasi *Product*) koju smo dobavili i vratili pogledu. Ime pogleda koji vraćamo je isto sa imenom metode (*Index.cshtml*).
- Prikazuje se pogled u okviru internet pretraživača. Na mestu u kodu gde stoji `@Model.Name` će se ispisati stvarno ime proizvoda koji smo prosledili iz kontrolera

7. SLOJ SERVISA

U većini slučajeva, sloj servisa (SL - Service Layer) se posmatra kao deo poslovnog sloja u domen model pristupu. Iako je ovo najčešći scenarijo, postoje i druga shvatanja i primene. U suštini, sloj servisa definiše interfejs za prezentacioni sloj kako bi se pozvale predefinisane akcije sistema. On predstavlja vrstu granice koja označava gde se završava prezentacioni sloj a gde počinje sloj sa poslovnom logikom. Sloj servisa je dizajniran kako bi održao vezu između prezentacionog sloja i sloja sa poslovnom logikom na minimumu, bez obzira na to kako je poslovna logika organizovana. Dakle, moguće je imati sloj servisa bez obzira na odabrani patern u poslovnom sloju.

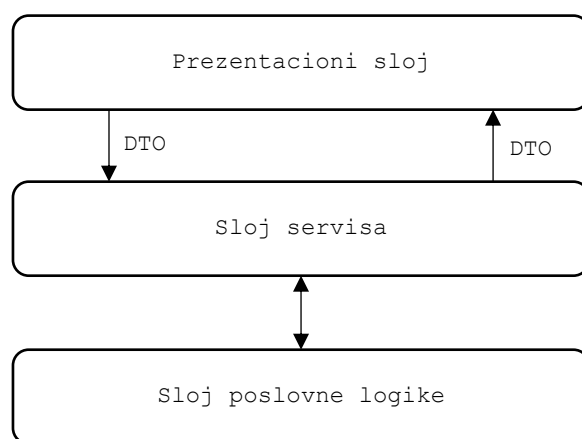
U sloju servisa, kod se poziva direktno iz korisničkog interfejsa. Poziva se metoda koja uzima određene podatke u zadatom formatu a vraća neke druge podatke. Podaci se u sloj i iz sloja prenose pomoću objekata za prenos podataka (DTO - Data Transfer Objects).

Prema definiciji za sloj servisa koju je dao Martin Fowler u svojoj knjizi "Patterns of Enterprise Application Architecture", sloje servisa je dodatni sloj koji postavlja granicu između dva susedna sloja. Danas je ova definicija široko prihvaćena.

Sloj servisa se obično mapira na poslovne slučajeve korišćenja. On se nalazi između prezentacionog sloja i sloja poslovne logike i predstavlja interfejs koji definiše granice sistema i operacije koje su omogućene klijentu. Njegova uloga je enkapsulacija poslovne logike, validacije tokova podataka, kao i koordinacije rada sa bazom podataka.

7.1. NAMENA SLOJA SERVISA

Sloj sa servisima se nalazi između dva dodirujuća logička sloja u sistemu. On date slojeve održava labavo povezane i istovremeno dovoljno razdvojene, pri čemu oni mogu da komuniciraju jedan sa drugim. U većini slučajeva SL (Service Layer) patern se koristi da definiše granice aplikacije između prezentacionog sloja i poslovnog sloja. Sledeća slika prikazuje ulogu SL-a u višeslojnoj aplikaciji.



Slika 7.1 - Uloga sloja servisa u višeslojnoj aplikaciji

U svojoj suštini, svaka interakcija sa korisnikom ima dva glavna učesnika: korisnički interfejs koji je implementiran od strane prezentacionog sloja, i modul koji odgovara na akciju korisnika a koji je implementiran u sloju servisa.

Sve interakcije započinju u prezentacionom sloju nakon čega se prenose u sloj sa servisima. Na osnovu dobijenog ulaza, SL pristupa komponentama u poslovnom sloju, uključujući servise, tokove rada, i objekte u domen modelu, i ukoliko je potrebno, poziva sloj za pristup podacima.

Uopšteno rečeno, sloj servisa predstavlja skup klasa koje predstavljaju logički povezane metode koje drugi sloj, u opštem slučaju prezentacioni sloj, može pozvati kako bi ispunio određeni slučaj korišćenja.

Nezavistan od bilo koji tehnologije, reč *servis* ukazuje na softverski kod koji servisira zahteve koje jedan sloj šalje ka drugom. Izraz *sloj servisa* se odnosi na kolekciju servisa koji zajednički kreiraju posrednički sloj između dva sloja koja međusobno komuniciraju.

Velikom broju ljudi reč servis označava više od dela koda koji servisira dolazne zahteve. Ova promena u gledištu je nastala kao posledica prednosti koje su doneli orijentacija ka servisima (SO - Service Orientation) i arhitektura bazirana na servisima (SOA - Service Oriented Architecture).

SO je način dizajniranja poslovnih procesa kao skupa međusobno povezanih servisa. Tu se ne govori o samoj tehnologiji, već je to prosto drugačiji pristup u načinu opisa rada datog poslovnog sistema.

SOA se odnosi na IT arhitekturu koja posmatra svoje resurse kao servise i vrši njihovo povezivanje statički ili na zahtev. U SOA svetu, aplikacije nastaju kao rezultat kompozicije i integracije nezavisnih servisa.

Servise izvršavaju klijenti, pri čemu klijent jedan servis može izvršiti više puta. Pod klijentima se podrazumevaju korisnici, drugi servisi u aplikaciji, kao i drugi delovi poslovne logike kao što su tokovi rada.

Kao primer upotrebe servisa možemo posmatrati aplikaciju koja kao zadatak ima konvertovanje novca između različitih valuta. Ovakva operacija predstavlja prikladno mesto za upotrebu servisa. Kada se definišu ulazni i izlazni podaci servisa, ostali delovi aplikacije koriste dati servis. Kod koji se u njemu nalazi se može upotrebiti nebrojeni broj puta.

Ovde se može postaviti sledeće pitanje: ako je cilj višestruka upotrebljivost, zašto se ne koristi obična klasa? U suštini, kada se implementira servis, za tu svrhu se koriste klase. Dakle, servisi i klase dele zajedničku osnovu, ali servis je više specijalizovan od klase jer koristi prednosti svog okruženja i poseduje autonoman život.

Da bi se bolje uočile razlike između servisa i klase, može se sagledati definicija servisa koju je dao OASIS (Organization for the Advancement of Structured Information Standards): Mehanizam da se omogući pristup jednoj ili više mogućnosti, gde je pristup omogućen korišćenjem predefinisane interfejsa pri čemu se izvršava u skladu sa ograničenjima i polisama navedenim u opisu servisa.

SOA se odnosi na principe i praksu dizajniranja skupa slabo povezanih servisa. Servisi su u suštini osnovne poslovne funkcionalnosti koje koristi jedna ili više poslovnih aplikacija. U realnim aplikacijama se često dešavaju promene, ali osnovne funkcionalnosti se uglavnom ne menjaju. Ukoliko su funkcionalnosti kreirane kao nezavisni servisi, sama aplikacija daje veću fleksibilnost i omogućava lakše i brže kreiranje poslovne aplikacije oko osnovnih poslovnih procedura.

Prilikom kreiranja SOA aplikacija postoje četiri osnovna principa koja treba slediti:

- **Granice su jasne** – interfejs servisa bi trebao da bude što jasniji i jednostavniji
- **Servisi su autonomni** – metode servisa ne bi trebale da zavise od drugih metoda u cilju obavljanja poslovnih transakcija. Klijent ne bi trebao da poziva metode u određenoj sekvenci kako bi obavio poslovnu transakciju. On bi trebao da pozove jedan servis i da u okviru jedne akcije dobije odgovor o uspehu ili neuspehu te transakcije.
- **Servisi prikazuju ugovore a ne klase** – servisi bi trebali da otkriju samo ugovore a ne i konkretne implementacije
- **Kompatibilnost servisa se zasniva na politici** – servis bi treba da otkrije politiku za čega se on može koristiti. Klijenti zatim koriste servis sa dobrim znanjem kako da ga koriste i šta da očekuju kao odgovor

7.2. PREDNOSTI SERVISNOG SLOJA

Servisni sloj nudi jedinstvenu ugovornu tačku između korisničkog interfejsa i središnjeg sloja, u kom se koncentriše logika aplikacije. Logika aplikacije predstavlja deo poslovne logike koja proističe direktno iz slučajeva korišćenja.

Servisi sa svojim izvršnim okruženjem omogućavaju da se delovi koda udalje od prezentacionog sloja. Sa serverske strane, pozvani metod iz serverskog sloja izvršava traženu logiku tako što poziva domen model, specifične aplikacione servise, radne tokove i sve ostalo iz čega je sačinjen poslovni sloj.

Bez servisnog sloja, pojavljuju se direktni pozivi ka aplikacionim servisima koji su zatraženi iz prezentacionog sloja. Rezultat je rizik da se pojavi više udaljenih poziva kako bi se ispunio zahtevani zadatak, što dovodi do loših performansi.

7.3. PATERNI U OKVIRU SLOJA SERVISA

Uloga sloja servisa je da predstavlja pristupnu tačku aplikaciji (naziva se i fasada). Sloj servisa pruža prezentacionom sloju snažno tipizirani model namenjen prikazu, koji se naziva i *prezentacioni model*.

Prezentacioni model koristi DTO koji se prosleđuju od strane aplikacije prezentacionom sloju kako bi bili prikazani. Ovi objekti se obično kreiraju u okviru sloja servisa i često imaju ekstenziju *ViewModel* (npr. *ProductViewModel*). Zbog kreiranja posebnih objekata koji služe za prenos podataka a koji su nezavisni od načina kreiranja domen modela, mora se kreirati mapiranje između prezentacionog modela i domen modela aplikacije.

Kako bi klijent mogao da vrši interakciju sa slojem servisa, koristiće se patern **Request/Response**. Deo request će biti pružen od strane klijenta i sadržaćе sve neophodne parametre.

7.4. PRIMER APLIKACIJE SA SLOJEM SERVISA

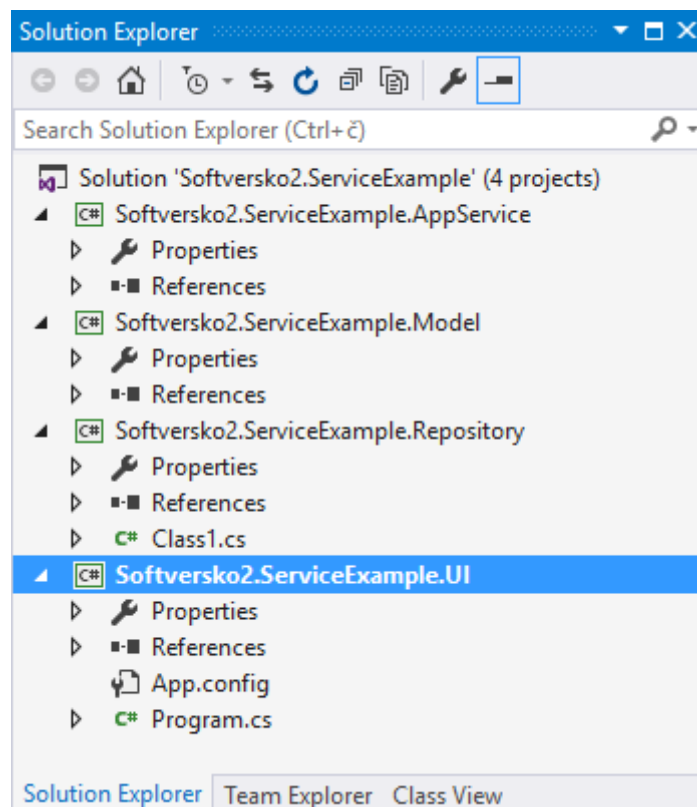
U cilju kreiranja aplikacije koja demonstrira sloj servisa, kreiraće se aplikacija koja modeluje domen bankarstva, a koja uključuje kreiranje naloga i transfer sredstava između njih. Da bi se uradila ovakva aplikacija, potrebno je uraditi sledeće korake:

1. Pokrenuti Visual Studio 2015 i kreirati novi projekt koji je tipa *Visual Studio Solution* → *Blank Solution*. Projekat kreirati pod imenom *Softversko2.ServiceExample*.
2. Za potrebe kreiranja sloja poslovne logike, u okviru projekta dodamo novi Class Library projekat pod imenom *Softversko2.ServiceExample.Model*. Iz novo kreiranog projekta treba obrisati klasu *Class1.cs* koju Visual Studio automatski kreira. Ovaj projekat će sadržati svu poslovnu logiku koja se nalazi u okviru aplikacije, odnosno da bi predstavili domen bankarstva. Projekat će definisati i ugovore u obliku interfejsa za dobavljanje podataka. *Model* projekat nema reference ni ka jednom drugom projektu, što osigurava da neće zavisiti od infrastrukture i organizacije samog rešenja.
3. Za potrebe kreiranja sloja servisa, u okviru projekta dodamo novi Class Library projekat pod imenom *Softversko2.ServiceExample.AppService*. Iz novo kreiranog projekta treba obrisati klasu *Class1.cs*. *AppService* projekat će predstavljati API u okviru aplikacije. Prezentacioni sloj će komunicirati sa *AppService* projektom pomoću poruka, koje predstavljaju jednostavnije objekte za prenos podataka. Ovaj sloj će takođe definisati i modele potrebne za prikaz, koji predstavljaju pojednostavljen domen model čija je namena samo prikaz podataka korisniku.
4. Za potrebe kreiranja sloja za pristup bazi podataka u okviru projekta dodajemo novi Class Library projekat pod imenom *Softversko2.ServiceExample.Repository*. Iz novo kreiranog projekta treba obrisati klasu *Class1.cs*. *Repository* projekat će sadržati implementaciju

repository interfejsa koji su definisani u okviru *Model* projekta. Ovaj projekta ima referencu ka *Model* projektu jer treba da dobavlja objekte domena iz skladišta.

5. Za potrebe kreiranja prezentacionog sloja dodajmo novi Console application projekat pod imenom *Softversko2.ServiceExample.UI*. Ovaj projekat ćemo postaviti da bude **Startup** što znači da će se on prvi izvršavati (u njemu će biti *main* metoda). Ovaj projekat komunicira samo sa *AppService* i dobija snažno tipiziran model koji je kreiran samo kako bi bio prikazan korisniku.
6. Sledeći korak je dodavanje referenci. U projekat *Repository* dodajemo referencu na projekat *Model*. U projekat *AppService* dodajemo referencu na projekat *Model* i na projekat *Repository*. Na kraju u konzolnu aplikaciju dodajemo referencu na *AppService*.

Nakon dodavanja projekata, izgled Solution Explorer-a u okviru Visual Studio-a bi trebao da bude kao na slici 7.2.



Slika 7.2 - Izgled Solution Explorer-a nakon dodavanja projekata u aplikaciju

7.4.1. KREIRANJE POSLOVNOG SLOJA

Kada su kireirani potrebni projekti, moguće je krenuti sa modelovanjem domena. U okviru *Model* projekta dodajemo klasu *BankAccount* sa sledećim kodom:

```
namespace Softversko2.ServiceExample.Model
{
    public class BankAccount
    {
        private Guid accountNo;
        private decimal balance;
        private string ownerName;

        public BankAccount() : this(Guid.NewGuid(), 0, "") { }

        public BankAccount(Guid accountNo, decimal balance, string ownerName)
```

```

    {
        this.accountNo = accountNo;
        this.balance = balance;
        this.ownerName = ownerName;
    }

    public Guid AccountNo
    {
        get { return accountNo; }
        internal set { accountNo = value; }
    }

    public decimal Balance
    {
        get { return balance; }
        internal set { balance = value; }
    }

    public string OwnerName
    {
        get { return ownerName; }
        set { ownerName = value; }
    }

    public bool CanWithdraw(decimal amount)
    {
        return (Balance >= amount);
    }

    public void Withdraw(decimal amount)
    {
        if (CanWithdraw(amount))
        {
            Balance -= amount;
        }
    }

    public void Deposit(decimal amount)
    {
        Balance += amount;
    }
}

```

Listing 7.1 - Kod u okviru klase *BankAccount*

Nakon ovoga nam je potrebna metoda koja će omogućiti skladištenje *BankAccount* objekata. Ova metoda ne treba da bude deo *Model* projekta pa ćemo dodati samo interfejs za *Repository* kako bi smo definisali ugovor za potrebe skladištenja i dobaljanja entiteta. Kreiraćemo interfejs *IBankAccountRepository* sa sledećim kodom:

```

namespace Softversko2.ServiceExample.Model
{
    public interface IBankAccountRepository
    {
        void Add(BankAccount bankAccount);
        IEnumerable<BankAccount> FindAll();
        BankAccount FindBy(Guid AccountId);
    }
}

```

Listing 7.2 - Kod u okviru interfejsa *IBankAccountRepository*

Određene akcije se ne uklapaju dovoljno da bi bile dodate kao metode domen entiteta. U takvim slučajevima se koristi domen servis. Akcije prebacivanja sredstava između dva računa predstavljaju odgovornost koja pripada klasi servisa. Dodajemo novu klasu u *Model* projekat pod imenom *BankAccountService*:

```
namespace Softversko2.ServiceExample.Model
{
    public class BankAccountService
    {
        private IBankAccountRepository bankAccountRepository;

        public BankAccountService(IBankAccountRepository bankAccountRepository)
        {
            this.bankAccountRepository = bankAccountRepository;
        }

        public void Transfer(Guid accountNoTo, Guid accountNoFrom,
            decimal amount)
        {
            BankAccount bankAccountTo =
                bankAccountRepository.FindBy(accountNoTo);
            BankAccount bankAccountFrom =
                bankAccountRepository.FindBy(accountNoFrom);
            if (bankAccountFrom.CanWithdraw(amount))
            {
                bankAccountTo.Deposit(amount);
                bankAccountFrom.Withdraw(amount);
            }
        }
    }
}
```

Listing 7.3 - Kod u okviru klase *BankAccountService*

7.4.2. KREIRANJE SLOJA ZA PRISTUP PODACIMA

Da bi omogućili čuvanje podataka i njihovo dobavljanje, u okviru *Repository* projekta će biti dodata nova klasa *BankAccountRepository*. Ova klasa će predstavljati implementaciju interfejsa *IBankAccountRepository*.

```
namespace Softversko2.ServiceExample.Repository
{
    public class BankAccountRepository : IBankAccountRepository
    {
        public static List<BankAccount> bankAccounts = new List<BankAccount>();

        public void Add(BankAccount bankAccount)
        {
            bankAccounts.Add(bankAccount);
        }

        public IEnumerable<BankAccount> FindAll()
        {
            return bankAccounts;
        }

        public BankAccount FindBy(Guid AccountId)
        {
            return bankAccounts.Where(x =>
                x.AccountNo == AccountId).FirstOrDefault();
        }
    }
}
```



```
}
```

Listing 7.4 - Kod u okviru klase *BankAccountRepository*

7.4.3. KREIRANJE SLOJA SERVISA

Sloj servisa omogućava korisniku laku interakciju sa sistemom. U projekat *AppService* ćemo dodati novi folder *ViewModel* i u okviru njega dve klasu *BankAccountView* sa sledećim kodom:

```
namespace Softversko2.ServiceExample.AppService.ViewModel
{
    public class BankAccountView
    {
        public Guid AccountNo { get; set; }
        public string Balance { get; set; }
        public string OwnerName { get; set; }
    }
}
```

Listing 7.5 - Kod u okviru klase *BankAccountView*

Klasa *BankAccountView* daje pojednostavljen prikaz domen modela u cilju prikazivanja podataka korisniku. Da bi smo transformisali entitete domena u entitete prikaza, potrebna nam je mapirajuća klasa. Kreiraćemo klasu *ViewModelMapper* sa statičkom metodom:

```
namespace Softversko2.ServiceExample.AppService.ViewModel
{
    public class ViewModelMapper
    {
        public static BankAccountView CreateBankAccountViewFrom(
            BankAccount acc)
        {
            BankAccountView bankView = new BankAccountView();
            bankView.AccountNo = acc.AccountNo;
            bankView.Balance = acc.Balance.ToString("C");
            bankView.OwnerName = acc.OwnerName;
            return bankView;
        }
    }
}
```

Listing 7.6 - Kod u okviru klase *ViewModelMapper*

U *AppService* projekat ćemo dodati još jedan folder pod imenom *Messages*. Ovaj folder će sadržati sve zahtev-odgovor objekte koji se koriste za komunikaciju sa slojem servisa. Pošto svi odgovori dele sličan skup osobina, moguće je kreirati abstraktnu baznu klasu *ResponseBase* sa sledećim kodom:

```
namespace Softversko2.ServiceExample.AppService.Messages
{
    public abstract class ResponseBase
    {
        public bool Success { get; set; }
        public string Message { get; set; }
    }
}
```

Listing 7.7 - Kod u okviru klase *ResponseBase*

Properti *Success* govori da li se pozvana metoda uspešno izvršila, a properti *Message* sadrži detalje izlaza nakon izvršavanja metode.

Sada je potrebno implementirati sve *zahtev-odgovor* objekte kreiranjem sledećih klasa:

```
namespace Softversko2.ServiceExample.AppService.Messages
{
    public class BankAccountCreateRequest
    {

```

```

        public string CustomerName { get; set; }
    }
}

```

Listing 7.8 - Kod u okviru klase *BankAccountCreateRequest*

```

namespace Softversko2.ServiceExample.AppService.Messages
{
    public class BankAccountCreateResponse : ResponseBase
    {
        public Guid BankAccountId { get; set; }
    }
}

```

Listing 7.9 - Kod u okviru klase *BankAccountCreateResponse*

```

namespace Softversko2.ServiceExample.AppService.Messages
{
    public class DepositRequest
    {
        public Guid AccountId { get; set; }
        public decimal Amount { get; set; }
    }
}

```

Listing 7.10 - Kod u okviru klase *DepositRequest*

```

namespace Softversko2.ServiceExample.AppService.Messages
{
    public class FindAllBankAccountResponse : ResponseBase
    {
        public IList<BankAccountView> BankAccountView { get; set; }
    }
}

```

Listing 7.11 - Kod u okviru klase *FindAllBankAccountReposnse*

```

namespace Softversko2.ServiceExample.AppService.Messages
{
    public class TransferRequest
    {
        public Guid AccountIdTo { get; set; }
        public Guid AccountIdFrom { get; set; }
        public decimal Amount { get; set; }
    }
}

```

Listing 7.12 - Kod u okviru klase *TransferRequest*

```

namespace Softversko2.ServiceExample.AppService.Messages
{
    public class TransferResponse : ResponseBase
    {
    }
}

```

Listing 7.13 - Kod u okviru klase *TransferResponse*

```

namespace Softversko2.ServiceExample.AppService.Messages
{
    public class WithdrawalRequest
    {
        public Guid AccountId { get; set; }
        public decimal Amount { get; set; }
    }
}

```

Listing 7.14 - Kod u okviru klase *WithdrawalRequest*

Nakon kreiranja svi objekata koji će biti korišćeni za poruke, možemo dodati servis koji vrši koordinaciju poziva metoda ka entitetima domena: servis i repository. Dodajemo novu klasu *ApplicationBankAccountService* u rut *AppService* projekta:

```
namespace Softversko2.ServiceExample.AppService
{
    public class ApplicationBankAccountService
    {
        private BankAccountService bankAccountService;
        private IBankAccountRepository bankRepository;

        public ApplicationBankAccountService() :
            this(new BankAccountRepository(),
                new BankAccountService(new BankAccountRepository()))
        { }

        public ApplicationBankAccountService(
            IBankAccountRepository bankRepository,
            BankAccountService bankAccountService)
        {
            this.bankRepository = bankRepository;
            this.bankAccountService = bankAccountService;
        }

        public BankAccountCreateResponse CreateBankAccount(
            BankAccountCreateRequest bankAccountCreateRequest)
        {
            BankAccountCreateResponse bankAccountCreateResponse = new
                BankAccountCreateResponse();
            BankAccount bankAccount = new BankAccount();
            bankAccount.OwnerName = bankAccountCreateRequest.CustomerName;
            bankRepository.Add(bankAccount);
            bankAccountCreateResponse.BankAccountId = bankAccount.AccountNo;
            bankAccountCreateResponse.Success = true;
            return bankAccountCreateResponse;
        }

        public void Deposit(DepositRequest depositRequest)
        {
            BankAccount bankAccount =
                bankRepository.FindBy(depositRequest.AccountId);
            bankAccount.Deposit(depositRequest.Amount);
        }

        public void Withdrawal(WithdrawalRequest withdrawalRequest)
        {
            BankAccount bankAccount =
                bankRepository.FindBy(withdrawalRequest.AccountId);
            bankAccount.Withdraw(withdrawalRequest.Amount);
        }

        public TransferResponse Transfer(TransferRequest request)
        {
            TransferResponse response = new TransferResponse();
            bankAccountService.Transfer(request.AccountIdTo, request.AccountIdFrom,
                request.Amount);
            response.Success = true;
            return response;
        }

        public FindAllBankAccountResponse GetAllBankAccounts()
        {

```

```

        FindAllBankAccountResponse findAllBankAccountResponse = new
            FindAllBankAccountResponse();
        IList<BankAccountView> bankAccountViews = new List<BankAccountView>();
        findAllBankAccountResponse.BankAccountView = bankAccountViews;
        foreach (BankAccount acc in bankRepository.FindAll())
        {
            bankAccountViews.Add(ViewMapper.CreateBankAccountViewFrom(acc));
        }
        findAllBankAccountResponse.Success = true;
        return findAllBankAccountResponse;
    }
}

```

Listing 7.15 - Kod u okviru klase *ApplicationBankAccountService*

Klasa *ApplicationBankAccountService* koordinira aktivnostima aplikacije i delegira sve poslovne zahteve ka domen modelu. Ovaj sloj ne sadrži poslovnu logiku i pomaže u sprečavanju da ne-poslovni kod "zagadi" domen model projekat. Ovaj sloj takođe transformiše entitete domena u data transfer objekte koji omogućavaju jednostavan API sa kojim radi prezentacioni sloj.

7.4.4. SLOJ KORISNIČKOG INTERFEJSA

Poslednja akcija je kreiranje korisničkog interfejsa koji će omogućiti kreiranje računa i transakcije. U okviru *main* metode u *UI* projektu dodajemo sledeći kod:

```

namespace Softversko2.ServiceExample.UI
{
    class Program
    {
        public static void ShowAllAccounts(ApplicationBankAccountService service)
        {
            FindAllBankAccountResponse response = new FindAllBankAccountResponse();
            response = service.GetAllBankAccounts();
            foreach (BankAccountView accView in response.BankAccountView)
            {
                Console.WriteLine("Account: " + accView.OwnerName +
                    ", ammount: " + accView.Balance);
            }
            Console.WriteLine();
        }

        static void Main(string[] args)
        {
            ApplicationBankAccountService service = new
                ApplicationBankAccountService();

            BankAccountCreateRequest createAccountRequest = new
                BankAccountCreateRequest();
            createAccountRequest.CustomerName = "John";
            BankAccountCreateResponse responseAcc1 =
                service.CreateBankAccount(createAccountRequest);

            BankAccountCreateRequest createAccountRequest1 = new
                BankAccountCreateRequest();
            createAccountRequest1.CustomerName = "Mike";
            BankAccountCreateResponse responseAcc2 =
                service.CreateBankAccount(createAccountRequest1);

            Console.WriteLine("Account creation: ");
            ShowAllAccounts(service);

            DepositRequest depositRequest1 = new DepositRequest();

```

```

depositRequest1.AccountId = responseAcc1.BankAccountId;
depositRequest1.Amount = 25846M;
service.Deposit(depositRequest1);

DepositRequest depositRequest2 = new DepositRequest();
depositRequest2.AccountId = responseAcc2.BankAccountId;
depositRequest2.Amount = 37551M;
service.Deposit(depositRequest2);

Console.WriteLine("Money deposit: ");
ShowAllAccounts(service);

WithdrawalRequest withdrawRequest1 = new WithdrawalRequest();
withdrawRequest1.AccountId = responseAcc1.BankAccountId;
withdrawRequest1.Amount = 6542M;
service.Withdrawal(withdrawRequest1);

WithdrawalRequest withdrawRequest2 = new WithdrawalRequest();
withdrawRequest2.AccountId = responseAcc2.BankAccountId;
withdrawRequest2.Amount = 8542M;
service.Withdrawal(withdrawRequest2);

Console.WriteLine("Money withdrawal: ");
ShowAllAccounts(service);

TransferRequest request = new TransferRequest();
request.AccountIdFrom = responseAcc1.BankAccountId;
request.AccountIdTo = responseAcc2.BankAccountId;
request.Amount = 11450M;
service.Transfer(request);

Console.WriteLine("Money transfer: ");
ShowAllAccounts(service);

Console.ReadKey();
    }
}
}

```

Listing - 7.16 - Kod u okviru klase *Program*

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 7.3

```

file:///c:/users/zdravko/documents/visu...
Account creation:
Account: John, ammount: $0.00
Account: Mike, ammount: $0.00

Money deposit:
Account: John, ammount: $25,846.00
Account: Mike, ammount: $37,551.00

Money withdrawal:
Account: John, ammount: $19,304.00
Account: Mike, ammount: $29,009.00

Money transfer:
Account: John, ammount: $7,854.00
Account: Mike, ammount: $40,459.00

```

Slika 7.3 - Ispis nakon pokretanja aplikacije

7.5. IDEMPOTENT PATTERN

Idempotent patern osigurava da će korisnik dobiti isti odgovor ukoliko uputi isti zahtev. On se primenjuje kod npr. naručivanja karata za neki događaj. Ukoliko korisnik više puta klikne na dugme za kupovinu karata koje je naručio, ove karte će biti plaćene jednom a on će dobiti informaciju da su karte kupljene.

Da bi demonstrirali ovaj patern, kreiraćemo aplikaciju `Softversko2.Service.Idempotent` kao Blank Solution. Zatim u projekat dodajemo konzolnu aplikaciju `Softversko2.Service.Idempotent.UI` koja će predstavljati prezentacioni sloj. Drugi projekat koji dodajemo je class library aplikacija pod imenom `Softversko2.Service.Idempotent.Service` koja će predstavljati sloj servisa u ovom demo primeru. Ostali slojevi su izostavljeni zbog jednostavnosti. U UI projekat dodajemo referencu na Service projekat.

U *Service* projekat dodajemo novu klasu *MessageResponseHistory* sa sledećim kodom:

```
namespace Softversko2.Service.Idempotent.Service
{
    public class MessageResponseHistory<T>
    {
        private Dictionary<string, T> _responseHistory;

        public MessageResponseHistory()
        {
            _responseHistory = new Dictionary<string, T>();
        }

        public bool IsAUniqueRequest(string correlationId)
        {
            return !_responseHistory.ContainsKey(correlationId);
        }

        public void LogResponse(string correlationId, T response)
        {
            if (_responseHistory.ContainsKey(correlationId))
                _responseHistory[correlationId] = response;
            else
                _responseHistory.Add(correlationId, response);
        }

        public T RetrievePreviousResponseFor(string correlationId)
        {
            return _responseHistory[correlationId];
        }
    }
}
```

Listing 7.17 - Kod u okviru klase *MessageResponseHistory.cs*

Klasa će u memoriji čuvati rezultat odgovora servera koji je pridružen datom ključu.

Pored ove klase, dodajemo klase koje će predstavljati DTO zahtev i odgovor. To su klase *PurchaseTicketRequest.cs* i *PurchaseTicketResponse.cs*.

```
namespace Softversko2.Service.Idempotent.Service
{
    public class PurchaseTicketRequest
    {
        public string TicketId { get; set; }
    }
}
```

Listing 7.18 - Kod u okviru klase *PurchaseTicketRequest.cs*

```
namespace Softversko2.Service.Idempotent.Service
{
```

```

public class PurchaseTicketResponse
{
    public bool Success { get; set; }
    public string Message { get; set; }
    public string TicketId { get; set; }
}

```

Listing 7.18 - Kod u okviru klase *PurchaseTicketResponse.cs*

Nakon dodavanja svih klasa koje služe kao podrška za implementaciju servisa, možemo dodati stvarnu klasu servisa pod imenom *TicketService* sa sledećim kodom:

```

namespace Softversko2.Service.Idempotent.Service
{
    public class TicketService
    {
        private static MessageResponseHistory<PurchaseTicketResponse>
            _reservationResponse = new
                MessageResponseHistory<PurchaseTicketResponse>();

        public PurchaseTicketResponse PurchaseTicket(
            PurchaseTicketRequest purchaseTicketRequest)
        {
            PurchaseTicketResponse response = new PurchaseTicketResponse();
            // Check for a duplicate transaction using the Idempotent pattern;
            if (_reservationResponse.IsAUniqueRequest(
                purchaseTicketRequest.TicketId))
            {
                // Real code to purchase ticket and alert database
                // ...
                response.Message = "Ticker successfully purchased at :" +
                    DateTime.Now;
                _reservationResponse.LogResponse(
                    purchaseTicketRequest.TicketId, response);
            }
            else
            {
                response = _reservationResponse.RetrievePreviousResponseFor(
                    purchaseTicketRequest.TicketId);
            }
            return response;
        }
    }
}

```

Listing 7.19 - Kod u okviru klase *TicketService.cs*

Važno je tačno objasniti šta radi prethodna klasa. Prvo, klasa *TicketService* poseduje referencu na statičku instancu *MessageResponseHistory* objekta. Ovo omogućava da sve poruke koje predstavljaju odgovor servisa budu logovane u skladu sa identifikatorom. Kada se primi novi zahtev u vidu poruke, servis može da proveri *MessageResponseHistory* kako bi odredio da li je ovaj zahtev već ranije bio obrađivan. Ovaj kod se nalazi u okviru *PurchaseTicket* metode.

Prezentacioni sloj predstavlja konzolna aplikacija. U okviru klase *Program.cs* u metodi *main* dodajemo sledeći kod:

```

namespace Softversko2.Service.Idempotent.UI
{
    class Program
    {
        static void Main(string[] args)
        {
            TicketService service = new TicketService();
            PurchaseTicketResponse response = new PurchaseTicketResponse();

```

```

        response = service.PurchaseTicket(new PurchaseTicketRequest()
            { TicketId = "123" });
        Console.WriteLine(response.Message);
        Thread.Sleep(1000);

        response = service.PurchaseTicket(new PurchaseTicketRequest()
            { TicketId = "456" });
        Console.WriteLine(response.Message);
        Thread.Sleep(1000);

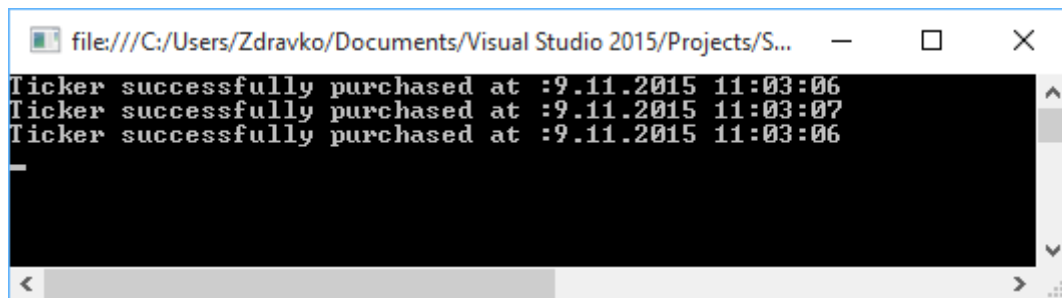
        response = service.PurchaseTicket(new PurchaseTicketRequest()
            { TicketId = "123" });
        Console.WriteLine(response.Message);
        Thread.Sleep(1000);

        Console.ReadKey();
    }
}

```

Listing 7.20 - Kod u okviru klase *Program.cs*

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 7.4. Sa slike se može videti da je u prva dva slučaja kreiran novi zahtev za naručivanjem karata. Treći zahtev koristi isti id karte kao i prvi, pa mu je i vraćen isti odgovor koji je generisan kada je prvi put upućen zahtev sa tim id-jem. To znači da se sada ne bi izvršio upit nad bazom i ostali proračuni, već bi se samo vratio isti odgovor koji je već bio generisan.



Slika 7.4 - Ispis nakon pokretanja programa

8. POSLOVNI SLOJ

Svaki softver koji prelazi minimalni nivo kompleksnosti je organizovan u slojevima. Svaki sloj predstavlja logičku celinu sistema. Moduli u poslovnom sloju uključuju sve funkcionalne algoritme i računanja koja omogućavaju da sistem radi i da vrši interakciju sa drugim slojevima, uključujući sloj za pristup bazi podataka i prezentacioni sloj.

Poslovni sloj je nervni centar bilo kog slojevitog sistema i sadrži većinu njegove logike. Iz tog razloga se ovaj sloj često naziva i *sloj poslovne logike* (BLL - Business Logic Layer). On sadrži svu logiku koja kontroliše funkcionisanje aplikacije.

Veoma teško je reći do kog nivoa složenosti može ići ovaj sloj. Mnogo je lakše odgovoriti na ovo pitanje kada se prouče zahtevi korisnika, kao i pravila i ograničenja koja se moraju primeniti. Logika gotovo da ne postoji u jednostavnom sistemu za arhiviranje koji poseduje korisnički interfejs (UI - User Interface) u obliku formi i bazu podataka. Nasuprot tome, u finansijskim aplikacijama se nalazi poprilično kompleksna logika, kao i u bilo kojoj aplikaciji koja modeluje neki realni poslovni proces.

Gde treba početi sa dizajnom BLL za realni sistem?

Kada oblast problema uključuje funkcionalnosti koje koriste nekoliko stotina entiteta, kao i hiljade poslovnih pravila, potrebno je strukturirano uputstvo o načinu kako kreirati komponente. Ovo uputstvo dolazi iz dve osnovne familije paterna: dizajn patern i patern arhitekture. Dizajn paterni predstavljaju opšte rešenje za problem u dizajnu softvera koji se često pojavljuje. Patern arhitekture je patern šireg opsega koji opisuje opštu šemu sistema i navodi odgovornosti svakog podsistema.

Uopšteno govoreći, BLL je deo softverskog sistema koji ima za cilj da odgovori na performanse poslovnih zahteva. On se sastoji od niza operacija koje se izvršavaju nad podacima. Podaci se modeluju na osnovu stvarnih entiteta koji se pojavljuju u domenu problema: računi, klijenti, narudžbine,... Operacije, sa druge strane, nastoje da modeluju poslovne procese, ili pojedine korake datih procesa: kreiranje računa, dodavanje klijenta, slanje narudžbine,...

8.1. RAZLAGANJE POSLOVNOG SLOJA

Koji delovi bi se videli kada bi imali priliku da vertikalno razložimo BLL?

U BLL-u bi pronašli

- *objektni model* koji modeluje učesnike u poslovnom procesu,
- *poslovna pravila* koja predstavljaju klijentsku politiku i zahteve,
- *servise* koji implementiraju autonomne funkcionalnosti, i
- *proces rada* koji definiše kako se dokumenti i podaci prenose iz jednog modula u drugi, kao i između slojeva.

Sigurnost je takođe veoma bitna i mora biti primenjena u svim slojevima, ali posebno u BLL gde se kod ponaša kao "čuvar" trajnih podataka. Sigurnost u BLL-u u osnovi predstavlja sigurnost baziranu na privilegijama. Na ovaj način se pristup poslovnim objektima ograničava samo na autorizovane korisnike.

8.1.1. OBJEKTNI MODEL DOMENA

Svrha objektnog modela domena je da pruži strukturni pogled na čitav sistem, uključujući i funkcionalne opise entiteta, njihovih relacija i odgovornosti. Model je kreiran na osnovu korisnikovih zahteva, a dokumentuje se pomoću UML jezika i to posebno pomoću dijagrama slučajeva korišćenja i klasnih dijagrama. Entiteti domena ukazuju na realne elemente iz stvarnog sveta koji poseduju određene podatke i operacije. Svaki entitet ima određenu ulogu u modelu i obezbeđuje ukupno željeno ponašanje

sistema. Entiteti poseduju svoje odgovornosti i vrše interakciju sa ostalim entitetima putem definisanog skupa relacija.

Veliki broj aplikacija koje se obično označavaju kao komplekse, su u suštini relativno jednostavne ako se posmatraju tehnički izazovi koji one postavljaju. Opšta slika da su date aplikacije kompleksne obično proističe iz nasleđene kompleksnosti problema koji one modeluju. Obično se ova kompleksnost svodi na kreiranje odgovarajućeg softverskog modela poslovnog procesa, a ne na njegovu implementaciju. Sa dobro dizajniranim modelom, moguće je rešiti bilo koji nivo kompleksnosti sa razumno uloženim trudom.

Sa spoljašnjeg stanovišta, BLL se može posmatrati kao mašinerija koja radi nad poslovnim objektima. U većini slučajeva, poslovni objekti (BO - Business Objects) predstavljaju implementaciju entiteta iz domena; drugim rečima predstavljaju klase koje obuhvataju podatke i ponašanje. U ostalim slučajevima, to su pomoćne klase koje vrše neka posebna izračunavanja. BLL određuje kako poslovni objekti međusobno vrše interakciju. On takođe zahteva pravila i tokove rada za module koji vrše interakciju i menjaju poslovne objekte.

BLL zauzima središnji segment slojevitog sistema i vrši razmenu informacija sa prezentacionim slojem i slojem za pristup bazi podataka. Ulaz i izlaz za BLL sloj ne moraju da budu poslovni objekti. U mnogim slučajevima, arhitektura radije koristi *objekte za prenos podataka* (DTO - Data Transfer Objects) koji prenose podatke kroz slojeve.

Koja je razlika između BO i DTO. Poslovni objekti sadrže i podatke i ponašanje, pa se mogu smatrati potpuno-označenim aktivnim objektima koji učestvuju u logici domena. Objekti za transfer podataka predstavljaju vrstu objekata sa vrednostima, odnosno, prost kontejner koji sadrži samo podatke ali ne i ponašanja. Podaci koji se nalaze u instancama poslovnih objekata se obično kopiraju u objekte za transfer podataka zbog serijalizacije. DTO objekti ne sadrže logička ponašanja, osim *get* i *set* svojstva. Za svaku klasu koja predstavlja entitet u modelu može postojati više objekata za prenos podataka. Ovo je zbog toga što DTO ne predstavlja jednostavnu kopiju objekta iz domena koja samo ne sadrži ponašanja. Umesto toga, to je objekat koji predstavlja podskup određenog objekta iz domena, a koji se koristi u određenom kontekstu. Npr. u jednoj metodi je potreban DTO za entitet klijenta koji sadrži samo ime kompanije i ID. U drugom delu koda je potreban DTO koji sadrži ime kompanije, ID, zemlju i kontakt informacije. U opštem slučaju, objekat domena predstavlja graf objekata (npr klijent uključuje narudžbine, koje uključuju detalje o narudžbinama, itd.).

8.1.2. POSLOVNA PRAVILA

Svaka organizacija u svetu je bazirana na skupu poslovnih pravila koja se kriste da bi se implementirale njene strategije. Strategije diktiraju gde se želi ići, dok poslovna pravila odgovaraju na pitanja kako doći do tamo.

Postoji više načina da se formulišu poslovna pravila. U zavisnosti od oblasti u kojoj se radi, poslovna pravila mogu biti veoma promenljiva. Ovo znači da bi BLL pravila trebalo implementirati na veoma fleksibilan način, poželjno kroz *mehanizam pravila* (rules engine). Na najvišem nivou apstrakcije, mehanizam pravila je deo softvera koji prihvata pravila u nekom formalnom formatu, a zatim ih primenjuje na poslovne objekte.

U praktičnom smislu, poslovna pravila su najčešće skup IF-THEN-ELSE naredbi koja su ručno mapirana u metode poslovnih objekata, kreirajući na taj način *logiku domena* ili *poslovnu logiku*.

8.2. PATERNI ZA KREIRANJE POSLOVNOG SLOJA

BLL je grupisan oko objektnog modela domena, odnosno grafa objekata koji modeluje entitete u domenu. Svaki poslovni objekat ima sopstvene podatke i ponašanje. Nakon kreiranja dijagrama klasa i upoznavanja sa modelom, kako pristupiti zadatku kreiranja poslovnog sloja?

Postoji nekoliko dizajn paterni za organizaciju logike domena, pri čemu svaki od njih poseduje drugačiji nivo kompleksnosti i drugačije ciljeve. Ovi paterni se mogu svrstati u nekoliko kategorija.

Prilikom dizajna BLL, može se primeniti ad hok pristup, međutim, istorijski gledano, nivo uspešnosti ovakvog pristupa nije značajan. Iz tog razloga se uvode paterni jer oni pomažu da se kompleksnost razume i da se sa njom uspešno izađe na kraj. Svaki patern poseduje svoje prednosti, ali i svoje slabosti. Može se reći da patern ima arhitektonske prednosti u odnosu na cenu.

Izbor paterna se ne bazira uvek na njegovoj ceni. Umesto toga, izbor je često baziran na bliskom poznavanju poslovnog domena. Neki paterni su odlični za relativno jednostavne aplikacije, ali na njima ne treba bazirati složene aplikacije.

Dakle, fokus je na razumevanju domena aplikacije, razumevanju skupa dostupnih i poznatih paterna, i odabira paterna koji će, prema našem mišljenju, dati najbolji odgovor na kompleksnost posmatranog problema.

8.2.1. PROCEDURALNI PATERNI

Pre uzleta koji je donelo objektno orjentisano programiranje, poslovni sloj je predstavljala kolekcija procedura, pri čemu je svaka od njih sadržala jedan zahtev iz prezentacionog sloja. Dobrim dizajnom se smatrala organizacija procedura na takav način da se u maksimalnoj meri ukloni suvišni kod, a da pri tome svi zahtevi budu u potpunosti ispunjeni.

Poslovni sloj se posmatrao kao niz povezanih transakcija. Počevši od dijagrama slučajeva korišćenja, svaki korak sa kojim je sistem trebao da se izbori je bio deljen u sitnije korake. Svaki korak je na kraju bio modelovan kroz jednu operaciju, koja se najčešće nazivala transakcija.

U ovu grupu paterna spada *skripta transakcija* (TS - Transaction Script).

Ukoliko je potrebno da se pošalje narudžbina za neku robu, logička transakcija je sačinjena od koraka kao što su provera da li je data roba dostupna, računanje ukupne cene, provera sredstava na računu korisnika, ažuriranje baze podataka i interakcija sa dostavljačima.

Istorijski posmatrano, TS je bio prvi patern koji je našao široku primenu. Tokom godina se pojavio još jedan sličan pristup pod imenom *modul tabele* (Table Module). Gledište je slično kao i kod TS, ali su sada operacije grupisane po podacima u tabelama. Operacije su definisane kao metode u objektu koji predstavlja tabelu slogova. Ovo je i dalje proceduralni pristup, ali sa jačom objektno orjentisanom vizijom.

8.2.2. OBJEKTNO BAZIRANI PATERNI

Potpuno objektno orjentisani patern vrši organizovanje logike domena kao grafa međusobno povezanih objekata, pri čemu svaki predstavlja entitet ili tačku od interesa i poseduje podatke i ponašanje. Koliko je blizu ovaj objektni model modelu podataka predstavljenom pomoću baze? To zavisi od kompleksnosti i potrebnog nivoa apstrakcije.

Veoma jednostavni objektni modeli dosta podsećaju na modele podataka predstavljene bazom. U ovom slučaju, objekti koji formiraju model su u suštini slogovi sa nekim dodatim metodama. Ovaj patern se u opštem slučaju naziva *patern aktivnog sloga* (Active Record Pattern).

Što se više apstrakcije dodaje u model, to se on više udaljava od modela podataka. Da bi se kreirao objektni model baziran na domenu, najčešće se polazi od vizije domena koji programer poseduje, a ne od strukture baze podataka. Dizajn vođen domenom nesumnjivo vodi ka postojanju značajne razlike između modela podataka i rezultujućeg modela domena. Ovaj patern se najčešće naziva *domen model patern* (Domain Model Pattern).

Dizajniranje modela koji predstavlja osnov za softverski proizvod koji u potpunosti zadovoljava korisničke zahteve je delikatan posao. Iako postoje zabeležene metrike i praksa, pronalaženje odgovarajućeg modela je najviše stvar iskustva.

8.3. GRUPE DIZAJN PATERNA

Dvadesettri dizajn paterna su okarakterisana u GoF dizajn patern knjizi. Svaki patern je smešten u jednu od tri podkategorije: paterni za kreiranje objekata (Creational), strukturni paterni (Structural) ili paterni ponašanja (Behavioral). Kroz ova predavanja će biti razmatrani oni paterni koji su bitni za razvoj ASP.NET aplikacija.

8.3.1. CREATIONAL DIZAJN PATERNI

Creational paterni se odnose na kreiranje objekata i referenciranje. Oni apstrakuju odgovornost kreiranja instanci objekata od klijenta. Na taj način, kod postaje slabo povezan, a odgovornost za kreiranje kompleksnih objekata se nalazi na jednom mestu što je u skladu sa SRP i SoC principima. U ovu grupu spadaju sledeći paterni:

- **Singleton** – omogućava da klasa bude instancirana jednom i da poseduje globalnu tačku pristupa
- **Factory** – omogućava da klasa delegira odgovornost za kreiranje odgovarajućeg objekta.
- **Abstract Factory** – obezbeđuje interfejs za kreiranje više povezanih objekata
- **Builder** – omogućava da različite verzije nekog objekta budu kreirane tako što razdvaja kreiranje samog objekta
- **Prototype** – omogućava da klase budu kopirane ili klonirane iz instance prototipa, a ne da se kreiraju nove instance

8.3.2. STRUCTURAL DIZAJN PATERNI

Structural paterni se odnose na sastavljanje i na relacije između objekata kako bi se ispunili zahtevi velikih sistema. U ovu grupu spadaju sledeći paterni:

- **Adapter** – omogućava klasama nekompatibilnih interfejsa da budu korišćene zajedno
- **Bridge** – razdvaja abstrakciju od njene implementacije, što omogućava da se apstrakcija i implementacija menjaju nezavisno jedna od druge
- **Composite** – omogućava grupama objekata koje predstavljaju hijerarhiju da budu tretirane na isti način kao jedna instanca objekta
- **Decorator** – omogućava da se dinamički okruži klasa i da se proširi njeno ponašanje
- **Facade** – omogućava jednostavan interfejs i kontroliše pristup nizu komplikovanih interfejsa i podsistema
- **Flyweight** – omogućava da se na efikasan način dele podaci između više malih klasa
- **Proxy** – obezbeđuje skladište za kompleksniju klasu čije instanciranje je skupo

8.3.3. BEHAVIORAL DIZAJN PATERNI

Behavioral paterni se odnose na komunikaciju između objekata u smislu odgovornosti i algoritama. Paterni u ovoj grupi enkapsuliraju kompleksno ponašanje i vrše njegovu apstrakciju od toka kontrole u sistemu. Na ovaj način kompleksni sistemi postaju lako razumljivi i jednostavni za održavanje. U ovu grupu spadaju sledeći paterni:

- **Chain of Responsibility** – omogućava komandama da se zajedno dinamički menjaju kako bi odgovorile na zahtev

- **Command** – enkapsulira metodu kao objekat i razdvaja izvršavanje komande od pozivaoca
- **Interpreter** – određuje kako oceniti rečenice u jeziku
- **Iterator** – omogućava kretanje kroz kolekciju na formalizovan način
- **Mediator** – definiše objekat koji omogućava komunikaciju između druga dva objekta, pri čemu oni ne znaju jedan za drugi
- **Memento** – omogućava vraćanje objekta u prethodno stanje
- **Observer** – definiše način na koji jedna ili više klasa treba da budu upozerenе na promene u drugoj klasi
- **State** – omogućava da objekat promeni svoje ponašanje tako što delegira na posebna i promenljiva stanja objekta
- **Strategy** – omogućava da određeni algoritam bude enkapsuliran unutar klase i da bude zamenjen tokom izvršavanja kako bi promenio ponašanje objekta
- **Template Method** – definiše kontrolu toka algoritma ali omogućava podklasama da preklope ili da implementiraju tokove izvršavanja
- **Visitor** – omogućava novoj funkcionalnosti da bude izvršena nad klasom bez uticaja na njenu strukturu

8.4. KAKO ODABRATI I PRIMENITI DIZAJN PATTERN

ostoji veliki broj paterna, pa se postavlja pitanje kako odabrati odgovarajući? Da bi se znalo koji dizajn patern odabrati i kako primeniti šablon rešenja na dati specifični problem, neophodno je razumeti sledeća uputstva:

- Paterni se ne mogu primeniti ukoliko ne znamo za njih. Prvi korak, koji je veoma važan, je da se proširi znanje i da se prouče paterni i principi kako u apstraktnoj, tako i u konkretnoj formi. Patern se može primeniti na veliki broj načina. Što se više različitih implementacija paterna vidi, to je lakše razumeti nameru paterna i kako jedan patern može imati različite implementacije
- Da li je potrebno uvesti kompleksnost dizajn paterna? Uobičajeno je da inženjeri pokušavaju da upotrebe patern kako bi rešili svaki problem, pogotovu kada tek uče o paternima. Uvek je potrebno proračunati dodatno vreme koje zahteva implementacija paterna i benefit koji će patern dati.
- Potrebno je generalizovati problem i identifikovati probleme na apstraktan način. Nakon toga se razmatra namera svakog paterna i princip kako bi se videlo da li trenutni problem odgovara problemu koji dati patern ili princip pokušavaju da reše. Treba imati na umu da su dizajn paterni rešenja na visokom nivou apstrakcije, pa je samim tim neophodno izvršiti apstrakciju trenutnog problema bez prevelikog razmatranja detalja.
- Treba razmatrati paterne koji imaju sličnu prirodu i paterne koji su u istoj grupi. To što smo neki patern ranije koristili ne znači da će to ubek biti pravi izbor za rešenje problema.
- Enkapsulirati razlike. Treba posmatrati šta će se verovatno promeniti u okviru aplikacije. Ukoliko znamo da će se specijalni popusti koje kuća daje menjati tokom vremena, treba tražiti patern koji će nam omogućiti promenu bez da moramo da menjamo ostatak aplikacije.
- Kada je patern odabran, treba osigurati upotrebu jezika paterna u skladu sa jezikom domena problema kada se dodeljuju imena učesnicima u rešenju. Na primer, ako se koristi strategy patern kako bi se kreiralo rešenje za cenu različitih dostava, onda ime treba kreirati FedExShippingCostStrategy. Korišćenjem uobičajenog patern rečnika zajedno sa jezikom

domena, napisani kod će postati lakši za čitanje i razumevanje i drugim inženjerima koji poseduju znanje paterna.

Kada se radi o dizajn paternima, ne postoji zamena za učenje. Što se više zna o svakom dizajn paternu, to je inženjer bolje obučen i moći će bolje da ih primeni.

9. STRUKTURNI DIZAJN PATERNI

Strukturni dizajn paterni imaju ulogu da unesu fleksibilnost, sigurnost i da produže život softverske aplikacije. Oni se baziraju na to kako su klase i objekti sastavljeni u cilju kreiranja većih struktura. Oni se mogu primeniti dok se sistem dizajnira, ili kasnije tokom održavanja i proširivanja.

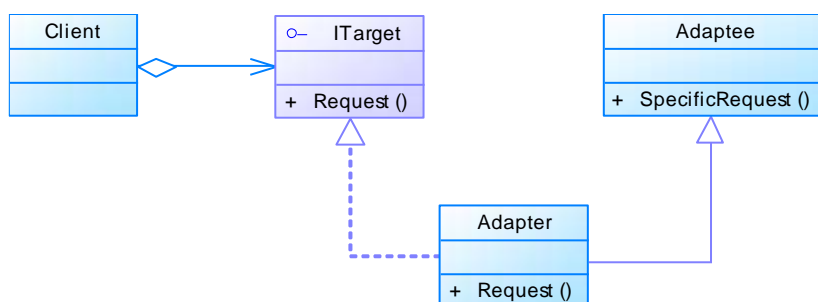
9.1. ADAPTER PATTERN

Adapter patern radi kao most koji povezuje dva interfejsa koji međusobno nisu kompatibilni. Ovaj patern spada u grupu strukturnih dizajn paternata jer kombinuje mogućnosti dva nezavisna interfejsa.

Patern uključuje jednu klasu koja je odgovorna za kombinovanje funkcionalnosti nezavisnih i nekompatibilnih interfejsa. Kao primer, možemo posmatrati čitač kartica koji se ponaša kao adapter između memorijske kartice i laptop-a. Da bi očitali memorijsku karticu, mi je ubacimo u čitač kartica, koji se nalazi u okviru laptop-a.

9.1.1. UML DIJAGRAM ADAPTER PATERNA

Važan doprinos Adapter paternata jeste u tome što promovise programiranje ka interfejsima. Klijent radi sa kodom koji je vezan za domen problema i koji je naveden u okviru *ITarget* interfejsa. Klasa *Adaptee* sadrži traženu funkcionalnost ali sa drugačijim interfejsom. Klasa *Adapter* implementira *ITarget* interfejs i prosleđuje pozive od *Client* klase ka klasi *Adaptee*. Pri tome on menja parametre i povratne tipove, kako bi ispunio zahteve. *Client* je svestan samo da postoji interfejs *ITarget*. Ovaj patern je prikazan UML dijagramom na slici 9.1.



Slika 9.1 - UML dijagram za Adapter patern

U prikazanom primeru imamo "class adapter" jer klasa *Adapter* implementira interfejs *ITarget* a nasleđuje klasu *Adaptee*. Kao alternativa nasleđivanju iz klase *Adaptee* jeste da se agregiše klasa *Adaptee*. Na ovaj način se dobija "object adapter". Razlike u dizajnu su pre svega u tome da li je potrebno potpuno preklopiti ponašanje iz klase *Adaptee* ili ga je potrebno samo proširiti. Ako je potrebno proklopiti ponašanje, koristimo *class adapter*, a ako je potrebno proširiti ponašanje koristimo *object adapter*.

Svrha *ITarget* interfejsa jeste da omogući objektima iz tipova koji su adaptirani (*Adaptee*) da se koriste na istim mestima sa drugim objektima koji implementiraju isti taj interfejs. Međutim, adaptirani objekti možda nemaju iste metode kao što je to definisano u interfejsu *ITarget*, pa sama upotreba interfejsa nije dovoljna. Zbog toga nam treba Adapter patern. Klasa *Adaptee* nudi slične funkcionalnosti kao što to definiše metoda *Request*, ali pod drugačijim imenom i sa moguće drugačijim parametrima. Klasa *Adaptee* je potpuno nezavisna od drugih klasa, uključujući i imenovanja koja se u njima koriste.

U okviru paternata imamo sledeće elemente:

- *ITarget* - interfejs koji *Client* želi da koristi
- *Adaptee* - implementacija koju je neophodno prilagoditi (adaptirati)

- *Adapter* - klasa koja implementira *ITarget* interfejs u skladu sa klasom *Adaptee*
- *Request* - operacija koju *Client* traži
- *SpecificRequest* - implementacija *Request* funkcionalnosti u okviru *Adaptee* klase

9.1.2. DEMO PRIMER

Adapter primer ćemo ilustrovati na jednostavnom primeru. Pretpostavimo da se tehnička očitavanja skupljaju i prikazuju sa velikim stepenom tačnosti, ali klijent želi da koristi samo gruba očitavanja. Interfejs bi radio sa celim brojevima, a stvarna implementacija sa brojevima koji imaju dvostruku preciznost. U cilju kreiranja primera, kreiramo konzolnu aplikaciju pod imenom *Softversko2.Adapter*. Primer je prikazan u okviru listinga 9.1.

```
namespace Softversko2.Adapter
{
    // Existing way requests are implemented
    class Adaptee
    {
        // Provide full precision
        public double SpecificRequest(double a, double b)
        {
            return a / b;
        }
    }

    // Required standard for requests
    interface ITarget
    {
        // Rough estimate required
        string Request(int i);
    }

    // Implementing the required standard via Adaptee
    class Adapter : Adaptee, ITarget
    {
        public string Request(int i)
        {
            return "Rough estimate is " + (int)Math.Round(SpecificRequest(i, 3));
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Showing the Adaptee in standalone mode
            Adaptee first = new Adaptee();
            Console.Write("Before the new standard\nPrecise reading: ");
            Console.WriteLine(first.SpecificRequest(5, 3));

            // What the client really wants
            ITarget second = new Adapter();
            Console.WriteLine("\nMoving to the new standard");
            Console.WriteLine(second.Request(5));

            Console.ReadKey();
        }
    }
}
```

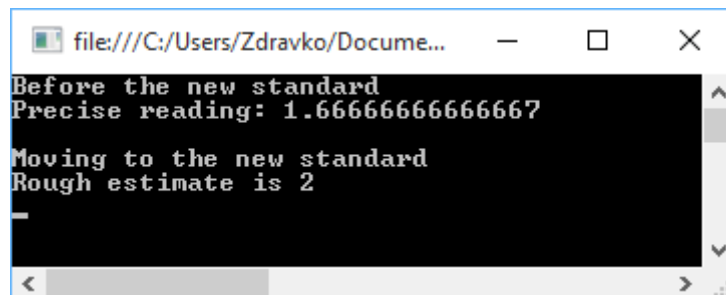
Listing 9.1 - Kod u okviru primera za Adapter patern

U okviru *main* metode vidimo dva načina upotrebe. U prvom, vidimo kako se klasa *Adaptee* može direktno upotrebiti i pozvati. Međutim, klijent želi da na drugi način koristi ovu metodu. Klasa *Adapter* implementira interfejs *ITarget* i nasleđuje klasu *Adaptee*. Zbog toga, ona može da primi poziv ka metodi *Request* (u potpisu string-int) i da poziv prosledi ka metodi *SpecificRequest* (sa potpisom double-double-double).

Karakteristika klasa koje predstavljaju adaptere jeste da mogu da dodaju dodatno ponašanje između *ITarget* interfejsa i *Adaptee* klasa. Drugim rečima, oni ne moraju da budu nevidljivi klijentima. U prethodnom primeru, *Adapter* dodaje tekst "Rough estimate is" kako bi ukazao da je metoda *Request* izmenjena pre poziva metode *SpecificRequest*.

Klase *Adapter* mogu dodati proizvoljnu količinu funkcionalnosti na *Adaptee* implementaciju u okviru *ITarget* interfejsa. Najjednostavnija implementaciju uključuje jednostavno pozivanje metode iz klase *Adaptee* sa odgovarajućim parametrima.

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 9.2.



Slika 9.2 - Ispis nakon pokretanja aplikacije

9.1.3. REALNI PRIMER

Pretpostavimo da je kompanija ugradila kontrolni sistem za revolucionarni vođeni avion *Seabird*. Avion ima u sebi kombinaciju dizajna aviona i broda: poseduje telo aviona ali kontrole i motor broda. Kompanija koja proizvodi *Seabird* je radila adaptiranja i prilagođenja u maksimalnoj meri kako bi omogućila kontrolu letelice koja je izgrađena od brodskih delova.

U okviru primera, imaćemo dvostruki adapter između klasa *Aircraft* i *Seacraft*. Kada se pokreću eksperimenti nad *Seabird*-om, kompanija će koristiti adapter, što će joj omogućiti da pristupi metodama i podacima u obe klase. Drugim rečima, *Seabird* će se ponašati u isto vreme kao letelica (*Aircraft*) i kao plovilo (*Seacraft*).

Da bi demonstrirali upotrebu ovog patern, kreiraćemo novu konzolnu aplikaciju u okviru Visual Studio-a pod imenom *Softversko2.Adapter.Craft*. U okviru projekta dodajemo interfejs *IAircraft* sa sledećim kodom:

```
namespace Softversko2.Adapter.Craft
{
    // ITarget interface
    public interface IAircraft
    {
        bool Airborne { get; }
        void TakeOff();
        int Height { get; }
    }
}
```

Listing 9.2 - Kod za interfejs *IAircraft*

Interfejs *IAircraft* poseduje dva svojstva: *Airborne* (da li je avion uzleteo) i *Height* (visina na kojoj se trenutno nalazi), kao i jednu metodu *TakeOff*. *IAircraft* za patern predstavlja *ITarget* interfejs.

Nakon toga dodajemo klasu *Aircraft* koja implementira *IAircraft* interfejs:

```

namespace Softversko2.Adapter.Craft
{
    // Target
    public sealed class Aircraft : IAircraft
    {
        int height;
        bool airborne;

        public Aircraft()
        {
            height = 0;
            airborne = false;
        }

        public void TakeOff()
        {
            Console.WriteLine("Aircraft engine takeoff");
            airborne = true;
            height = 200; // Meters
        }

        public bool Airborne
        {
            get { return airborne; }
        }

        public int Height
        {
            get { return height; }
        }
    }
}

```

Listing 9.3 - Kod u okviru klase *Aircraft*

Sledeći korak je dodavanje interfejs *ISecraft.cs* i klase *Seacraft.cs* koja ga implementira.

```

namespace Softversko2.Adapter.Craft
{
    // Adaptee interface
    public interface ISecraft
    {
        int Speed { get; }
        void IncreaseRevs();
    }
}

```

Listing 9.4 - Kod u okviru interfejsa *ISecraft*

```

namespace Softversko2.Adapter.Craft
{
    // Adaptee implementation
    public class Seacraft : ISecraft
    {
        int speed = 0;

        public virtual void IncreaseRevs()
        {
            speed += 10;
            Console.WriteLine("Seacraft engine increases revs to " +
                               speed + " knots");
        }

        public int Speed
        {

```

```

        get { return speed; }
    }
}

```

Listing 9.5 - Kod u okviru klase *Seacraft*

Interfejs *ISeacraft* poseduje jedan properti *Speed* (trenutna brzina kojom se plovilo kreće) i jednu metodu *IncreaseRevs* koja služi da poveća broj obrtaja motora kako bi se povećala brzina. . Ove metode implementira *Seacraft* klasa. *ISeacraft* predstavlja *IAdaptee* interfejs za adapter patern.

Nakon ovoga dodajemo klasu *Seabird.cs* sa sledećim kodom:

```

namespace Softversko2.Adapter.Craft
{
    // Adapter
    public class Seabird : Seacraft, IAircraft
    {
        int height = 0;

        // A two-way adapter hides and routes the Target's methods
        // Use Seacraft instructions to implement this one
        public void TakeOff()
        {
            while (!Airborne)
                IncreaseRevs();
        }

        // Routes this straight back to the Aircraft
        public int Height
        {
            get { return height; }
        }

        // This method is common to both Target and Adaptee
        public override void IncreaseRevs()
        {
            base.IncreaseRevs();
            if (Speed > 40)
                height += 100;
        }

        public bool Airborne
        {
            get { return height > 50; }
        }
    }
}

```

Listing 9.6 - Kod u okviru klase *Seabird*

U okviru paterna, Adapter klasa je nasleđena iz Adaptee klase (*Seacraft*) i implementira *ITarget* interfejs (*IAircraft*). Adapter onda mora da uradi određene izmene kako bi povezao različite interfejse.

Poslednji korak je izmena u okviru metode *main* u klasi *Program* kako bi se pozivale odgovarajuće metode i kako bi se videla upotreba Adapter paterna.

```

namespace Softversko2.Adapter.Craft
{
    class Program
    {
        static void Main(string[] args)
        {
            // No adapter
            Console.WriteLine("Experiment 1: test the aircraft engine");
        }
    }
}

```

```

        IAircraft aircraft = new Aircraft();
        aircraft.TakeOff();
        if (aircraft.Airborne)
            Console.WriteLine("The aircraft engine is fine, flying at "
                               + aircraft.Height + "meters");

        // Classic usage of an adapter
        Console.WriteLine("\nExperiment 2: Use the engine in the Seabird");
        IAircraft seabird = new Seabird();
        seabird.TakeOff(); // And automatically increases speed
        Console.WriteLine("The Seabird took off");

        // Two-way adapter: using seacraft instructions on an IAircraft object
        // (where they are not in the IAircraft interface)
        Console.WriteLine("\nExperiment 3: Increase the speed of the Seabird:");
        (seabird as ISeacraft).IncreaseRevs();
        (seabird as ISeacraft).IncreaseRevs();
        if (seabird.Airborne)
            Console.WriteLine("Seabird flying at height " + seabird.Height +
                               " meters and speed " + (seabird as ISeacraft).Speed + " knots");

        Console.WriteLine("Experiments successful; the Seabird flies!");

        Console.ReadKey();
    }
}

```

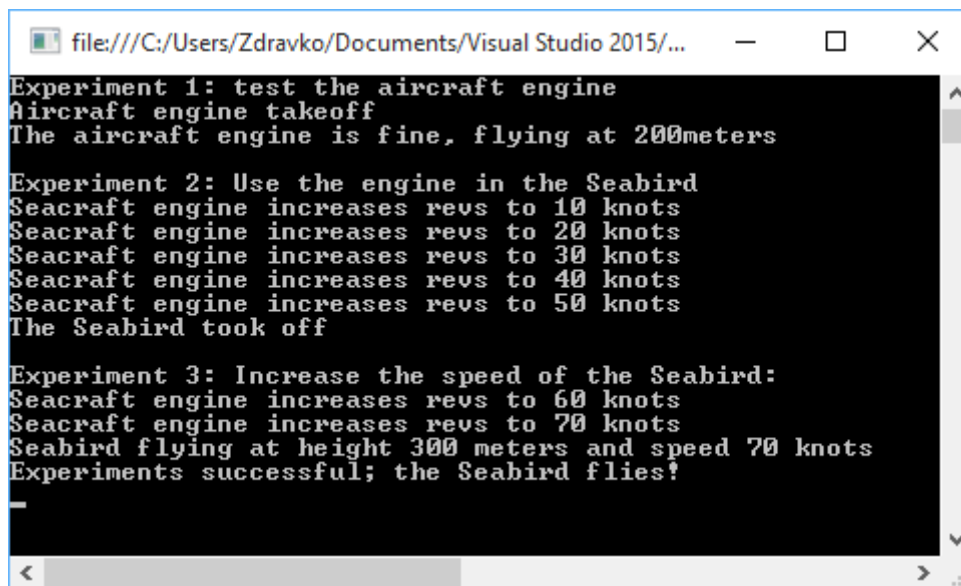
Listing 9.7 - Kod u okviru klase *Program*

Klase koje predstavljaju različite delove nude različite metode: *TakeOff* za letelicu i *IncreaseRevs* za plovilo. U jednostavnom adapteru, radila bi samo metoda *TakeOff*. U okviru ovog, dvostrukog adaptera, mi pristupamo i metodi *IncreaseRevs* koju adaptiramo kako bi uključili inofrmacije koje bi inače bile dostupne od strane Target klase.

Dvostruki adapter takođe rešava probleme sa varijablama, odnosno sa *Airborne*, *Speed* i *Height*. Varijable iz klase *Aircraft* (Target) su adaptirane kako bi čuvale lokalne informacije. Varijable u klasi *Seacraft* (Adaptee) rutiramo.

Rezultat svega prethodnog je prikazan kao eksperiment 3 u okviru *main* metode. Pozivi ka *seabird.Airborne* i *seabird.Height* predstavljaju regularne adapter metode. Međutim, mogućnost da tretiramo *Seabird* kao *Seacraft* je mogućnost koju unosi dvostruko adaptiranje.

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 9.3.



```
file:///C:/Users/Zdravko/Documents/Visual Studio 2015/...
Experiment 1: test the aircraft engine
Aircraft engine takeoff
The aircraft engine is fine, flying at 200meters

Experiment 2: Use the engine in the Seabird
Seacraft engine increases revs to 10 knots
Seacraft engine increases revs to 20 knots
Seacraft engine increases revs to 30 knots
Seacraft engine increases revs to 40 knots
Seacraft engine increases revs to 50 knots
The Seabird took off

Experiment 3: Increase the speed of the Seabird:
Seacraft engine increases revs to 60 knots
Seacraft engine increases revs to 70 knots
Seabird flying at height 300 meters and speed 70 knots
Experiments successful; the Seabird flies!
```

Slika 9.3 - Ispis dobijen nakon pokretanja aplikacije

9.2. COMPOSITE PATTERN

Composite patern se odnosi na sisteme koji poseduju dosta objekata. On ima široku promenu. Njegova funkcija je da se jedna komponenta i grupa komponenti mogu tretirati na isti način. Tipične operacije nad komponentama uključuju add, remove, display, find i group.

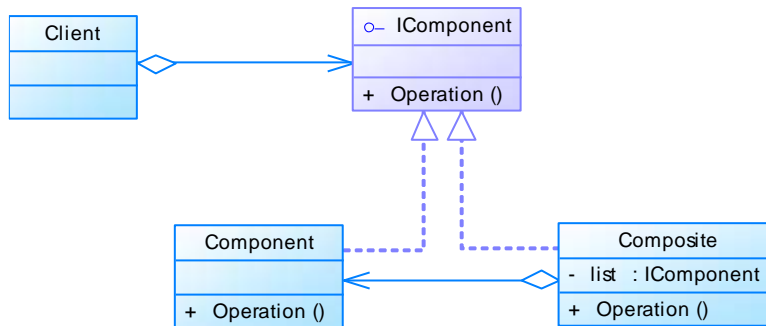
Aplikacije koje koriste grupisanje podataka su u velikoj upotrebi. Npr. možemo posmatrati aplikaciju za puštanje muzike (iTunes, Winamp, ...) ili aplikacije za kreiranje albuma sa slikama (Flickr ili iPhoto). Stavke se smeštaju u velike liste, koje se zatim na određeni način strukturiraju.

Ako posmatramo iPhoto, možemo videti da postoje različiti načini na koje možemo prikazivati slike koje su u njega unete: hronološki ili na osnovu imena događaja pod kojim su unete (npr. letovanje). Jedna fotografija se može pojaviti u okviru više albuma. Kreiranje albuma stvara kompozitni objekat, koji ne uključuje stvarno kopiranje fotografija na više lokacija. Važna osobina vezana za Composite patern jeste da operacije koje se odnose na fotografije i albume fotografija, treba da imaju ista imena i efekte, bez obzira na to da li se implementacije razlikuju. Npr. korisniku treba omogućiti da pogleda fotografiju, ali i da pogleda album (koji sadrži fotografije).

Composite patern jeste jedan od najjednostavnijih. On mora da radi sa dva tipa: komponentama (Components) i objektima sastavljenim iz komponenti (Composites). Oba tipa implementiraju interfejs sa zajedničkim osobinama. Composite objekti se sastoje od Components objekata. U većini slučajeva, operacije nad Composite objektima se implementiraju tako što se pozivaju ekvivalente operacije nad njihovim Component objektima.

9.2.1. UML DIJAGRAM COMPOSITE PATERNA

UML dijagram Composite paterna je prikazan na slici 9.4.



Slika 9.4 - UML dijagram za Composite patern

Osnovni gradivni elementi za Composite patern su:

- *IComponent* - definiše operacije koje će biti primenjene nad objektima oba tipa
- *Operation* - izvršava operacije nad objektima koji implementiraju *IComponent*
- *Component* - Izvršava operacije koje se mogu primentiti nad jednim objektom koji se ne može dalje dekomponovati
- *Composite* - izvršava operacije koje se mogu primeniti nad kompozitnim objektima koristeći listu komponenti koja se čuva lokalno, i u najčešćem slučaju, koristeći ekvivalentne operacije iz *Componet* objekta

Klijent komunicira samo sa *IComponent* interfejsom, što pojednostavljuje posao.

9.2.2. DEMO PRIMER

U cilju demonstracije Composite paterna, prvo ćemo kreirati demo primer koji ilustruje upotrebu ovog paterna. Kreiramo konzolnu aplikaciju u okviru Visual Studio-a *Softverko2.Composite*. U okviru ovog projekta u okviru klase *Program.cs* dodajemo sledeći kod:

```

namespace Softversko2.Composite
{
    class Program
    {
        interface IComponent
        {
            void Add(IComponent c);
            void Remove(IComponent c);
            void Display(int depth);
        }

        class Composite : IComponent
        {
            private List<IComponent> _children = new List<IComponent>();
            private string name;

            public Composite(string name)
            {
                this.name = name;
            }

            public void Add(IComponent c)
            {
                _children.Add(c);
            }

            public void Remove(IComponent c)
            {

```

```

        _children.Remove(c);
    }

    public void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);
        foreach (IComponent component in _children)
        {
            component.Display(depth + 2);
        }
    }
}

class Component : IComponent
{
    private string name;

    public Component(string name)
    {
        this.name = name;
    }

    public void Add(IComponent c)
    {
        Console.WriteLine("Cannot add to a leaf");
    }

    public void Remove(IComponent c)
    {
        Console.WriteLine("Cannot remove from a leaf");
    }

    public void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);
    }
}

static void Main(string[] args)
{
    // Create a tree structure
    Composite root = new Composite("root");
    root.Add(new Component("Leaf A"));
    root.Add(new Component("Leaf B"));

    Composite comp = new Composite("Composite X");
    comp.Add(new Component("Leaf XA"));
    comp.Add(new Component("Leaf XB"));

    root.Add(comp);
    root.Add(new Component("Leaf C"));

    // Add and remove a leaf
    Component leaf = new Component("Leaf D");
    root.Add(leaf);
    root.Remove(leaf);

    // Recursively display tree
    root.Display(1);

    // Wait for user
    Console.ReadKey();
}

```

```

    }
}
}

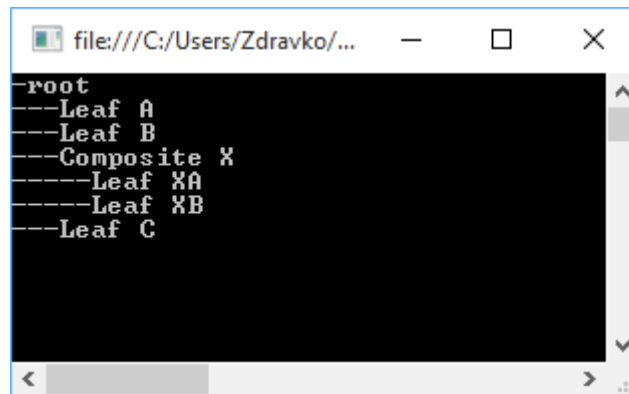
```

Listing 9.8 - Kod u okviru klase *Program*

Prvo se kreira interfejs *IComponent* koji sadrži tri metode: *Add*, *Remove* i *Display*. Ovo su metode koje će implementirati i klasa *Component* koja predstavlja jednu komponentu, kao i klasa *Composite* koja predstavlja kompoziciju komponenti.

Klasa *Composite* sadrži kao atribut listu objekata tipa *IComponent* kako bi kreirala kolekciju.

Nakon pokretanja prethodnog koda, dobijamo ispis koji je prikazan na slici 9.5.



Slika 9.5 - Ispis nakon pokretanja aplikacije

9.2.3. REALNI PRIMER

U okviru ovog primera ćemo kreirati primer sa slikama i albumima. Dizajn i implementacija Composite paterna bi bili isti i za druge primere jer su nezavisni od vrste komponenti za koju se on kreira. Na isti način bi se kreirao i primer koji grupiše ljude ili bankovne račune. Operacije nad kompozitnom klasom moraju da vrše iteraciju kroz strukturu komponenti. Da bi realizovali fleksibilnost komponenti bilo kog tipa i vezu između delova i grupe, možemo koristiti C# generičke klase.

Deklarisanjem *IComponent*, *Component* i *Composite* tipova kao generičkih, kreiramo implementaciju Composite paterna koji ćemo instancirati konkretnom klasom u okviru *main* metode.

U cilju kreiranja ovog projekta, u okviru Visual Studio-a kreiramo konzolnu aplikaciju pod imenom *Softversko2.Composite.Images*. U okviru aplikacije prvo dodajemo interfejs *IComponent* sa sledećim kodom:

```

namespace Softversko2.Composite.Images
{
    public interface IComponent<T>
    {
        void Add(IComponent<T> c);
        IComponent<T> Remove(T s);
        string Display(int depth);
        IComponent<T> Find(T s);
        T Name { get; set; }
    }
}

```

Listing 9.9 - Kod u okviru interfejsa *IComponent*

Interfejs *IComponent* je kreiran kao generički tipa *T*. U okviru njega kreiramo metode *Add*, *Remove* i *Find* koje koriste tip *IComponent<T>*, dok metode *Remove* i *Find* koriste tip *T*. Objekat tipa *IComponent* je ili pojedinačna komponenta (*Component*) ili kopozicija komponenti (*Composite*) i imaće atribut koji je tipa *IComponent*, pored stvarnog tipa elementa koji se skladišti. Element je tipa *T*, pa u slučaju metoda *Find* i *Remove*, mi prosleđujemo kao parametar stvarne vrednosti elementa koji je

tipa *T*. Npr. Ukoliko je interfejs instanciran tako da je tip *T* zapravo tip *Image*, mi ćemo proslediti objekat tipa *Image* ka metodi *Find*, a od metode ćemo dobiti referencu ka *IComponent* objektu. Objekat koji vraćamo će sadržati *Image* objekat ukoliko je pretraga bila uspešna.

Nakon toga kreiramo klasu *Component.cs* koja implementira interfejs *IComponent*.

```
namespace Softversko2.Composite.Images
{
    public class Component<T> : IComponent<T>
    {
        public T Name { get; set; }

        public Component(T name)
        {
            Name = name;
        }

        public void Add(IComponent<T> c)
        {
            Console.WriteLine("Cannot add to an item");
        }

        public IComponent<T> Remove(T s)
        {
            Console.WriteLine("Cannot remove directly");
            return this;
        }

        public string Display(int depth)
        {
            return new String('-', depth) + Name + "\n";
        }

        public IComponent<T> Find(T s)
        {
            if (s.Equals(Name))
                return this;
            else
                return null;
        }
    }
}
```

Listing 9.10 - Kod za klasu *Component*

Klasa *Component* implementira *IComponent* interfejs. Nemaju sve metode iz *IComponent* interfejsa svrhu u okviru *Component* klase. Dodavanje i uklanjanje imaju smisla samo kod kompozitne klase (*Composite.cs*), pa se prikazuje jednostavna poruka o grešci.

Nakon toga kreiramo klasu *Composite.cs*.

```
namespace Softversko2.Composite.Images
{
    public class Composite<T> : IComponent<T>
    {
        List<IComponent<T>> list;

        public T Name { get; set; }

        public Composite(T name)
        {
            Name = name;
            list = new List<IComponent<T>>();
        }
    }
}
```

```

    public void Add(IComponent<T> c)
    {
        list.Add(c);
    }

    IComponent<T> holder = null;

    // Finds the item from a particular point in the structure
    // and returns the composite from which it was removed
    // If not found, return the point as given
    public IComponent<T> Remove(T s)
    {
        holder = this;
        IComponent<T> p = holder.Find(s);
        if (p != null)
        {
            (holder as Composite<T>).list.Remove(p);
            return holder;
        }
        else
            return this;
    }

    // Recursively looks for an item
    // Returns its reference or else null
    public IComponent<T> Find(T s)
    {
        holder = this;
        if (Name.Equals(s)) return this;
        IComponent<T> found = null;
        foreach (IComponent<T> c in list)
        {
            found = c.Find(s);
            if (found != null)
                break;
        }
        return found;
    }

    // Displays items in a format indicating level in the composite structure
    public string Display(int depth)
    {
        StringBuilder s = new StringBuilder(new String('-', depth));
        s.Append("Set " + Name + " length : " + list.Count + "\n");
        foreach (IComponent<T> component in list)
        {
            s.Append(component.Display(depth + 2));
        }
        return s.ToString();
    }
}

```

Listing 9.11 - Kod u okviru klase *Composite*

Klasa *Composite.cs* također implementira *IComponent* interfejs. Metode *Find* i *Remove* su složenije od metoda sa istim imenom u okviru klase *Component*, kako bi mogle da rade sa proizvoljnim strukturama komponenti i kompozicija komponenti. Neke od karakteristika ove klase su:

- Klasa *Composite* čuva kao listu lokalnu strukturu koja se sastoji od objekata tipa *Componenti* i *Composite*. Kada su elementi tipa *Composite*, kreira se i novi objekat, ali i nova lista. Lista je deklarirana kao:

List<IComponent <T>> list;

Ovo govori da bilo koji generički tip može poslužiti kao parametar za drugi generički tip.

- U okviru metode *Remove* prvo se pronalazi element u okviru strukture, a zatim, ukoliko se on nalazi u strukturi, uklanjamo ga iz strukture koja se čuva lokalno u klasi *Composite*:

(holder as Composite<T>).list.Remove(p);

Varijabla *holder* je tipa *IComponent* pa se mora kastovati u tip *Composite* kako bi se moglo pristupiti njenom atributu *list*.

- Poziv ka metodi *Find* će otići ka odgovarajućoj metodi u zavisnosti od stvarnog tipa u vreme izvršavanja. Ovo je u skladu sa logikom Composite paterna da se objekti tipa *Component* i *Composite* tretiraju na isti način.

Poslednji korak je izmena koda u okviru *main* metode u klasi *Program.cs*.

```
namespace Softversko2.Composite.Images
{
    class Program
    {
        static void Main(string[] args)
        {
            IComponent<string> album = new Composite<string>("Album");
            IComponent<string> point = album;

            // Create and manipulate a structure
            Console.WriteLine("\t\t AddSet Home");
            IComponent<string> home = new Composite<string>("Home");
            album.Add(home);
            point = home;

            Console.WriteLine("\t\t AddPhoto Dinner.jpg");
            point.Add(new Component<string>("Dinner.jpg"));

            Console.WriteLine("\t\t AddSet Pets");
            IComponent<string> pets = new Composite<string>("Pets");
            point.Add(pets);
            point = pets;

            Console.WriteLine("\t\t AddPhoto Dog.jpg");
            point.Add(new Component<string>("Dog.jpg"));
            Console.WriteLine("\t\t AddPhoto Cat.jpg");
            point.Add(new Component<string>("Cat.jpg"));

            Console.WriteLine("\t\t Find Album");
            point = album.Find("Album");

            Console.WriteLine("\t\t AddSet Garden");
            IComponent<string> garden = new Composite<string>("Garden");
            point.Add(garden);
            point = garden;

            Console.WriteLine("\t\t AddPhoto Spring.jpg");
            point.Add(new Component<string>("Spring.jpg"));
            Console.WriteLine("\t\t AddPhoto Summer.jpg");
            point.Add(new Component<string>("Summer.jpg"));
            Console.WriteLine("\t\t AddPhoto Flowers.jpg");
            point.Add(new Component<string>("Flowers.jpg"));
            Console.WriteLine("\t\t AddPhoto Trees.jpg");
            point.Add(new Component<string>("Trees.jpg"));

            Console.WriteLine("\t\t Display");
```

```

        Console.WriteLine(album.Display(0));

        Console.WriteLine("\t\t Find Pets");
        point = album.Find("Pets");

        Console.WriteLine("\t\t Find Garden");
        point = album.Find("Garden");

        Console.WriteLine("\t\t Remove Flowers.jpg");
        point = point.Remove("Flowers.jpg");

        Console.WriteLine("\t\t AddPhoto BetterFlowers.jpg");
        point.Add(new Component<string>("BetterFlowers.jpg"));

        Console.WriteLine("\t\t Display");
        Console.WriteLine(album.Display(0));

        Console.WriteLine("\t\t Find Home");
        point = album.Find("Home");

        Console.WriteLine("\t\t Remove Pets");
        point = point.Remove("Pets");

        Console.WriteLine("\t\t Display");
        Console.WriteLine(album.Display(0));

        Console.ReadKey();
    }
}

```

Listing 9.12 - Kod u okviru klase *Program*

U okviru ovog primera mi dodajemo imena slika u kolekciju. U okviru primera ne koristimo stvarne slike već samo njihova imena. Prilikom korišćenja programa, jedan od glavnih problema je da odredimo gde se trenutno nalazimo. Na početku kreiramo prazan skup pod imenom "Album". U okviru aplikacije koristimo sledeće komande:

- *AddSet* - dodajemo novi prazan skup sa određenim imenom
- *AddPhoto* - dodajemo novu sliku sa određenim imenom
- *Find* - pronalazi komponentu pod određenim imenom (skup ili sliku)
- *Remove* - uklanja komponentu pod određenim imenom (skup ili sliku) i pozicioniramo se na skup iz kog je izvršeno uklanjanje
- *Display* - prikazuje celu strukturu

Iz prethodnog možemo videti da metode *Find* i *Remove* rade i sa tipom *Component* i sa tipom *Composite*.

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 9.6.

```
file:///C:/Users/Zdravko/Documents/Visual Studio 2015/Projects...
AddSet Home
AddPhoto Dinner.jpg
AddSet Pets
AddPhoto Dog.jpg
AddPhoto Cat.jpg
Find Album
AddSet Garden
AddPhoto Spring.jpg
AddPhoto Summer.jpg
AddPhoto Flowers.jpg
AddPhoto Trees.jpg
Display
Set Album length :2
--Set Home length :2
----Dinner.jpg
----Set Pets length :2
-----Dog.jpg
-----Cat.jpg
--Set Garden length :4
----Spring.jpg
----Summer.jpg
----Flowers.jpg
----Trees.jpg

Find Pets
Find Garden
Remove Flowers.jpg
AddPhoto BetterFlowers.jpg
Display
Set Album length :2
--Set Home length :2
----Dinner.jpg
----Set Pets length :2
-----Dog.jpg
-----Cat.jpg
--Set Garden length :4
----Spring.jpg
----Summer.jpg
----Trees.jpg
----BetterFlowers.jpg

Find Home
Remove Pets
Display
Set Album length :2
--Set Home length :1
----Dinner.jpg
--Set Garden length :4
----Spring.jpg
----Summer.jpg
----Trees.jpg
----BetterFlowers.jpg
```

Silka 9.6 - Ispis nakon pokretanja aplikacije *Images* za ilustraciju Composite paterna

9.3. DECORATOR PATTERN

Uloga Decorator paterna jeste da dinamički omogući dodavanje novih stanja (podataka članova) i ponašanja (metoda) postojećem objektu. Postojeći objekat ne zna da se nad njim primenjuje Decorator patern, odnosno ne zna da mu se dodaju nova stanja i ponašanja. Zbog toga je ovaj patern veoma koristan prilikom izmena u okviru postojećeg sistema. Ključna tačka implementacije kod ovog paterna jeste da "dekorator" objekti u isto vreme i implementira isti interfejs kao i originalna klasa i sadrži instancu njegove implementacije (originalnu klasu).

Osnovna karakteristika ovaog paterna jeste da se ne oslanja na nasleđivanje osnovne klase kao bi joj se dodala nova stanja i ponašanja. Ako bi nova klasa trebala da nasledi postojeću klasu, kako bi dodala jednu ili dve nove metode, ona bi sadržala i sve što već postoji u originalnoj klasi, pa bi tako postala dosta "velik" objekat. Umesto toga, nova klasa implementira isti interfejs kao i originalna klasa i dodaje nova ponašanja, što znači da je ovakav objekat "mali". Nova klasa može da:

- implementira bilo koju metodu iz interfejsa, i da na taj način promeni ponašanje originalne klase
- doda nova stanja i ponašanja
- pristupi bilo kojoj javnoj metodi originalne klase koja joj se prosleđuje kroz konstruktor

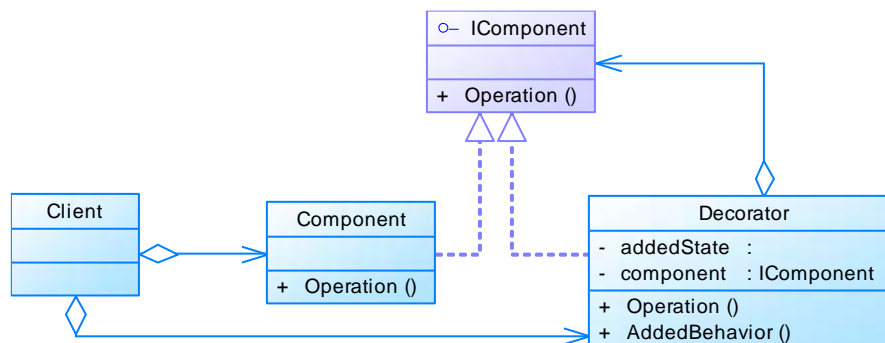
Prednosti ovog paterna su:

- originalni objekat nije svestan da postoji bilo koji "dekorator" objekat
- ne postoji jedna velika klasa koja sadrži sve funkcionalnosti i stanja u sebi
- svaki novi "dekorator" objekat je nezavistan od prethodnih

9.3.1. UML DIJAGRAM DECORATOR PATERNA

Učesnici u Decorator paternu su prikazani pomoću UML dijagrama na slici 9.7. Osnovni delovi UML dijagrama su:

- *Component* - originalna klasa za objekat kojem ćemo dodavati ili modifikovati stanja i ponašanja.
- *IComponent* - interfejs koji identifikuje klase objekata koji se mogu "dekorisati"
- *Decorator* - klasa koja implementira *IComponent* interfejs i dodaje stanja i/ili ponašanja (može biti više ovakvih klasa)



Slika 9.7 - UML dijagram za Decorator patern

Centralni deo UML dijagrama predstavlja *Decorator* klasa. Ona uključuje dva tipa relacija sa *IComponent* interfejsom:

- *is-a* - relacija *is-a* je prikazana isprekidanom linijom koja ide od klase *Decorator* do interfejsa *IComponent*. Ovo govori da *Decorator* implementira *IComponent* interfejs. To znači da se *Decorator* objekat može upotrebiti na svakom mestu gde se očekuje *IComponent*. Klasa *Component* takođe implementira *IComponent* interfejs, pa se ove dve klase mogu koristiti jedna umesto druge (polimorfizam). Ovo je središte Decorator paterna.
- *has-a* - relacija *has-a* je prikazana pomoću pune linije između interfejsa *IComponent* i klase *Decorator*, pri čemu je uz klasu *Decorator* dodat ne obojeni romb. Ovo govori da klasa *Decorator* koristi jednu ili više instanci implementacija *IComponent* interfejsa, i da "dekorisani" objekti mogu "živeti" duže od originalnih objekata. *Decorator* klasa poseduje atribut *component* koji je tipa *IComponent* kako bi uključio *Operaciju* koju želi da promeni. Na ovaj način Decorator patern uspeva da ostvari svoj cilj.

Metoda *addedBehavior* i atribut *addedState* u okviru klase *Decorator* predstavljaju opcione načine na koje se može proširiti originalni *Component* objekat.

9.3.2. DEMO PRIMER

U cilju demonstracije Decorator paterna, prvo ćemo kreirati demo primer koji ilustruje upotrebu ovog paterna. Kreiramo konzolnu aplikaciju u okviru Visual Studio-a *Softverko2.Decorator*. U okviru ovog projekta u okviru klase *Program.cs* dodajemo sledeći kod:

```
namespace Softverko2.Decorator
{
    class Program
    {
        interface IComponent
        {
            string Operation();
        }

        class Component : IComponent
        {
            public string Operation()
            {
                return "I am walking ";
            }
        }

        class DecoratorA : IComponent
        {
            IComponent component;

            public DecoratorA(IComponent c)
            {
                component = c;
            }

            public string Operation()
            {
                string s = component.Operation();
                s += "and listening to Classic FM ";
                return s;
            }
        }

        class DecoratorB : IComponent
        {
            IComponent component;
            public string addedState = "past the Coffee Shop ";

            public DecoratorB(IComponent c)
            {
                component = c;
            }

            public string Operation()
            {
                string s = component.Operation();
                s += "to school ";
                return s;
            }

            public string AddedBehavior()
            {

```

```

        return "and I bought a cappuccino ";
    }
}

class Client
{
    static void Display(string s, IComponent c)
    {
        Console.WriteLine(s + c.Operation());
    }

    static void Main(string[] args)
    {
        Console.WriteLine("Decorator Pattern\n");

        IComponent component = new Component();
        Display("1. Basic component: ", component);
        Display("2. A-decorated : ", new DecoratorA(component));
        Display("3. B-decorated : ", new DecoratorB(component));
        Display("4. B-A-decorated : ", new DecoratorB(
            new DecoratorA(component)));
        // Explicit DecoratorB
        DecoratorB b = new DecoratorB(new Component());
        Display("5. A-B-decorated : ", new DecoratorA(b));
        // Invoking its added state and added behavior
        Console.WriteLine("\t\t\t" + b.addedState + b.AddedBehavior());

        Console.ReadKey();
    }
}
}
}
}

```

Listing 9.13 - Kod u okviru klase *Program*

U okviru primera prvo kreiramo *IComponent* interfejs i jednostavnu klasu *Component* koja ga implementira. Pored toga dodajemo dva "dekorator" objekata, pri čemu oba implementiraju isti ovaj interfejs. Svaki takođe uključuje i instancu implementacije *IComponent* interfejsa, a to zapravo objekat koji će biti dekorisan. Klasa *DecoratorA* je veoma jednostavna i implementira metodu *Operation* tako što poziva njenu implementaciju nad objektom kojeg dekoriše i dodaje određeni tekst. Klasa *DecoratorB* je nešto složenija. Ona takođe implementira metodu *Operation* na svoj način ali dodaje i atribut *addedState* i metodu *addedBehavior*.

Klasa *Client* je odgovorna za kreiranje komponenti i "dekoratora" u različitim konfiguracijama i za prikazivanje rezultata nakon pozivanja metode *Operation*.

Nakon pokretanja prethodnog koda, dobijamo ispis koji je prikazan na slici 9.8.

```

file:///c:/users/zdravko/documents/visual studio 2015/Projects/SE/Demo/Softverko2.Decorator...
Decorator Pattern
1. Basic component: I am walking
2. A-decorated : I am walking and listening to Classic FM
3. B-decorated : I am walking to school
4. B-A-decorated : I am walking and listening to Classic FM to school
5. A-B-decorated : I am walking to school and listening to Classic FM
                    past the Coffee Shop and I bought a cappuccino

```

Slika 9.8 - Ispis nakon pokretanja aplikacije

9.3.3. REALNI PRIMER

Kao realni primer, kreiraćemo projekat u okviru kojeg postojećim klasama koje predstavljaju knjige i filmove, dodajemo funkcionalnost da se mogu izdavati. Kreiramo konzolnu aplikaciju *Softversko2.Decorator.Library*. U okviru aplikacije prvo dodajemo interfejs *ILibraryItem* sa kodom koji je prikazan u listingu 9.14.

```
namespace Softversko2.Decorator.Library
{
    interface ILibraryItem
    {
        void Display();
    }
}
```

Listing 9.14 - Kod u okviru interfejsa *ILibraryItem*

Nakon ovoga dodajemo dva klase *Book* i *Video* koje implementiraju ovaj interfejs.

```
namespace Softversko2.Decorator.Library
{
    class Book : ILibraryItem
    {
        private string _author;
        private string _title;

        // Constructor
        public Book(string author, string title)
        {
            this._author = author;
            this._title = title;
        }

        public void Display()
        {
            Console.WriteLine("\nBook ----- ");
            Console.WriteLine(" Author: {0}", _author);
            Console.WriteLine(" Title: {0}", _title);
        }
    }
}
```

Listing 9.15 - Kod u okviru klase *Book*

```
namespace Softversko2.Decorator.Library
{
    class Video : ILibraryItem
    {
        private string _director;
        private string _title;
        private int _playTime;

        // Constructor
        public Video(string director, string title, int playTime)
        {
            this._director = director;
            this._title = title;
            this._playTime = playTime;
        }

        public void Display()
        {
            Console.WriteLine("\nVideo ----- ");
            Console.WriteLine(" Director: {0}", _director);
            Console.WriteLine(" Title: {0}", _title);
        }
    }
}
```

```

        Console.WriteLine(" Playtime: {0}\n", _playTime);
    }
}

```

Listing 9.16 - Kod u okviru klase *Video*

Nakon dodavanja osnovnih klasa, dodajemo klasu *Borrowable* koja predstavlja "dekorator" klasu jer dodaje nove funkcionalnosti postojećim klasama.

```

namespace Softversko2.Decorator.Library
{
    class Borrowable : ILibraryItem
    {
        private ILibraryItem libraryItem;
        protected List<string> borrowers = new List<string>();

        // Constructor
        public Borrowable(ILibraryItem libraryItem)
        {
            this.libraryItem = libraryItem;
        }

        public void BorrowItem(string name)
        {
            borrowers.Add(name);
        }

        public void ReturnItem(string name)
        {
            borrowers.Remove(name);
        }

        public void Display()
        {
            libraryItem.Display();
            foreach (string borrower in borrowers)
            {
                Console.WriteLine(" borrower: " + borrower);
            }
        }
    }
}

```

Listing 9.17 - Kod u okviru klase *Borrowable*

Na kraju u okviru klase *Program* menjamo kod kako bi main metoda kreirala originalne i dekorisane objekte.

```

namespace Softversko2.Decorator.Library
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create book
            Book book = new Book("Worley", "Inside ASP.NET");
            book.Display();

            // Create video
            Video video = new Video("Spielberg", "Jaws", 92);
            video.Display();

            // Make video borrowable, then borrow and display
            Console.WriteLine("\nMaking video borrowable:");
        }
    }
}

```

```

        Borrowable borrowvideo = new Borrowable(video);
        borrowvideo.BorrowItem("Customer #1");
        borrowvideo.BorrowItem("Customer #2");

        borrowvideo.Display();

        // Wait for user
        Console.ReadKey();
    }
}

```

Listing 9.18 - Kod u okviru klase *Program.cs*

9.4. FACADE PATTERN

Uloga Facade paterna jeste da omogući drugačije poglede viskog nivoa na delove sistema čiji su detalji sakriveni od korisnika. U opštem slučaju, operacije koje mogu biti korisne sa stanovišta korisnika, mogu biti sačinjene od različitih delova podsistema.

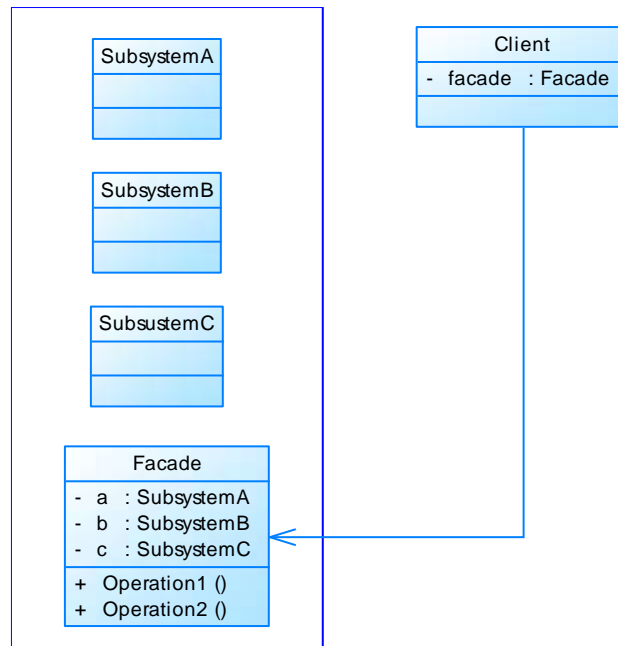
Kao ilustracija ovog paterna mogu se posmatrati jednostavni interfejsi koji vode do složenih podsistema, a koji su danas veoma česti. Oni se mogu kreirati kako bi se brže pristupalo delovima sistema koji se često koriste, ili kako bi se napravila razlika između običnih korisnika i korisnika sa privilegijama.

Tipičan primer gde se može videti upotreba ovog paterna je naručivanje robe preko interneta. Ukoliko često naručujemo robu preko nekog sajta, mi u okviru njega možemo sačuvati podatke o našoj kreditnoj kartici i o adresi gde roba treba da bude dostavljena. Na ovaj način značajno ubrzavamo proces naručivanja robe.

Skrivanje informacija je jedan od osnovnih koncepata u programiranju. Ono što razlikuje Facade patern od Adapter ili Decorator paterna je to što su interfejsi koji se kreiraju u potpunosti novi. Oni nisu povezani sa postojećim zahtevima, niti se moraju podudarati sa postojećim interfejsima. Takođe, može se kreirati više Facade paterna oko postojećeg skupa podsistema.

9.4.1. UML DIJAGRAM ZA FACADE PATTERN

Sa UML dijagrama koji je prikazan na slici 9.9 se može videti da se u okviru ovog paterna podsistemi grupišu kako bi radili zajedno i kako bi kreirali operacije na najvišem nivou.



Slika 9.9 - UML dijagram za Facade patern

U okviru prethodnog UML dijagrama imamo sledeće učesnike:

- *Subsystem* - klasa koja pruža određene operacije
- *Facade* - klasa koja nudi nekoliko operacija visokog nivoa koje se dobijaju pomoću operacija iz podsistema
- *Client* - poziva operacije visokog nivoa iz Facade klase

9.4.2. DEMO PRIMER

Ovaj patern je jednostavan za implementaciju. U okviru demo primera imamo tri podsistema koji su implementirani kao klase. Facade je statička klasa koja instancira tri podsistema kao objekte sa imenima *a*, *b*, i *c*. U okviru primera, metoda *Operation1* poziva metode iz podsistema *a* i *b*. Namena fasade je da kreira interfejs ka operacijama koje ostaju sakrivene.

Da bi demonstrirali upotrebu, kreiramo konzolnu aplikaciju *Softversko2.Facade*. U okviru aplikacije dodajemo kod koji je prikazan u okviru fajla *Program*.

```

namespace Softversko2.Facade
{
    class SubsystemA
    {
        internal string A1()
        {
            return "Subsystem A, Method A1\n";
        }

        internal string A2()
        {
            return "Subsystem A, Method A2\n";
        }
    }

    class SubsystemB
    {
        internal string B1()
        {

```

```

        return "Subsystem B, Method B1\n";
    }
}

class SubsystemC
{
    internal string C1()
    {
        return "Subsystem C, Method C1\n";
    }
}

public static class Facade
{
    static SubsystemA a = new SubsystemA();
    static SubsystemB b = new SubsystemB();
    static SubsystemC c = new SubsystemC();

    public static void Operation1()
    {
        Console.WriteLine("Operation 1\n" +
            a.A1() +
            a.A2() +
            b.B1());
    }

    public static void Operation2()
    {
        Console.WriteLine("Operation 2\n" +
            b.B1() +
            c.C1());
    }
}

class Program
{
    static void Main(string[] args)
    {
        Facade.Operation1();
        Facade.Operation2();

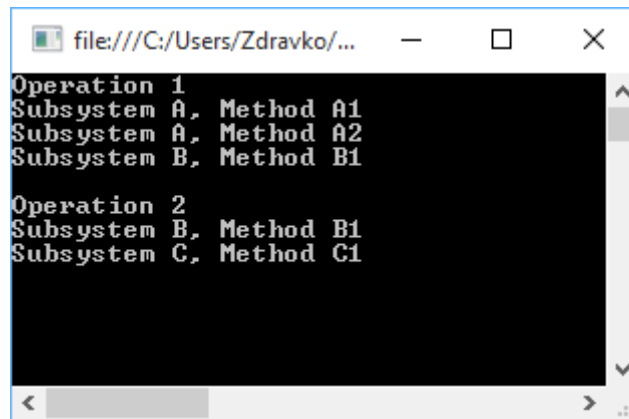
        Console.ReadKey();
    }
}
}

```

Listing 9.19 - Kod u okviru klase *Program*

Sve u okviru Facade klase treba da bude javno kako bi mu klijentski kod, koji često može biti u drugom delu aplikacije, mogao pristupiti.

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 9.10.



Slika 9.10 - Kod nakon pokretanja aplikacije

9.4.3. REALNI PRIMER

Kao realni primer, proširćemo primer koji je korišćen u okviru Composite paterna koji pokazuje kako se fotografije mogu smestiti u direktorijume sa proizvoljnom konfiguracijom. Komande su se u okviru Composite paterna oslanjale na trenutnu poziciju, što je moćan, ali možda malo zbunjujuć koncept. Zbog toga možemo primeniti Facade patern koji će omogućiti korisnicima da dodaju sliku nakon navođenja komande:

```
upload photoname setname
```

koja će se prevesti u sledeći skup naredbi:

```
find album
AddPhoto photoname
display
```

Postojeći kod koji je prikazan u okviru primera za Composite patern se neće menjati. Dodajemo samo Facade klasu, a menjamo test program kako bi se videla upotreba novog paterna.

Da bi demonstrirali upotrebu Facade paterna kreiramo konzolnu aplikaciju u okviru Visual Studio-a pod imenom Softversko2.Facade.Photos. U okviru aplikacije dodajemo klasu *CompositeContainer.cs* koja će u sebi sadržati kod iz Composite paterna.

```
namespace Softversko2.Facade.Photos
{
    public class CompositeContainer
    {
        public interface IComponent<T>
        {
            void Add(IComponent<T> c);
            IComponent<T> Remove(T s);
            string Display(int depth);
            IComponent<T> Find(T s);
            T Name { get; set; }
        }

        public class Component<T> : IComponent<T>
        {
            public T Name { get; set; }

            public Component(T name)
            {
                Name = name;
            }

            public void Add(IComponent<T> c)
            {
                // ...
            }
        }
    }
}
```

```

    {
        Console.WriteLine("Cannot add to an item");
    }

    public IComponent<T> Remove(T s)
    {
        Console.WriteLine("Cannot remove directly");
        return this;
    }

    public string Display(int depth)
    {
        return new String('-', depth) + Name + "\n";
    }

    public IComponent<T> Find(T s)
    {
        if (s.Equals(Name))
            return this;
        else
            return null;
    }
}

public class Composite<T> : IComponent<T>
{
    List<IComponent<T>> list;

    public T Name { get; set; }

    public Composite(T name)
    {
        Name = name;
        list = new List<IComponent<T>>();
    }

    public void Add(IComponent<T> c)
    {
        list.Add(c);
    }

    IComponent<T> holder = null;

    // Finds the item from a particular point in the structure
    // and returns the composite from which it was removed
    // If not found, return the point as given
    public IComponent<T> Remove(T s)
    {
        holder = this;
        IComponent<T> p = holder.Find(s);
        if (holder != null)
        {
            (holder as Composite<T>).list.Remove(p);
            return holder;
        }
        else
            return this;
    }

    // Recursively looks for an item
    // Returns its reference or else null
    public IComponent<T> Find(T s)

```

```

    {
        holder = this;
        if (Name.Equals(s)) return this;
        IComponent<T> found = null;
        foreach (IComponent<T> c in list)
        {
            found = c.Find(s);
            if (found != null)
                break;
        }
        return found;
    }

    // Displays items in a format indicating level in composite structure
    public string Display(int depth)
    {
        StringBuilder s = new StringBuilder(new String('-', depth));
        s.Append("Set " + Name + " length : " + list.Count + "\n");
        foreach (IComponent<T> component in list)
        {
            s.Append(component.Display(depth + 2));
        }
        return s.ToString();
    }
}

```

Listing 9.20 - Kod u okviru klase *CompositeContainer*

Nakon ovoga dodajemo klasu *ImageFacade* koja implementira Facade patern. Kod u okviru ove klase je dat u nastavku:

```

namespace Softversko2.Facade.Photos
{
    public static class ImageFacade
    {
        static CompositeContainer.IComponent<string> album = new
        CompositeContainer.Composite<string>("Album");

        public static void AddAlbum(string albumName, string parentAlbumName)
        {
            CompositeContainer.IComponent<string> point =
            album.Find(parentAlbumName);
            CompositeContainer.IComponent<string> newAlbum = new
            CompositeContainer.Composite<string>(albumName);
            point.Add(newAlbum);
            Console.WriteLine(album.Display(0));
        }

        public static void AddImage(string imageName, string albumName)
        {
            CompositeContainer.IComponent<string> point = album.Find(albumName);
            point.Add(new CompositeContainer.Component<string>(imageName));
            Console.WriteLine(album.Display(0));
        }
    }
}

```

Listing 9.21 - Kod u okviru klase *ImageFacade*

Poslednji korak je izmena u okviru *main* metode u klasi *Program*.

```

namespace Softversko2.Facade.Photos
{

```



```

class Program
{
    static void Main(string[] args)
    {
        ImageFacade.AddAlbum("Pets", "Album");
        ImageFacade.AddImage("Dog.jpg", "Pets");
        ImageFacade.AddImage("Cat.jpg", "Pets");

        ImageFacade.AddAlbum("Small pets", "Pets");
        ImageFacade.AddImage("Fish.jpg", "Small pets");
        Console.ReadKey();
    }
}

```

Listing 9.22 - Kod u okviru klase *Program.cs*

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 9.11.

```

file:///C:/Users/Zdravko/...
Set Album length :1
--Set Pets length :0
Set Album length :1
--Set Pets length :1
----Dog.jpg
Set Album length :1
--Set Pets length :2
----Dog.jpg
----Cat.jpg
Set Album length :1
--Set Pets length :3
----Dog.jpg
----Cat.jpg
----Set Small pets length :0
Set Album length :1
--Set Pets length :3
----Dog.jpg
----Cat.jpg
----Set Small pets length :1
-----Fish.jpg

```

Slika 9.11 - Ispis koji se dobija nakon pokretanja aplikacije

9.5. PROXY PATTERN

Proxy patern kreira objekte koji kontrolišu kreiranje novih objekata ili pristup određenim objektima. Proxy je obično mali javni (public) objekat koji stoji ispred kompleksnijeg privatnog (private) objekta kome se može pristupiti kada su ispunjeni određeni uslovi.

Kao ilustracija za upotrebu ovog objekta se može posmatrati Facebook. Karakteristika ovakvog sistema jeste da veliki broj ljudi pristupa sistemu samo da bi videlo šta ostali rade, ali ne i da aktivno učestvuju ili dodaju sadržaj. Zbog toga se ovakav sistem može kreirati tako da se svakom novom članu daje prazna stranica sve dok ne počne da dodaje prijatelje i postavlja status i slike.

Još jedna karakteristika ovakvih sistema jeste da se prvo moramo registrovati a zatim logovati svaki put kada hoćemo da mu pristupimo. Kada smo logovani, možemo pristupiti stranicama naših prijatelja. Sve akcije koje možemo izvesti (slanje poruka, pisanje na zid, ...) počinju u našem internet pretraživaču

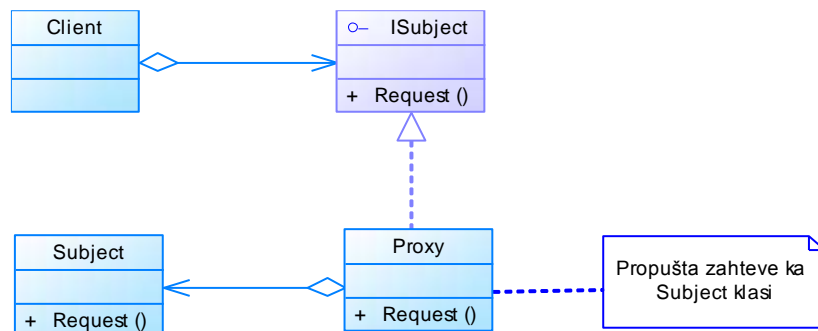
a zatim se šalju putem interneta do najbližeg Facebook servera. Objekti su Facebook stranice, a proxy objekti su mehanizmi koji dozvoljavaju registrovanje, logovanje i pristup.

Postoji više tipova proxy objekata. Najčešće korišćeni su:

- *virtual proxies* - prepuštaju kreiranje objekta drugom objektu (kada je proces kreiranja spor)
- *authentication proxies* - proveravaju da li su parametri za pristup korektni
- *remote proxies* - kodiraju zahtev i šalju ga preko mreže
- *smart proxies* - menjaju zahtev pre nego što ga proslede

9.5.1. UML DIJAGRAM PROXY PATERNA

UML dijagram Proxy paterna je prikazan na slici 9.12.



Slika 9.12 - UML dijagram za Proxy patern

Učesnici u okviru paterna su:

- *ISubject* - zajednički interfejs za klase *Subject* i *Proxy* koji omogućava da se one upotrebljavaju jedna umesto druge
- *Subject* - klasa kojoj želimo da pristupimo
- *Proxy* - klasa koja kreira, kontroliše i omogućava pristup ka klasi *Subject*
- *Request* - operacija u okviru klase *Subject* koju omogućava *Proxy*

Klasa *Proxy* implementira *ISubject* interfejs. Svaki *Proxy* objekat sadrži referencu ka klasi *Subject* u okviru koje se zapravo izvršava tražena funkcionalnost. *Proxy* predstavlja pristupnu tačku. Njegova važnost može biti povećana ako kreiramo klasu *Subject* kao privatnu, tako da joj klijent ne može pristupiti osim kroz *Proxy*.

9.5.2. DEMO PRIMER

U cilju demonstracije Proxy paterna, prvo ćemo kreirati demo primer koji ilustruje upotrebu ovog paterna. Ovaj primer ima za cilj da prikaže:

- kako se može razdvojiti klasa *Client* sa jedne strane i klase *Proxy* i *Subject* sa druge strane, upotrebom jakog modifikatora pristupa
- kako virtuelni i zaštitni proxy omogućavaju prosleđivanje poziva

Kreiramo konzolnu aplikaciju u okviru Visual Studio-a *Softverko2.Proxy*. U okviru ovog projekta u okviru klase *Program* dodajemo sledeći kod:

```

namespace Softversko2.Proxy
{
    class Program
    {
        class SubjectAccessor
  
```

```

{
    public interface ISubject
    {
        string Request();
    }

    private class Subject
    {
        public string Request()
        {
            return "Subject Request " + "Choose left door\n";
        }
    }

    public class Proxy : ISubject
    {
        Subject subject;

        public string Request()
        {
            // A virtual proxy creates the object only on its first call
            if (subject == null)
            {
                Console.WriteLine("Subject inactive");
                subject = new Subject();
            }
            Console.WriteLine("Subject active");
            return "Proxy: Call to " + subject.Request();
        }
    }

    public class ProtectionProxy : ISubject
    {
        // An authentication proxy first asks for a password
        Subject subject;
        string password = "Abracadabra";

        public string Authenticate(string supplied)
        {
            if (supplied != password)
                return "Protection Proxy: No access";
            else
                subject = new Subject();
            return "Protection Proxy: Authenticated";
        }

        public string Request()
        {
            if (subject == null)
                return "Protection Proxy: Authenticate first";
            else
                return "Protection Proxy: Call to " +
                    subject.Request();
        }
    }
}

class Client
{
    static void Main()
    {
        Console.WriteLine("Proxy Pattern\n");
    }
}

```

```

        SubjectAccessor.ISubject proxy = new SubjectAccessor.Proxy();
        Console.WriteLine(proxy.Request());
        Console.WriteLine(proxy.Request());

        SubjectAccessor.ISubject pProxy = new
            SubjectAccessor.ProtectionProxy();
        Console.WriteLine(pProxy.Request());
        Console.WriteLine((pProxy as SubjectAccessor.ProtectionProxy)
            .Authenticate("Secret"));
        Console.WriteLine((pProxy as SubjectAccessor.ProtectionProxy)
            .Authenticate("Abracadabra"));
        Console.WriteLine(pProxy.Request());

        Console.ReadKey();
    }
}
}
}

```

Listing 9.23 - Kod u okviru klase *Program.cs*

Nakon pokretanja aplikacije, dobijamo ispis koji je prikazan u okviru slike 9.13.

```

file:///C:/Users/Zdravko/Documents/Visual Studio 2015/Projects/SE/Demo/Softversko2.Proxy...
Proxy Pattern
Subject inactive
Subject active
Proxy: Call to Subject Request Choose left door
Subject active
Proxy: Call to Subject Request Choose left door
Protection Proxy: Authenticate first
Protection Proxy: No access
Protection Proxy: Authenticated
Protection Proxy: Call to Subject Request Choose left door

```

Slika 9.13 - Ispis nakon pokretanja demo aplikacije za Proxy patern

Program je strukturiran u okviru klase *SubjectAccessor* čija je svrha da grupiše proksije i objekat kojem se pristupa. Interfejs i proxy objekti su kreirani kao public, kako bi oni i njihovi javni članovi bili u potpunosti dostupni klijentskim klasama. Namera i virtual i protection proxy-ja jeste da omoguće pristup klasi *Subject*. Zbog toga je klasa *Subject* kreirana kao private. Metoda *Request* je definisana kao javna, ali je vidljiva samo klasama koje mogu da vide ovu klasu, a to su klase koje se nalaze unutar *SubjectAccessor* klase.

U okviru aplikacije smo demonstrirali način na koji se mogu koristiti prixy-ji i *Subject* klasa. Oni su namenjeni da se uvek koriste zajedno, pa smo ih zato i grupisali u okviru jedne klase.

9.5.3. REALNI PRIMER

Da bi istražili upotrebu proxy-ja, kreiraćemo primer društvene mreže pod imenom *SpaceBook*. Ova aplikacija čuva tekstualne stranice koje klijenti unose pod njihovim login imenom, pa su neophodni autentikacija i kreiranje stranica. Čisto logovanje korisnika na sistem im neće obezbediti prostor u okviru aplikacije. Da bi dobili prostor, moraju prvo kreirati neki tekst. Zbog toga, od samog početka, sistem mora omogućiti korisnicima da pišu na stranicama drugih korisnika.

U cilju kreiranja ovog projekta, u okviru Visual Studio-a kreiramo konzolnu aplikaciju pod imenom *Softversko2.Proxy.SpaceBook*. U okviru aplikacije prvo dodajemo klasu *SpaceBookSystem* sa sledećim kodom:

```

namespace Softversko2.Proxy.SpaceBook
{
    class SpaceBookSystem
    {
        // The Subject
        private class SpaceBook
        {
            static SortedList<string, SpaceBook> community =
                new SortedList<string, SpaceBook>(100);
            string pages;
            string name;
            string gap = "\n\t\t\t\t\t";

            static public bool IsUnique(string name)
            {
                return community.ContainsKey(name);
            }

            internal SpaceBook(string n)
            {
                name = n;
                community[n] = this;
            }

            internal void Add(string s)
            {
                pages += gap + s;
                Console.Write(gap + "=====" + name + "'s SpaceBook =====");
                Console.Write(pages);
                Console.WriteLine(gap + "=====");
            }

            internal void Add(string friend, string message)
            {
                community[friend].Add(message);
            }

            internal void Poke(string who, string friend)
            {
                community[who].pages += gap + friend + " poked you";
            }
        }

        // The Proxy
        public class MySpaceBookProxy
        {
            // Combination of a virtual and authentication proxy
            SpaceBook mySpaceBook;
            string password;
            string name;
            bool loggedIn = false;

            void Register()
            {
                Console.WriteLine("Let's register you for SpaceBook");
                do
                {
                    Console.WriteLine("All SpaceBook names must be unique");
                    Console.Write("Type in a user name: ");
                    name = Console.ReadLine();
                } while (!SpaceBook.IsUnique(name));
            }
        }
    }
}

```

```

        Console.WriteLine("Type in a password: ");
        password = Console.ReadLine();
        Console.WriteLine("Thanks for registering with SpaceBook");
    }

    bool Authenticate()
    {
        Console.WriteLine("Welcome " + name + ". Please type in password: ");
        string supplied = Console.ReadLine();
        if (supplied == password)
        {
            loggedIn = true;
            Console.WriteLine("Logged into SpaceBook");
            if (mySpaceBook == null)
                mySpaceBook = new SpaceBook(name);
            return true;
        }
        Console.WriteLine("Incorrect password");
        return false;
    }

    public void Add(string message)
    {
        Check();
        if (loggedIn) mySpaceBook.Add(message);
    }

    public void Add(string friend, string message)
    {
        Check();
        if (loggedIn)
            mySpaceBook.Add(friend, name + " said: " + message);
    }

    public void Poke(string who)
    {
        Check();
        if (loggedIn)
            mySpaceBook.Poke(who, name);
    }

    void Check()
    {
        if (!loggedIn)
        {
            if (password == null)
                Register();
            if (mySpaceBook == null)
                Authenticate();
        }
    }
}
}
}
}

```

Listing 9.24 - Kod u okviru klase *SpaceBookSystem*

Klasa *SpaceBook* čuva statičku listu svih trenutnih korisnika. Koristimo ugrađenu *SortedList* klasu koju indeksiramo pomoću imena korisnika. Konstruktor postavlja instancu *SpaceBook* klase u okviru liste pod određenim imenom. Postoje dve *Add* metode. Prva je za dodavanje poruke za datog korisnika, a druga je dodavanje poruke na stranicu drugog korisnika. Tu je i metoda *Poke* koja kreira unapred definisano poruku na stranicu osobe koju navedemo.

SpaceBook poseduje jednu javnu metodu *IsUnique* koja proverava da li ime već postoji u kolekciji, pa ga novi korisnik ne može koristiti.

Authentikacija je kreirana kroz klasu *MySpaceBookProxy*. Kada korisnik pozove metodu *Add*, ova metoda ga odmah vodi na proveru statusa (metoda *Check*), nakon čega se vrše korak registracije i unosa lozinke, a nakon toga autentikacije na osnovu unete lozinke.

Nakon ovoga menjamo kod u okviru metode *main* u klasi *Program*:

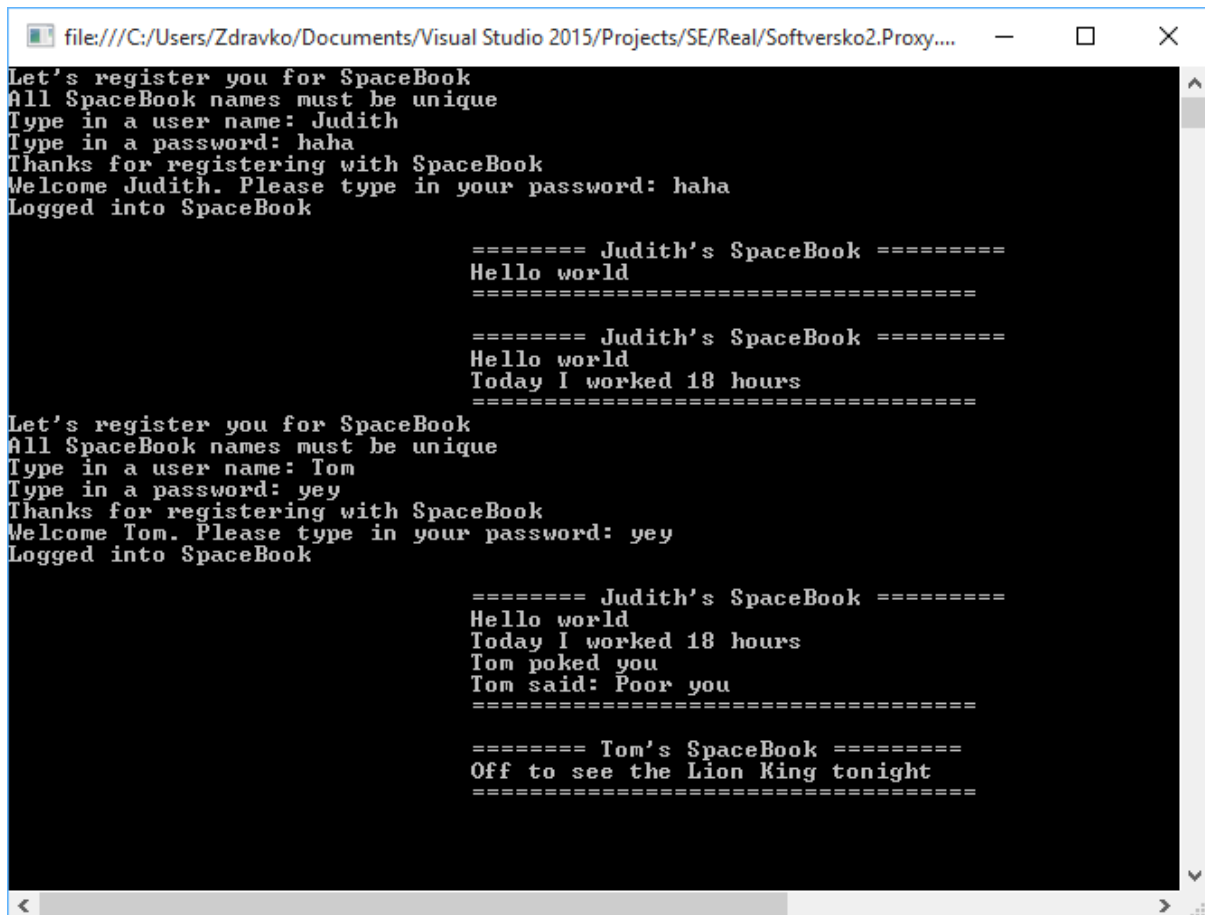
```
namespace Softversko2.Proxy.SpaceBook
{
    class Program
    {
        static void Main(string[] args)
        {
            SpaceBookSystem.MySpaceBookProxy me = new
                SpaceBookSystem.MySpaceBookProxy();
            me.Add("Hello world");
            me.Add("Today I worked 18 hours");
            SpaceBookSystem.MySpaceBookProxy tom = new
                SpaceBookSystem.MySpaceBookProxy();
            tom.Poke("Judith");
            tom.Add("Judith", "Poor you");
            tom.Add("Off to see the Lion King tonight");

            Console.ReadKey();
        }
    }
}
```

Listing 9.25 - Kod u okviru klase *Program*

U okviru *main* metode, provo se kreira novi korisnik (pod imenom "Judith" zbog daljeg izvršavanja koje je hard kodovano) tako što se kreira instanca *MySpaceBookProxy* klase. Nakon toga ovaj korisnik kreira dve poruke. Zatim se kreira korisnik (nazvali smo ga "Tom"). On prvo poziva metodu *Poke* nad korisnikom "Judith", pa dodaje poruku na njen nalog i na svoj nalog.

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 9.XX.



```
file:///C:/Users/Zdravko/Documents/Visual Studio 2015/Projects/SE/Real/Softversko2.Proxy...
Let's register you for SpaceBook
All SpaceBook names must be unique
Type in a user name: Judith
Type in a password: haha
Thanks for registering with SpaceBook
Welcome Judith. Please type in your password: haha
Logged into SpaceBook

===== Judith's SpaceBook =====
Hello world
=====

===== Judith's SpaceBook =====
Hello world
Today I worked 18 hours
=====

Let's register you for SpaceBook
All SpaceBook names must be unique
Type in a user name: Tom
Type in a password: yey
Thanks for registering with SpaceBook
Welcome Tom. Please type in your password: yey
Logged into SpaceBook

===== Judith's SpaceBook =====
Hello world
Today I worked 18 hours
Tom poked you
Tom said: Poor you
=====

===== Tom's SpaceBook =====
Off to see the Lion King tonight
=====
```

Silka 9.14 - Ispis nakon pokretanja aplikacije SpaceBook za ilustraciju Proxy paterna

Možemo primetiti da se u okviru koda ne kreira *ISubject* interfejs. Klase *SpaceBook* i *MySpaceBookProxy* se nalaze u okviru klase *SpaceBookSystem* i sarađuju kroz agregaciju (*MySpaceBookProxy* sadrži referencu na *SpaceBook* objekat).

9.6. BRIDGE PATTERN

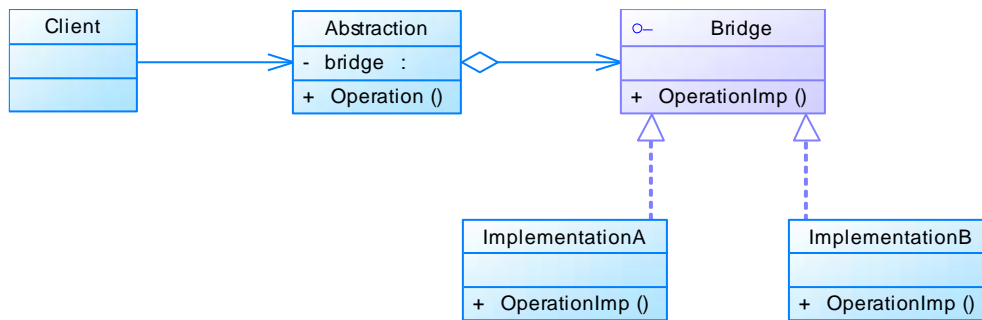
Bridge patern se koristi kada želimo da razdvojimo abstrakciju od implementacije, kako bi mogle da postoje nezavisno. Ovaj patern je koristan kada se razvije nova verzija softvera koja treba da zameni postojeću verziju, ali stara verzija mora i dalje da postoji zbog postojeće baze klijenata. Klijentski kod se neće menjati jer zavisi od apstrakcije. Klijent će morati da odabere koju verziju želi da koristi.

Bridge je veoma jednostavan i veoma moćan patern. Nad postojećom implementacijom, možemo dodati novu, zajedno sa Bridge interfejsom i klasom koja će predstavljati apstrakciju. Na ovaj način dobijamo generalizaciju nad početnim dizajnom.

Ovaj patern koristimo kada postoje opcije koje se ne implementiraju uvek na isti način.

9.6.1. UML DIJAGRAM BRIDGE PATERNA

UML dijagram koji prikazuje Bridge patern je prikazan na slici 9.15.



Slika 9.15 - UML dijagram za Bridge patern

Uloge elemenata prikazanih na UML dijagramu u okviru Bridge paterna su:

- *Abstraction* - interfejs kojem pristupa klijent
- *Operation* - metoda koju klijent poziva
- *Bridge* - interfejs koji definiše one delove klase *Abstraction* koji se razlikuju u implementacijama
- *ImplementationA*, *ImplementationB* - implementacije za *Bridge* interfejs
- *OperationImp* - metoda u okviru *Bridge* interfejsa koju poziva metoda *Operation* iz klase *Abstraction*

9.6.2. DEMO PRIMER

U cilju demonstracije Bridge paterna, prvo ćemo kreirati demo primer koji ilustruje upotrebu ovog paterna. Kreiramo konzolnu aplikaciju u okviru Visual Studio-a *Softverko2.Bridge*. U okviru ovog projekta u okviru klase *Program* dodajemo sledeći kod:

```

namespace Softversko2.Bridge
{
    class Program
    {
        class Abstraction
        {
            Bridge bridge;

            public Abstraction(Bridge implementation)
            {
                bridge = implementation;
            }

            public string Operation()
            {
                return "Abstraction" + " <<< BRIDGE >>>> " + bridge.OperationImp();
            }
        }

        interface Bridge
        {
            string OperationImp();
        }

        class ImplementationA : Bridge
        {
            public string OperationImp()
            {
                return "ImplementationA";
            }
        }
    }
}

```

```

    }

    class ImplementationB : Bridge
    {
        public string OperationImp()
        {
            return "ImplementationB";
        }
    }

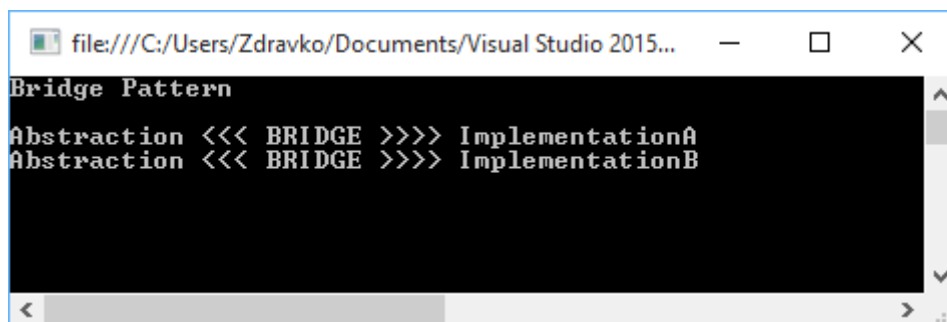
    static void Main()
    {
        Console.WriteLine("Bridge Pattern\n");
        Console.WriteLine(new Abstraction(new ImplementationA()).Operation());
        Console.WriteLine(new Abstraction(new ImplementationB()).Operation());

        Console.ReadKey();
    }
}

```

Listing 9.26 - Kod u okviru klase *Program*

U okviru klijenta, svaka instanca klase *Abstraction* ima svoju sopstvenu implementaciju. Nakon toga se pozivaju metode *Operation*. Svaki poziv ide ka drugoj *OperationImp* metodi. Kao rezultat može se videti ispis koji je prikazan na slici 9.16.



Slika 9.16 - Ispis nakon pokretanja demo aplikacije za Bridge patern

9.6.3. REALNI PRIMER

Bridge patern nam pruža mogućnost da proširimo primer SpaceBook koji je prikazan u okviru Proxy paterna. Ovde ćemo videti kako različiti paterni međusobno sarađuju. Prvo ponovo posmatramo SpaceBook aplikaciju. Pretpostavimo da se radi na novoj verziji ovog sistema pod imenom OpenBook. Glavna karakteristika OpenBook sistema jeste to što ne zahteva autentikaciju putem lozinke. On podrazumeva da svi korisnici odmah imaju prava da pristupe sistemu. Iz ovoga se može zaključiti da OpenBook sistem izbacuje proxy iz SpaceBook sistema. Cilj je da korisnici oba sistema mogu da vide stranice korisnika drugog sistema.

Zbog toga ćemo imati dizajn u okviru koga MyOpenBook i MySpaceBookProxy moraju da implementiraju zajednički interfejs. Ovaj interfejs ćemo nazvati *IBridge*. Klasu koja predstavlja apstakciju ćemo nazvati *Portal*. Ona će čuvati kopiju objekta odgovarajućeg sistema i prosledivati svaku metodu iz interfejsa na odgovarajuću verziju objekta.

U cilju kreiranja ovog projekta, u okviru Visual Studio-a kreiramo konzolnu aplikaciju pod imenom *Softversko2.Bridge.OpenBook*. U okviru aplikacije prvo dodajemo interfejs *IBridge* sa sledećim kodom:

```

namespace Softversko2.Bridge.OpenBook
{
    interface IBridge

```

```

{
    void Add(string message);
    void Add(string friend, string message);
    void Poke(string who);
}

```

Listing 9.27 - Kod za interfejs *IBridge*

Nakon ovoga dodajemo klasu *Portal* sa sledećim kodom:

```

namespace Softversko2.Bridge.OpenBook
{
    class Portal
    {
        IBridge bridge;

        public Portal(IBridge aSpaceBook)
        {
            bridge = aSpaceBook;
        }

        public void Add(string message)
        {
            bridge.Add(message);
        }

        public void Add(string friend, string message)
        {
            bridge.Add(friend, message);
        }

        public void Poke(string who)
        {
            bridge.Poke(who);
        }
    }
}

```

Listing 9.28 - Kod za klasu *Portal*

Portal je veoma jednostavna klasa koja samo izvršava apstrakciju. Ona ima kao atribut implementaciju za interfejs *IBridge.cs*. U okviru svojih metoda, ona prosleđuje zahtev ka konkretnim implementacijama ovog interfejsa.

Nakon ovoga dodajemo klasu *SpaceBookSystem* sa sledećim kodom:

```

namespace Softversko2.Bridge.OpenBook
{
    class SpaceBookSystem
    {
        // The Subject
        private class SpaceBook
        {
            static SortedList<string, SpaceBook> community =
                new SortedList<string, SpaceBook>(100);
            string pages;
            string name;
            string gap = "\n\t\t\t\t\t";

            static public bool IsUnique(string name)
            {
                return community.ContainsKey(name);
            }
        }
    }
}

```

```

internal SpaceBook(string n)
{
    name = n;
    community[n] = this;
}

internal void Add(string s)
{
    pages += gap + s;
    Console.Write(gap + "=====" + name + "'s SpaceBook =====");
    Console.Write(pages);
    Console.WriteLine(gap + "=====");
}

internal void Add(string friend, string message)
{
    community[friend].Add(message);
}

internal void Poke(string who, string friend)
{
    community[who].pages += gap + friend + " poked you";
}
}

// The Proxy
public class MySpaceBookProxy : IBridge
{
    // Combination of a virtual and authentication proxy
    SpaceBook mySpaceBook;
    string password;
    string name;
    bool loggedIn = false;

    void Register()
    {
        Console.WriteLine("Let's register you for SpaceBook");
        do
        {
            Console.WriteLine("All SpaceBook names must be unique");
            Console.Write("Type in a user name: ");
            name = Console.ReadLine();
        } while (!SpaceBook.IsUnique(name));

        Console.Write("Type in a password: ");
        password = Console.ReadLine();
        Console.WriteLine("Thanks for registering with SpaceBook");
    }

    bool Authenticate()
    {
        Console.Write("Welcome " + name + ". Please type in password: ");
        string supplied = Console.ReadLine();
        if (supplied == password)
        {
            loggedIn = true;
            Console.WriteLine("Logged into SpaceBook");
            if (mySpaceBook == null)
                mySpaceBook = new SpaceBook(name);
            return true;
        }
        Console.WriteLine("Incorrect password");
    }
}

```

```

        return false;
    }

    public void Add(string message)
    {
        Check();
        if (loggedIn) mySpaceBook.Add(message);
    }

    public void Add(string friend, string message)
    {
        Check();
        if (loggedIn)
            mySpaceBook.Add(friend, name + " said: " + message);
    }

    public void Poke(string who)
    {
        Check();
        if (loggedIn)
            mySpaceBook.Poke(who, name);
    }

    void Check()
    {
        if (!loggedIn)
        {
            if (password == null)
                Register();
            if (mySpaceBook == null)
                Authenticate();
        }
    }
}

public class MyOpenBook : IBridge
{
    // Combination of a virtual and authentication proxy
    SpaceBook myOpenBook;
    string name;

    public MyOpenBook(string n)
    {
        name = n;
        myOpenBook = new SpaceBook(name);
    }

    public void Add(string message)
    {
        myOpenBook.Add(message);
    }

    public void Add(string friend, string message)
    {
        myOpenBook.Add(friend, name + " : " + message);
    }

    public void Poke(string who)
    {
        myOpenBook.Poke(who, name);
    }
}

```

```
}
}
```

Listing 9.29 - Kod u okviru klase *SpaceBookSystem*

Poslednji korak je izmena u okviru metode *main* u klasi *Program*.

```
namespace Softversko2.Bridge.OpenBook
{
    class Program
    {
        static void Main(string[] args)
        {
            Portal me = new Portal(new SpaceBookSystem.MySpaceBookProxy()); //Judith
            me.Add("Hello world");
            me.Add("Today I worked 18 hours");
            Portal tom = new Portal(new SpaceBookSystem.MyOpenBook("Tom"));
            tom.Poke("Judith");
            tom.Add("Judith", "Poor you");
            tom.Add("Hey, I'm on OpenBook - it's cool!");

            Console.ReadKey();
        }
    }
}
```

Listing 9.30 - Kod u okviru klase *Program*

U okviru *main* metode vidimo da prvi korisnik (kome ćemo dodeliti ime "Judith") koristi staru verziju *SpaceBook* koja koristi *Proxy*. Korisnik pod imenom "Tom" će koristiti novu verziju *OpenBook*. Prilikom pokretanja aplikacije, korisnik "Judith" će morati da se registruje kako bi pristupila aplikaciji, dok će "Tom" moći da koristi aplikaciju bez registrovanja.

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 9.17.

```
file:///C:/Users/Zdravko/Documents/Visual Studio 2015/Projects/SE/Real/Softversko2.Bridge...
Let's register you for SpaceBook
All SpaceBook names must be unique
Type in a user name: Judith
Type in a password: haha
Thanks for registering with SpaceBook
Welcome Judith. Please type in password: haha
Logged into SpaceBook

===== Judith's SpaceBook =====
Hello world
=====

===== Judith's SpaceBook =====
Hello world
Today I worked 18 hours
=====

===== Judith's SpaceBook =====
Hello world
Today I worked 18 hours
Tom poked you
Tom : Poor you
=====

===== Tom's SpaceBook =====
Hey, I'm on OpenBook - it's cool!
=====
```

Silka 9.17 - Ispis nakon pokretanja aplikacije *OpenBook* za ilustraciju *Bridge* paterna

10. DIZAJN PATERNI ZA KREIRANJE OBJEKATA

Cilj ovih paterna jeste da razdvoje sistem od načina na koji su objekti kreirani, od čega su sastavljeni i kako se prikazuju. Ovi paterni povećavaju fleksibilnost sistema. Oni obuhvataju znanje o tome koje klase sistem koristi, ali sakrivaju detalje kako se instance tih klasa kreiraju i kombinuju. Vremenom su programeri uvideli da kreiranje sistema pomoću nasleđivanja suviše kruto. Paterni za kreiranje objekata su nastali kako bi razrešili ovu blisko povezivanje.

10.1. PROTOTYPE PATTERN

Prototype patern se odnosi na kloniranje objekta. On ima dve prednosti: ubrzava instanciranje veoma velikih klasa koje se dinamički učitavaju (kada je kopiranje objekata brže), i čuva znanje o sastavnim delovima velikih struktura podataka koje se mogu kopirati bez da je potrebno znati podklase od kojih su kreirane.

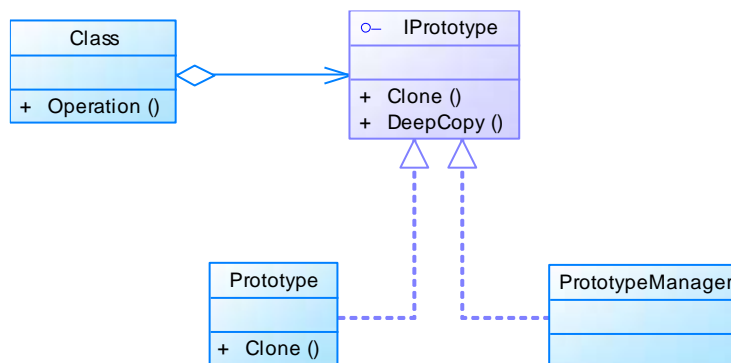
Kao primer upotrebe ovog paterna, možemo posmatrati aplikaciju koja radi sa slikama i koja čuva grupe fotografija. U određenom momentu, mi možemo poželeti da arhiviramo jednu grupu fotografija, tako što ćemo ih kopirati u drugi album. U ovom slučaju, arhiva postaje kontejner za prototipove koji se može kopirati kada god je to neophodno.

Objekti se obično instanciraju od klasa koje predstavljaju deo programa. Prototype patern predstavlja alternativu ovom pristupu jer se objekti kreiraju na osnovu prototipova.

Sušтина kod ovog paterna jeste da program kreira objekat željenog tipa, ne tako što će da kreira novu instancu, već tako što će da kopira praznu instancu date klase. Proces kopiranja (kloniranja) se može ponoviti više puta. Kopije predstavljaju nezavisne objekte kojima se vrednosti mogu menjati bez da to utiče na prototip. Tokom izvršavanja programa se mogu dodavati novi prototipovi i ti bilo iz novih klasa, bilo iz već postojećih prototipova

10.1.1. UML DIJAGRAM ZA PROTOTYPE PATTERN

UML dijagram za Prototype patern je prikazan na slici 10.1.



Slika 10.1 - UML dijagram za Prototype patern

Sa slike 10.1 se može videti da koristimo klasu *PrototypeManager*. Iako postoje i drugačiji pristupi u okviru ovog paterna, ova klasa omogućava veoma fleksibilan rad, jer čuva indeksiranu listu prototipova koje je moguće klonirati.

Osnovni elementi u okviru ovog paterna su:

- *IPrototype* - predstavlja interfejs koji govori da prototipovi moraju biti takvi da ih je moguće klonirati
- *Prototype* - klasa koju je moguće klonirati
- *PrototypeManager* - sadrži listu tipova koje je moguće klonirati i njihove ključeve

- *Client* - dodaje prototipove u listu i zahteva klonove

10.1.2.DEMO PRIMER

Implementacija Prototype paterna u C# programskom jeziku ima veliku pomoć koju nude dve mogućnosti .NET Framework-a: kloniranje i serijalizacija. Obe karakteristike su omogućene putem interfejsa iz *System* imenskog prostora.

Kao primer, kreiraćemo program koji kreira tri prototipa, pri čemu se svaki sastoji od atributa *country*, *capital* i *language*. Poslednji atribut (*language*) se odnosi na drugu klasu pod imenom *DeeperData*. Svrha ove klase jeste da kreira referencu u okviru prototipa. Primer će prikazati kako referenca na objekat pravi razliku između plitkog kopiranja (*shallow*) i dubokog kopiranja (*deep*).

U cilju kreiranja primera dodajemo konzolnu aplikaciju *Softversko2.Prototype*. U okviru aplikacije u fajlu *Program* kucamo sledeći kod:

```
namespace Softversko2.Prototype
{
    [Serializable()]
    public abstract class IPrototype<T>
    {
        // Shallow copy
        public T Clone()
        {
            return (T)this.MemberwiseClone();
        }

        // Deep Copy
        public T DeepCopy()
        {
            MemoryStream stream = new MemoryStream();
            BinaryFormatter formatter = new BinaryFormatter();
            formatter.Serialize(stream, this);
            stream.Seek(0, SeekOrigin.Begin);
            T copy = (T)formatter.Deserialize(stream);
            stream.Close();
            return copy;
        }
    }

    [Serializable()]
    // Helper class used to create a second level data structure
    class DeeperData
    {
        public string Data { get; set; }

        public DeeperData(string s)
        {
            Data = s;
        }

        public override string ToString()
        {
            return Data;
        }
    }

    [Serializable()]
    class Prototype : IPrototype<Prototype>
    {
        // Content members
        public string Country { get; set; }
        public string Capital { get; set; }
    }
}
```



```

    public DeeperData Language { get; set; }

    public Prototype(string country, string capital, string language)
    {
        Country = country;
        Capital = capital;
        Language = new DeeperData(language);
    }

    public override string ToString()
    {
        return Country + "\t\t" + Capital + "\t\t->" + Language;
    }
}

[Serializable()]
class PrototypeManager : IPrototype<Prototype>
{
    public Dictionary<string, Prototype> prototypes
        = new Dictionary<string, Prototype> {
            { "Germany",
              new Prototype("Germany", "Berlin", "German") },
            { "Italy",
              new Prototype("Italy", "Rome", "Italian") },
            { "Australia",
              new Prototype("Australia", "Canberra", "English") }
        };
}

class Program : IPrototype<Prototype>
{
    static void Report(string s, Prototype a, Prototype b)
    {
        Console.WriteLine("\n" + s);
        Console.WriteLine("Prototype " + a + "\nClone " + b);
    }

    static void Main(string[] args)
    {
        PrototypeManager manager = new PrototypeManager();
        Prototype c2, c3;

        // Make a copy of Australia's data
        c2 = manager.prototypes["Australia"].Clone();
        Report("Shallow cloning Australia\n=====",
            manager.prototypes["Australia"], c2);

        // Change the capital of Australia to Sydney
        c2.Capital = "Sydney";
        Report("Altered Clone's shallow state, prototype unaffected",
            manager.prototypes["Australia"], c2);

        // Change the language of Australia (deep data)
        c2.Language.Data = "Chinese";
        Report("Altering Clone deep state: prototype affected *****",
            manager.prototypes["Australia"], c2);

        // Make a copy of Germany's data
        c3 = manager.prototypes["Germany"].DeepCopy();
        Report("Deep cloning Germany\n=====",
            manager.prototypes["Germany"], c3);
    }
}

```

```

        // Change the capital of Germany
        c3.Capital = "Munich";
        Report("Altering Clone shallow state, prototype unaffected",
            manager.prototypes["Germany"], c3);

        // Change the language of Germany (deep data)
        c3.Language.Data = "Turkish";
        Report("Altering Clone deep state, prototype unaffected",
            manager.prototypes["Germany"], c3);

        Console.ReadKey();
    }
}

```

Listing 10.1 - Kod u okviru fajla *Program* za demonstraciju Prototype paterna

Prvo se dodaje abstraktna generička klasa *IPrototype*. Ona u sebi sadrži dve metode: *Clone* i *DeepCopy*. Metoda *Clone* koristi metodu *MemberwiseClone* koja je deo .NET Framework-a i koja je dostupna nad svim objektima. Ona kopira vrednosti svih polja i referenci i vraća referencu na kreiranu kopiju. Međutim, ona ne kopira ono na šta pokazuju reference u okviru objekata. Drugim rečima, ona ostvaruje plitko kopiranje. Ona je prihvatljiva nad objektima koji nemaju reference. Da bi se mogle čuvati kompletne vrednosti objekata, kao i njihovih podobjekata, potrebno je koristiti duboko kopiranje.

Kreiranje opšteg algoritma koji omogućava da se obuhvati svaki link u okviru strukture i da se data struktura ponovo kreira na drugom mestu je u .NET obuhvaćeno u okviru procesa koji se zove serijalizacija. Objekti se kopiraju na zadatu destinaciju, i mogu se ponovo, ukoliko je to potrebno, vratiti. Kao destinacija u serijalizaciji se mogu odabrati diskovi, internet, ali i sam memorijski prostor. Duboko kopiranje sadrži serijalizaciju i deserijalizaciju u jednoj metodi. Ova metoda *DeepCopy* se nalazi u okviru apstraktne metode *IPrototype*.

Nakon pokretanja aplikacije, dobija se ispis koji je prikazan na slici 10.2.

```

file:///C:/Users/Zdravko/Documents/Visual Studio 2015/Projects/SE/Demo/Soft...
Shallow cloning Australia
=====
Prototype Australia      Canberra      ->English
Clone Australia          Canberra      ->English
Altered Clone's shallow state, prototype unaffected
Prototype Australia      Canberra      ->English
Clone Australia          Sydney        ->English
Altering Clone deep state: prototype affected *****
Prototype Australia      Canberra      ->Chinese
Clone Australia          Sydney        ->Chinese
Deep cloning Germany
=====
Prototype Germany        Berlin      ->German
Clone Germany            Berlin      ->German
Altering Clone shallow state, prototype unaffected
Prototype Germany        Berlin      ->German
Clone Germany            Munich      ->German
Altering Clone deep state, prototype unaffected
Prototype Germany        Berlin      ->German
Clone Germany            Munich      ->Turkish

```

Slika 10.2 - Ispis nakon pokretanja aplikacije

Metoda *main* sadrži više primera koji ilustruju efekte kloniranja i dubokog kopiranja. U prvom primeru, Australia je plitko kopirana. Nakon toga vršimo izmene u okviru klona, tako da je glavni grad

u okviru prototipa Canberra a u okviru klona Sydney. Međutim, izmena jezika u Chinese menja vrednost i u prototipu i u klonu, što svakako nije željeno ponašanje. Ovo je dobijeno zbog plitkog kopiranja jer u okviru njega prototip i klon referenciraju isti *DeeperData* objekat.

U sledećem primeru smo klonirali Germany korišćenjem dubokog kopiranja. U ispisu možemo videti da u ovom slučaju radi korektno i izmena vrednosti za glavni grad i izmena vrednosti za jezik.

10.1.3.REALNI PRIMER

Kao realni primer za Prototype patern, proširićemo aplikaciju sa slikama koju smo kreirali u okviru Composite paterna. Uključićemo arhiviranje korišćenjem sledećih komandi:

```
Archive set
Retrieve set
Display archive
```

Arhiva predstavlja dodatnu komponentu na istom nivou kao i album. Da bi arhivirali određeni album koristimo naredbu:

```
archive = point.Share(parameter, archive);
```

Da bi vratili album (Retrieve) koristimo naredbu:

```
point = arhive.Share(parameter, album);
```

U cilju demonstracije ovog paterna, kreiramo novu konzolnu aplikaciju u okviru Visual Studio-a pod imenom Softversko2.Prototype.Images. U okviru projekta provo dodajemo klasu *CompositeContainer* sa sledećim kodom:

```
namespace Softversko2.Prototype.Images
{
    class CompositeContainer
    {
        // The Interface
        public interface IComponent<T>
        {
            void Add(IComponent<T> c);
            IComponent<T> Remove(T s);
            string Display(int depth);
            IComponent<T> Find(T s);
            IComponent<T> Share(T s, IComponent<T> home);
            string Name { get; set; }
        }

        // The Composite
        [Serializable()]
        public class Composite<T> : IPrototype<IComponent<T>>, IComponent<T>
        {
            List<IComponent<T>> list;

            public string Name { get; set; }

            public Composite(string name)
            {
                Name = name;
                list = new List<IComponent<T>>();
            }

            public void Add(IComponent<T> c)
            {
                list.Add(c);
            }

            // Finds the item from a particular point in the structure

```

```

// and returns the composite from which it was removed
// If not found, return the point as given
public IComponent<T> Remove(T s)
{
    holder = this;
    IComponent<T> p = holder.Find(s);
    if (holder != null)
    {
        (holder as Composite<T>).list.Remove(p);
        return holder;
    }
    else
        return this;
}

IComponent<T> holder = null;

// Recursively looks for an item
// Returns its reference or else null
public IComponent<T> Find(T s)
{
    holder = this;
    if (Name.Equals(s)) return this;
    IComponent<T> found = null;
    foreach (IComponent<T> c in list)
    {
        found = c.Find(s);
        if (found != null)
            break;
    }
    return found;
}

public IComponent<T> Share(T set, IComponent<T> toHere)
{
    IPrototype<IComponent<T>> prototype =
        this.Find(set) as IPrototype<IComponent<T>>;
    IComponent<T> copy = prototype.DeepCopy() as IComponent<T>;
    toHere.Add(copy);
    return toHere;
}

// Displays items in a format indicating their level in the structure
public string Display(int depth)
{
    String s = new String('-', depth) + "Set " + Name +
        " length :" + list.Count + "\n";
    foreach (IComponent<T> component in list)
    {
        s += component.Display(depth + 2);
    }
    return s;
}
}

// The Component
[Serializable()]
public class Component<T> : IPrototype<IComponent<T>>, IComponent<T>
{
    public string Name { get; set; }

    public Component(string name)

```

```

    {
        Name = name;
    }

    public void Add(IComponent<T> c)
    {
        Console.WriteLine("Cannot add to an item");
    }

    public IComponent<T> Remove(T s)
    {
        Console.WriteLine("Cannot remove directly");
        return this;
    }

    public string Display(int depth)
    {
        return new String('-', depth) + Name + "\n";
    }

    public IComponent<T> Find(T s)
    {
        if (s.Equals(Name)) return this;
        else
            return null;
    }

    public IComponent<T> Share(T set, IComponent<T> toHere)
    {
        IPrototype<IComponent<T>> prototype =
            this.Find(set) as IPrototype<IComponent<T>>;
        IComponent<T> copy = prototype.Clone() as IComponent<T>;
        toHere.Add(copy);
        return toHere;
    }
}
}
}

```

Listing 10.2 - Kod u okviru klase *CompositeContainer*

U okviru *Composite* klase je dodata metoda *Share* koja omogućava kopiranje albuma sa slikama. Nakon toga dodajemo abstraktnu klasu *IPrototype* kako bi omogućili primenu Prototype paterna.

```

namespace Softversko2.Prototype.Images
{
    [Serializable()]
    public abstract class IPrototype<T>
    {
        // Shallow copy
        public T Clone()
        {
            return (T)this.MemberwiseClone();
        }

        // Deep Copy
        public T DeepCopy()
        {
            MemoryStream stream = new MemoryStream();
            BinaryFormatter formatter = new BinaryFormatter();
            formatter.Serialize(stream, this);
            stream.Seek(0, SeekOrigin.Begin);
            T copy = (T)formatter.Deserialize(stream);
        }
    }
}

```

```

        stream.Close();
        return copy;
    }
}

```

Listing 10.3 - Kod u okviru abstraktne klase *IPrototype.cs*

Posledni korak je izmena koda u okviru metode *main* u klasi *Program.cs*.

```

namespace Softversko2.Prototype.Images
{
    class Program
    {
        static void Main(string[] args)
        {
            CompositeContainer.IComponent<string> album =
                new CompositeContainer.Composite<string>("Album");
            CompositeContainer.IComponent<string> point = album;
            CompositeContainer.IComponent<string> archive =
                new CompositeContainer.Composite<string>("Archive");

            // Create and manipulate a structure
            Console.WriteLine("\t\t AddSet Home");
            CompositeContainer.IComponent<string> home =
                new CompositeContainer.Composite<string>("Home");
            album.Add(home);
            point = home;

            Console.WriteLine("\t\t AddPhoto Dinner.jpg");
            point.Add(new CompositeContainer.Component<string>("Dinner.jpg"));

            Console.WriteLine("\t\t AddSet Pets");
            CompositeContainer.IComponent<string> pets =
                new CompositeContainer.Composite<string>("Pets");
            point.Add(pets);
            point = pets;

            Console.WriteLine("\t\t AddPhoto Dog.jpg");
            point.Add(new CompositeContainer.Component<string>("Dog.jpg"));
            Console.WriteLine("\t\t AddPhoto Cat.jpg");
            point.Add(new CompositeContainer.Component<string>("Cat.jpg"));

            Console.WriteLine("\t\t Find Album");
            point = album.Find("Album");

            Console.WriteLine("\t\t AddSet Garden");
            CompositeContainer.IComponent<string> garden =
                new CompositeContainer.Composite<string>("Garden");
            point.Add(garden);
            point = garden;

            Console.WriteLine("\t\t AddPhoto Spring.jpg");
            point.Add(new CompositeContainer.Component<string>("Spring.jpg"));
            Console.WriteLine("\t\t AddPhoto Summer.jpg");
            point.Add(new CompositeContainer.Component<string>("Summer.jpg"));
            Console.WriteLine("\t\t AddPhoto Flowers.jpg");
            point.Add(new CompositeContainer.Component<string>("Flowers.jpg"));
            Console.WriteLine("\t\t AddPhoto Trees.jpg");
            point.Add(new CompositeContainer.Component<string>("Trees.jpg"));

            Console.WriteLine("\t\t Display");
            Console.WriteLine(album.Display(0));
        }
    }
}

```

```

        Console.WriteLine("\t\t Find Pets");
        point = album.Find("Pets");

        Console.WriteLine("\t\t Archive Pets");
        archive = point.Share("Pets", archive);

        Console.WriteLine("\t\t Display archive");
        Console.WriteLine(archive.Display(0));

        Console.WriteLine("\t\t Find Album");
        point = album.Find("Album");

        Console.WriteLine("\t\t Remove Home");
        point = point.Remove("Home");

        Console.WriteLine("\t\t Find Album");
        point = album.Find("Album");

        Console.WriteLine("\t\t Remove Garden");
        point = point.Remove("Garden");

        Console.WriteLine("\t\t Display");
        Console.WriteLine(album.Display(0));

        Console.WriteLine("\t\t Retrive Pets");
        point = archive.Share("Pets", album);

        Console.WriteLine("\t\t Display");
        Console.WriteLine(album.Display(0));

        Console.ReadKey();
    }
}

```

Listing 10.4 - Kod u okviru metode *main* u klasi *Program.cs*

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 10.3.

```
file:///C:/Users/Zdravko/Documents/...
AddSet Home
AddPhoto Dinner.jpg
AddSet Pets
AddPhoto Dog.jpg
AddPhoto Cat.jpg
Find Album
AddSet Garden
AddPhoto Spring.jpg
AddPhoto Summer.jpg
AddPhoto Flowers.jpg
AddPhoto Trees.jpg
Display
Set Album length :2
--Set Home length :2
----Dinner.jpg
--Set Pets length :2
----Dog.jpg
----Cat.jpg
--Set Garden length :4
----Spring.jpg
----Summer.jpg
----Flowers.jpg
----Trees.jpg

Find Pets
Archive Pets
Display archive
Set Archive length :1
--Set Pets length :2
----Dog.jpg
----Cat.jpg

Find Album
Remove Home
Find Album
Remove Garden
Display
Set Album length :0

Retrive Pets
Display
Set Album length :1
--Set Pets length :2
----Dog.jpg
----Cat.jpg
```

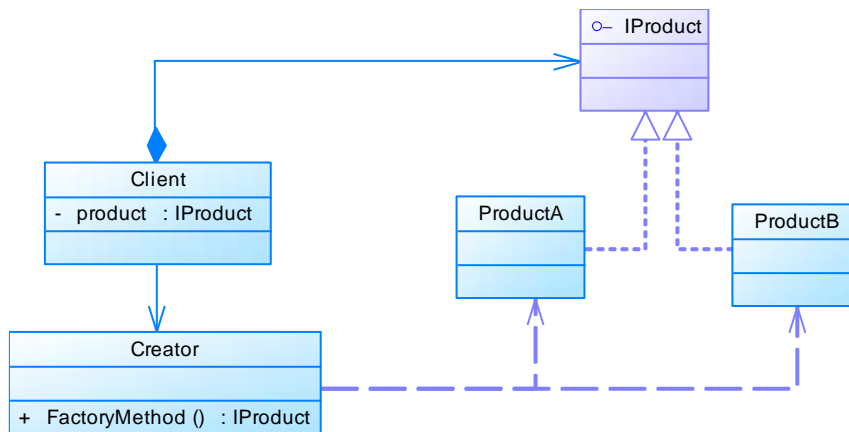
Slika 10.3 - Ispis nakon pokretanja programa

10.2. FACTORY METHOD PATTERN

Factory patern definiše interfejs za kreiranje objekta, ali omogućava podklasama da odluče instanca koje klase će biti kreirana. Factory patern omogućava da klasa prebaci instanciranje na podklase. Različite podklase mogu implementirati isti interfejs. Factory metod patern instancira odgovarajuću podklasu na osnovu informacija koje mu daje klijent ili na osnovu informacija koje izvlači iz trenutnih stanja.

10.2.1.UML DIJAGRAM ZA FACTORY METHOD PATERN

UML dijagram Factory paterna je prikazan na slici 10.4.



Slika 10.4 - UML model za Factory metohod patern

Klijent kreira promenljivu *product* ali za njeno instanciranje poziva metodu *FactoryMethod*. Ova metoda sadrži logiku koji će proizvod biti kreiran.

Uloge elemenara sa slike 10.4 su:

- *IProduct* - interfejs za proizvode
- *ProductA* i *ProductB* - klase koje implementiraju interfejs
- *Creator* - sadrži *FactoryMethod* metodu za kreiranje objekata
- *FactoryMethod* - odlučuje koju klasu će da instancira

Dizajn ovog paterna omogućava da se logika o tome koji objekat će biti kreiran, nalazi na jednom mestu.

10.2.2.DEMO PRIMER

U cilju demonstracije ovog paterna, možemo kreirati konzolnu aplikaciju pod imenom *Softversko2.FactoryPattern*. U okviru projekta dodajemo interfejs *IShape* sa sledećim kodom:

```

namespace Softversko2.FactoryPattern
{
    interface IShape
    {
        void draw();
    }
}
  
```

Listing 10.5 - Interfejs *IShape*

Nakon ovoga dodajemo konkretne klase koje implementiraju dati interfejs:

```

namespace Softversko2.FactoryPattern
{
    class Rectangle : IShape
    {
        public void draw()
        {
            Console.WriteLine("Inside Rectangle::draw() method.");
        }
    }
}
  
```

Listing 10.6 - Kod u okviru klase *Rectangle*

```

namespace Softversko2.FactoryPattern
{
    class Square : IShape
  
```

```

    {
        public void draw()
        {
            Console.WriteLine("Inside Square::draw() method.");
        }
    }
}

```

Listing 10.7 - Kod u okviru klase *Square*

```

namespace Softversko2.FactoryPattern
{
    class Circle : IShape
    {
        public void draw()
        {
            Console.WriteLine("Inside Circle::draw() method.");
        }
    }
}

```

Listing 10.8 - Kod u okviru klase *Circle*

Nakon ovoga kreiramo Factory klasu koja ima zadatak da generiše objekte konkretnih klasa na osnovu datih informacija. Dodajemo klasu *ShapeFactory.cs* koja je prikazana u listingu 10.9.

```

namespace Softversko2.FactoryPattern
{
    class ShapeFactory
    {
        public IShape getShape(String shapeType)
        {
            if (shapeType == null)
            {
                return null;
            }
            if (shapeType.Equals("CIRCLE"))
            {
                return new Circle();
            }
            else if (shapeType.Equals("RECTANGLE"))
            {
                return new Rectangle();
            }
            else if (shapeType.Equals("SQUARE"))
            {
                return new Square();
            }
            return null;
        }
    }
}

```

Listing 10.9 - Kod u okviru klase *ShapeFactory*

Poslednji korak je korišćenje Factory klase kroz *main* metodu. Zbog toga u okviru klase *Program* dodajemo sledeći kod:

```

namespace Softversko2.FactoryPattern
{
    class Program
    {
        static void Main(string[] args)

```

```

{
    ShapeFactory shapeFactory = new ShapeFactory();

    //get an object of Circle and call its draw method.
    IShape shape1 = shapeFactory.getShape("CIRCLE");
    shape1.draw();

    //get an object of Rectangle and call its draw method.
    IShape shape2 = shapeFactory.getShape("RECTANGLE");
    shape2.draw();

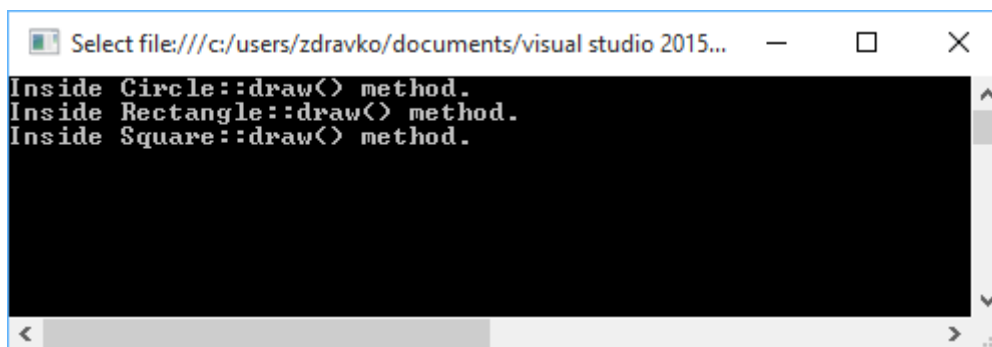
    //get an object of Square and call its draw method.
    IShape shape3 = shapeFactory.getShape("SQUARE");
    shape3.draw();

    // Wait for user
    Console.ReadKey();
}
}

```

Listing 10.10 - Kod u okviru klase *Program*

Ako sada pokrenemo aplikaciju, dobijamo ispis koji je prikazan na slici 10.5.



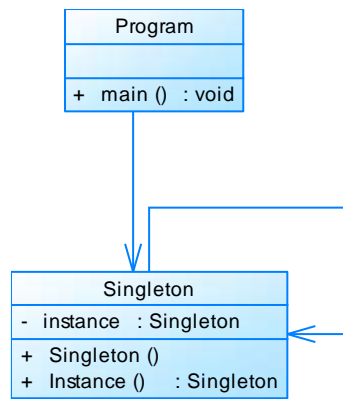
Slika 10.5 - Ispis nakon pokretanja aplikacije

10.3. SINGLETON PATTERN

Singleton pattern ograničava instanciranje klase i osigurava da samo jedna instanca date klase postoji i pruža globalnu tačku pristupa ka toj instanci. Patern osigurava da je klasa instancirana samo jednom i da su svi zahtevi upućeni ka tom jednom i samo jednom objektu

10.3.1. UML DIJAGRAM ZA SINGLETON PATTERN

UML dijagram klasa koji predstavlja singleton patern je prikazan na slici 10.6. Sa slike se može videti da kreiramo samo klasu *Singleton* koja obezbeđuje da će biti kreirana samo jedna njena instanca i klasu *Program* koja sadrži *main* metodu.



Slika 10.6 - Dijagram klasa za singleton patern

Ovaj patern obezbeđuje funkcionalnost tako što menja postojeću klasu. Modifikacije su:

- kreiramo konstruktor kao private
- dodamo private static objekat koji je interno instanciran korišćenjem private konstruktora
- dodajemo public static properti koji pristupa private objektu

Javni properti je jedini vidljiv izvan klase. Svi zahtevi za kreiranjem instance klase idu kroz ovaj properti. On pristupa privatnom static objektu i instancira ga ukoliko on već ne postoji.

10.3.2.DEMO PRIMER

U cilju demonstracije ovog paterna, možemo kreirati konzolnu aplikaciju pod imenom *Softversko2.SingletonPattern*. U okviru projekta dodajemo klasu *Singleton* sa sledećim kodom:

```

namespace Softversko2.SingletonPattern
{
    class Singleton
    {
        private static Singleton _instance;

        // Constructor is 'protected'
        protected Singleton() { }

        public static Singleton Instance()
        {
            // Note: this is not thread safe.
            if (_instance == null)
            {
                _instance = new Singleton();
            }
            return _instance;
        }
    }
}
  
```

Listing 10.11 - Kod za klasu Singleton

Nakon toga u okviru metode *main* u klasi *Program.cs* dodajemo sledeći kod:

```

namespace Softversko2.SingletonPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            // Constructor is protected -- cannot use new
            Singleton s1 = Singleton.Instance();
        }
    }
}
  
```

```

Singleton s2 = Singleton.Instance();

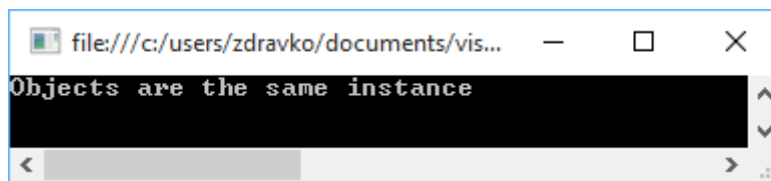
// Test for same instance
if (s1 == s2)
{
    Console.WriteLine("Objects are the same instance");
}

// Wait for user
Console.ReadKey();
}
}
}

```

Listing 10.12 - Kod u okviru klase *Program* u cilju demonstracije Singleton paterna

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 10.7.



Slika 10.7 - Ispis nakon pokretanja aplikacije

10.3.3.REALNI PRIMER

Primer realne upotrebe Singleton paterna može biti u slučaju LoadBalancing objekta (Load balancing predstavlja distribuciju zahteva na više servera u cilju postizanja što boljeg vremena odziva). Samo jedna instanca objekta koji vrši Load balancing može postojati, jer serveri mogu dinamički da postaju dostupni ili nedostupni, pa svaki zahtev mora prolaziti kroz jedan objekat koji poseduje znanje o stanju servera. Za potrebe ovog projekta kreiramo novu konzolnu aplikaciju pod imenom *Softversko2.SingletonPattern.Real*. U okviru njega ćemo dodati klasu pod imenom *LoadBalancer* sa sledećim kodom:

```

namespace Softversko2.SingletonPattern.Real
{
    class LoadBalancer
    {
        private static LoadBalancer _instance;
        private List<string> _servers = new List<string>();
        private Random _random = new Random();

        // Lock synchronization object
        private static object syncLock = new object();

        // Constructor (protected)
        protected LoadBalancer()
        {
            // List of available servers
            _servers.Add("ServerI");
            _servers.Add("ServerII");
            _servers.Add("ServerIII");
            _servers.Add("ServerIV");
            _servers.Add("ServerV");
        }

        public static LoadBalancer GetLoadBalancer()
        {
            // Support multithreaded applications through
            // 'Double checked locking' pattern which (once

```

```

        // the instance exists) avoids locking each
        // time the method is invoked
        if (_instance == null)
        {
            lock (syncLock)
            {
                if (_instance == null)
                {
                    _instance = new LoadBalancer();
                }
            }
        }

        return _instance;
    }

    // Simple, but effective random load balancer
    public string Server
    {
        get
        {
            int r = _random.Next(_servers.Count);
            return _servers[r].ToString();
        }
    }
}

```

Listing 10.13 - Kod u okviru klase *LoadBalancer.rs*

Nakon ovoga u okviru klase *Program* u metodi *main* dodajemo sledeći kod:

```

namespace Softversko2.SingletonPattern.Real
{
    class Program
    {
        static void Main(string[] args)
        {
            LoadBalancer b1 = LoadBalancer.GetLoadBalancer();
            LoadBalancer b2 = LoadBalancer.GetLoadBalancer();
            LoadBalancer b3 = LoadBalancer.GetLoadBalancer();
            LoadBalancer b4 = LoadBalancer.GetLoadBalancer();

            // Same instance?
            if (b1 == b2 && b2 == b3 && b3 == b4)
            {
                Console.WriteLine("Same instance\n");
            }

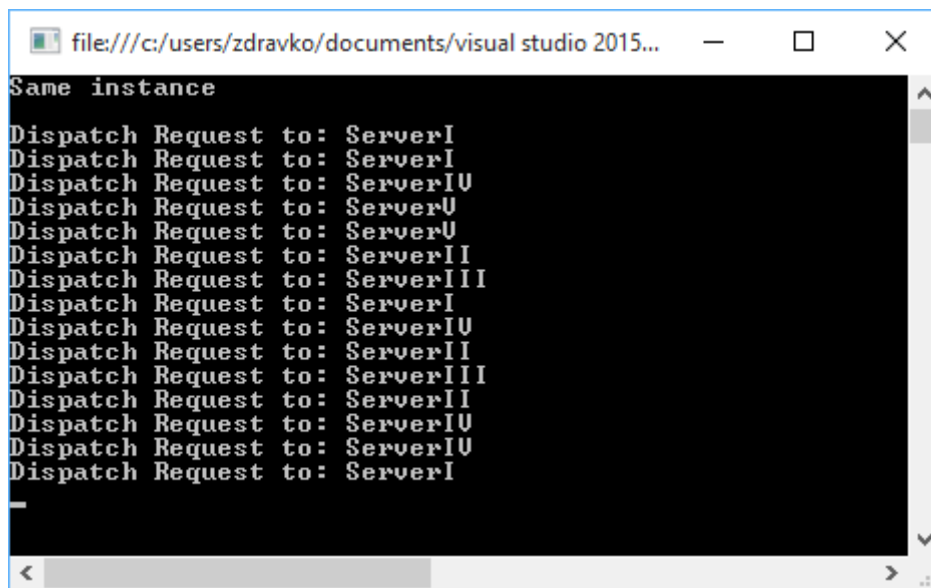
            // Load balance 15 server requests
            LoadBalancer balancer = LoadBalancer.GetLoadBalancer();
            for (int i = 0; i < 15; i++)
            {
                string server = balancer.Server;
                Console.WriteLine("Dispatch Request to: " + server);
            }

            // Wait for user
            Console.ReadKey();
        }
    }
}

```

Listing 10.14 - Kod u okviru klase *Program*

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 10.8.



```
file:///c:/users/zdravko/documents/visual studio 2015...
Same instance
Dispatch Request to: ServerI
Dispatch Request to: ServerI
Dispatch Request to: ServerIU
Dispatch Request to: ServerU
Dispatch Request to: ServerU
Dispatch Request to: ServerII
Dispatch Request to: ServerIII
Dispatch Request to: ServerI
Dispatch Request to: ServerIU
Dispatch Request to: ServerII
Dispatch Request to: ServerIII
Dispatch Request to: ServerIU
Dispatch Request to: ServerIU
Dispatch Request to: ServerI
```

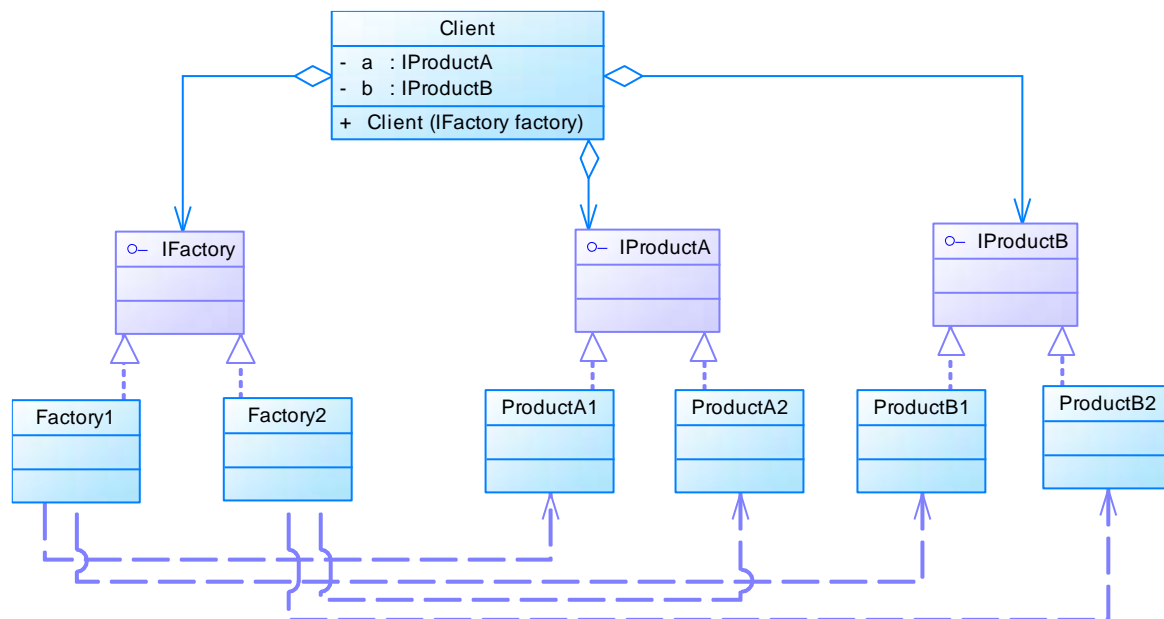
Slika 10.8 - Ispis nakon pokretanja aplikacije

10.4. ABSTRACT FACTORY PATTERN

Ovaj patern podržava kreiranje objekata koji postoje u grupama i koji su dizajnirani da zajedno postoje. Abstract factory se može razložiti na konkretne factory paterne, pri čemu svaki od njih kreira različite proizvode različitih tipova i u različitim kombinacijama. Patern izoluje definiciju proizvoda i ime njegove klase od klijenta tako da je jedini način da se on dobavi kroz factory. Zbog toga se grupe proizvoda mogu jednostavno menjati ili ažurirati bez da se menja struktura koju vidi klijent.

10.4.1. UML DIJAGRAM ZA ABSTRACT FACTORY PATTERN

UML dijagram za ovaj primer je prikazan na slici 10.9.



Slika 10.9 - UML dijagram za Abstract Factory patern

U okviru ovog paterna imamo dosta učesnika, ali je sam dizajn zapravo veoma jednostavan. Klasa *Client* poseduje konkretnu implementaciju za *IFactory* interfejs koji predstavlja *Abstract Factory* interfejs. Kroz njega, ona zahteva objekte (u primeru sa slike, tipa *IProductA* i *IProductB*). Međutim, ovo su apstraktni tipovi podataka. Konkretno factory klase određuju koji će tačno objekat klijent dobiti. Ovo omogućava da sistem bude nezavistan od toga kako se objekti kreiraju, od čega su sastavljeni i kako su implementirani.

Učesnici u okviru ovog paterna su:

- *AbstractFactory* - interfejs koji poseduje *Create* operaciju za svaki od apstraktnih objekata
- *Factory1*, *Factory2* - implementacije za sve *AbstractFactory* operacije kreiranja
- *IProductA*, *IProductB* - interfejs za tip objekta sa svojim sopstvenim operacijama
- *ProductA1*, *ProductA2*, *ProductB1*, *ProductB2* - klase koje implementiraju *AbstractProduct* interfejs i definišu objekte koji će biti kreirani od strane odgovarajućih factory klasa
- *Client* - klasa koja pristupa samo *AbstractFactory* i *AbstractProduct* interfejsima

10.4.2. PRIMER PATERNA

U okviru ovog paterna, kreiraćemo primer u kojem se poznate robne marke (npr. Guccis) kopira robnom markom sličnog imena (npr. Poochy), ali sa robom lošijeg kvaliteta.

Da bi demonstrirali primer, kreiramo u okviru Visual Studio-a konzolnu aplikaciju pod imenom *Softversko2.AbstractFactory*. U okviru aplikacije dodajemo sledeći kod:

```
namespace Softversko2.AbstractFactory
{
    interface IFactory<Brand> where Brand : IBrand
    {
        IBag CreateBag();
        IShoes CreateShoes();
    }

    // Concrete Factories (both in the same one)
    class Factory<Brand> : IFactory<Brand> where Brand : IBrand, new()
    {
        public IBag CreateBag()
        {
            return new Bag<Brand>();
        }

        public IShoes CreateShoes()
        {
            return new Shoes<Brand>();
        }
    }

    // Product 1
    interface IBag
    {
        string Material { get; }
    }

    // Product 2
    interface IShoes
    {
        int Price { get; }
    }

    // Concrete Product 1
```



```

class Bag<Brand> : IBag where Brand : IBrand, new()
{
    private Brand myBrand;

    public Bag()
    {
        myBrand = new Brand();
    }

    public string Material { get { return myBrand.Material; } }
}

// Concrete Product 2
class Shoes<Brand> : IShoes where Brand : IBrand, new()
{
    private Brand myBrand;

    public Shoes()
    {
        myBrand = new Brand();
    }

    public int Price { get { return myBrand.Price; } }
}

interface IBrand
{
    int Price { get; }
    string Material { get; }
}

class Gucci : IBrand
{
    public int Price { get { return 1000; } }
    public string Material { get { return "Crocodile skin"; } }
}

class Poochy : IBrand
{
    public int Price { get { return new Gucci().Price / 3; } }
    public string Material { get { return "Plastic"; } }
}

class Groundcover : IBrand
{
    public int Price { get { return 2000; } }
    public string Material { get { return "South african leather"; } }
}

class Client<Brand> where Brand : IBrand, new()
{
    public void ClientMain()
    {
        IFactory<Brand> factory = new Factory<Brand>();

        IBag bag = factory.CreateBag();
        IShoes shoes = factory.CreateShoes();

        Console.WriteLine("I bought a Bag which is made from " + bag.Material);
        Console.WriteLine("I bought some shoes which cost " + shoes.Price);
    }
}

```

```

class Program
{
    static void Main(string[] args)
    {
        // Call Client twice
        new Client<Poochy>().ClientMain();
        new Client<Gucci>().ClientMain();
        new Client<Groundcover>().ClientMain();

        Console.ReadKey();
    }
}

```

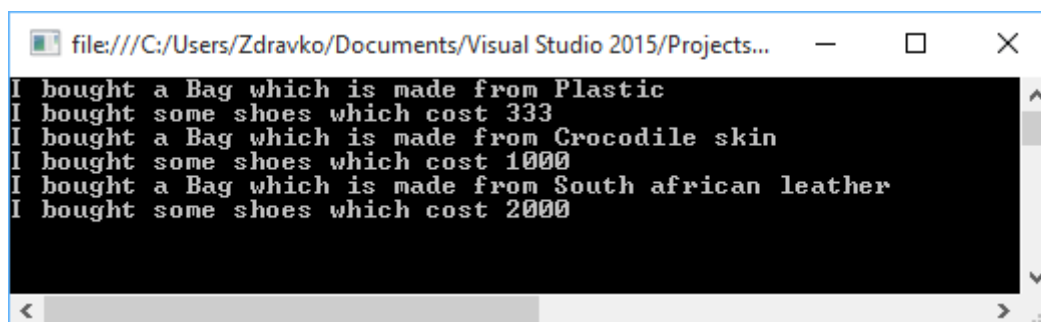
Listing 10.15 - Kod u okviru fajla *Program.cs* za demonstraciju Adapter paterna

Implementacija za Abstract Factory patern nije bazirana na običnom nasleđivanju. Ona koristi generičke tipove podataka kako bi pojednostavila kreiranje factory klasa. Umesto više factory podklasa, imamo samo jednu koja je pri tome generička. To su u listingu 10.15 interfejs *IFactory<Brand>* i *Factory<Brand>*.

Factory klasa će kreirati torbe i cipele. Pošto je struktura factory klase uvek ista, kreirali smo generičku factory klasu koja je bazirana na brendu. Ova klasa uključuje i ograničenja koja se moraju poštovati.

Proizvodi koje kreiramo u okviru ovog primera su torbe i cipele. Torbe govore od kog materijala su napravljene, a cipele prikazuju svoju cenu. Proizvode smo deklarirali kao generičke.

Nakon pokretanja aplikacije, dobijamo ispis koji je prikazan na slici 10.10



```

file:///C:/Users/Zdravko/Documents/Visual Studio 2015/Projects...
I bought a Bag which is made from Plastic
I bought some shoes which cost 333
I bought a Bag which is made from Crocodile skin
I bought some shoes which cost 1000
I bought a Bag which is made from South african leather
I bought some shoes which cost 2000

```

Slika 10.10 - Ispis nakon pokretanja aplikacije

11. DIZAJN PATERNI PONAŠANJA

Behavioral paterni se odnose na algoritme i komunikaciju između njih. Operacije koje čine neki algoritam mogu biti podeljene između različitih klasa, što unosi kompleksnost sa kojom je teško raditi. Behavioral paterni beleže kako podeliti operacije između klasa i optimizuju način na koji treba da se vrši komunikacija.

11.1. STRATEGY PATTERN

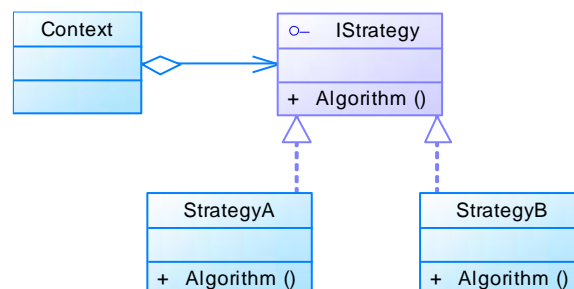
Strategy patern uključuje uklanjanje algoritma iz klase u kojoj se nalazio i njegovo prebacivanje u posebne klase. Mogu postojati različiti algoritmi (strategije) koji se mogu primeniti za posmatrani problem. Ukoliko se algoritmi nalazu u jednom fajlu odakle se i pozivaju, dobićemo teško čitljiv kod sa dosta uslovnih iskaza. Strategy patern omogućava klijentu da odabere koji algoritam želi da koristi iz familije postojećih algoritama i pruža jednostavan način da se pristupi tom algoritmu.

Neke od ključnih činjenica kada se radi o implementaciji Strategy paterna su:

- Context će vrlo često sadržati *switch* metodu ili ugnježdene *if* iskaze, kojima će se davati informacije kako bi se odredilo koju strategiju primeniti
- Ukoliko se strategije jednostavne metode, one se mogu implementirati i bez izdvajanja u posebne klase. Mehanizam delegiranja se može upotrebiti kako bi se ubacila odabrana strategija u *Context* u vreme izvršavanja
- Proširive metode se mogu koristiti kako bi se definisale nove strategije nezavisno od originalne klase koju podržavaju

11.1.1. UML DIJAGRAM ZA STRATEGY PATTERN

UML dijagram za Strategy patern je prikazan na slici 11.1.



Slika 11.1 - UML dijagram za Strategy patern

U okviru datog domena, odgovarajuća strategija se bira iz dostupne grupe strategija. Algoritam u okviru odabrane strategije se zatim izvršava do kraja.

U okviru UML dijagram vidimo nekoliko učesnika čije su uloge sledeće:

- *Context* - klasa koja poseduje osnovne informacije koji algoritam treba da se izvrši
- *IStrategy* - definiše zajednički interfejs za sve strategije
- *StrategyA*, *StrategyB* - klase koje uključuju algoritme koji implementiraju *IStrategy* interfejs

11.1.2. DEMO PRIMER

U cilju kreiranja demo primera za Strategy patern, kreiraćemo konzolnu aplikaciju u okviru Visual Studio-a pod imenom Softversko2.Strategy. U okviru aplikacije u fajl *Program.cs* dodajemo sledeći kod:

```

namespace Softversko2.Strategy
{
    // The Context
    class Context
    {
        // Context state
        public const int start = 5;
        public int Counter = 5;

        // Strategy aggregation
        IStrategy strategy = new Strategy1();

        // Algorithm invokes a strategy method
        public int Algorithm()
        {
            return strategy.Move(this);
        }

        // Changing strategies
        public void SwitchStrategy()
        {
            if (strategy is Strategy1)
                strategy = new Strategy2();
            else
                strategy = new Strategy1();
        }
    }

    // Strategy interface
    interface IStrategy
    {
        int Move(Context c);
    }

    // Strategy 1
    class Strategy1 : IStrategy
    {
        public int Move(Context c)
        {
            return ++c.Counter;
        }
    }

    // Strategy 2
    class Strategy2 : IStrategy
    {
        public int Move(Context c)
        {
            return --c.Counter;
        }
    }

    // Client
    static class Program
    {
        static void Main()
        {
            Context context = new Context();
            context.SwitchStrategy();
            Random r = new Random(37);
            for (int i = Context.start; i <= Context.start + 15; i++)
            {

```

```

        if (r.Next(3) == 2)
        {
            Console.Write("|| ");
            context.SwitchStrategy();
        }
        Console.Write(context.Algorithm() + " ");
    }
    Console.WriteLine();

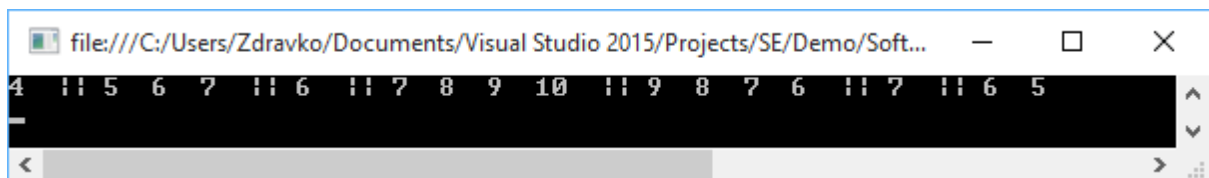
    Console.ReadKey();
}
}
}

```

Listing 11.1 - Kod u okviru fajla *Program*

U okviru primera, klijent poziva *Algorithm* metodu iz klase *Context*, koja poziva odgovarajuću metodu trenutno primenjene strategije. Strategije poseduju metodu *Move* koja broji u napred ili nazad. Slučajan broj koji se generiše u okviru klijenta određuje da li će brojanje ići u napred ili nazad.

Ako pokrenemo aplikaciju, dobijamo ispis koji je prikazan na slici 11.2.



Slika 11.2 - Prikaz nakon pokretanja aplikacije za Strategy patern

11.1.3.REALNI PRIMER

U cilju demonstracije Strategy paterna, kreiraćemo primer koji koristi više strategija sortiranja. Kreiraćemo konzolnu aplikaciju u Visual Studio-u pod nazivom Softversko2.Strategy.Sort.

```

namespace Softversko2.Strategy.Sort
{
    interface ISortStrategy<T> where T : IComparable<T>
    {
        void Sort(List<T> list);
    }

    // A 'ConcreteStrategy' class
    class QuickSort<T> : ISortStrategy<T> where T : IComparable<T>
    {
        public void Sort(List<T> list)
        {
            list.Sort(); // Default is Quicksort
            Console.WriteLine("QuickSorted list ");
        }
    }

    // A 'ConcreteStrategy' class
    class ShellSort<T> : ISortStrategy<T> where T : IComparable<T>
    {
        public void Sort(List<T> list)
        {
            //list.ShellSort(); not-implemented
            Console.WriteLine("ShellSorted list ");
        }
    }
}

```

```

// A 'ConcreteStrategy' class
class MergeSort<T> : ISortStrategy<T> where T : IComparable<T>
{
    public void Sort(List<T> list)
    {
        //list.MergeSort(); not-implemented
        Console.WriteLine("MergeSorted list ");
    }
}

// The 'Context' class
class SortedList
{
    private List<string> _list = new List<string>();
    private ISortStrategy<string> _sortstrategy;

    public void SetSortStrategy(ISortStrategy<string> sortstrategy)
    {
        this._sortstrategy = sortstrategy;
    }

    public void Add(string name)
    {
        _list.Add(name);
    }

    public void Sort()
    {
        _sortstrategy.Sort(_list);

        // Iterate over list and display results
        foreach (string name in _list)
        {
            Console.WriteLine(" " + name);
        }
        Console.WriteLine();
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Two contexts following different strategies
        SortedList studentRecords = new SortedList();

        studentRecords.Add("Samual");
        studentRecords.Add("Jimmy");
        studentRecords.Add("Sandra");
        studentRecords.Add("Vivek");
        studentRecords.Add("Anna");

        studentRecords.SetSortStrategy(new QuickSort<string>());
        studentRecords.Sort();

        studentRecords.SetSortStrategy(new ShellSort<string>());
        studentRecords.Sort();

        studentRecords.SetSortStrategy(new MergeSort<string>());
        studentRecords.Sort();

        // Wait for user
    }
}

```

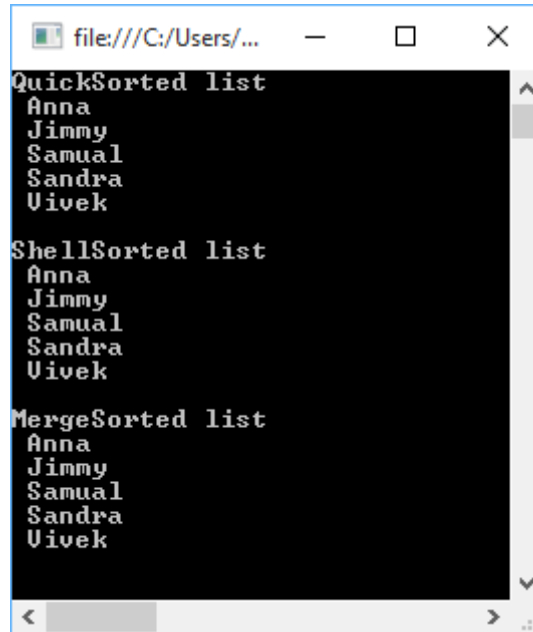
```

        Console.ReadKey();
    }
}

```

Listing 11.2 - Kod za Strategy patern primer sa sortiranjem

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 11.3.



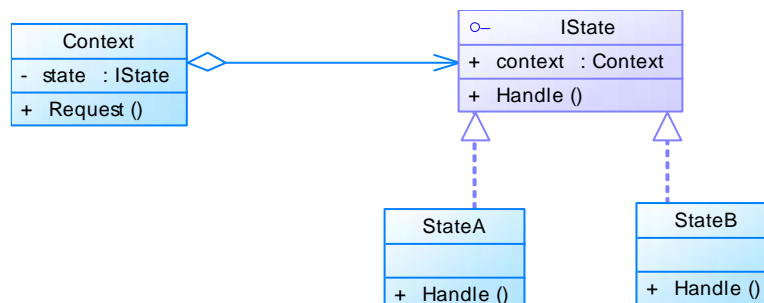
Slika 11.3 - Ispis nakon pokretanja aplikacije

11.2. STATE PATTERN

State patern se može posmatrati kao dimanička verzija Strategy paterna. Kada se stanje u okviru nekog objekta menja, on može promeniti svoje ponašanje tako što će se pozvati druge operacije. Ovo se postiže pomoću varijable tipa objekta koja se menja u svoju podklasu.

11.2.1. UML DIJAGRAM ZA STATE PATTERN

State patern sadrži zanimljivu interakciju između *Context* i *State* klase. *Context* predstavlja inormacije koje su uglavnom ne promnljive, dok *State* može da se menja u okviru raspoloživih opcija tokom izvršavanja programa. UML dijagram za ovaj patern je prikazan na slici 11.4.



Slika 11.4 - UML dijagram za State patern

U okviru dijagrama možemo primetiti više učesnika. Njihove uloge su:

- *Context* - klasa koja sadrži instancu *State* objekta koji definiše trenutno stanje i koji predstavlja interfejs od interesa za klijente

- *IState* - definiše interfejs za određeno stanje objekta *Context*
- *StateA* i *StateB* - klase koje implementiraju ponašanje zavisna od stanja objekta *Context*

Context poseduje varijablu tipa *IState* koja na početku poseduje referencu ka određenom state objektu (npr. *StateA*). Svi zahtevi se prosleđuju ka metodi *Handle* u okviru state objekta. Kao što se može videti sa UML dijagrama, state objekat ima pun pristup podacima u okviru objekta *Context*. Zbog toga, u bilo kom momentu, bilo *Context*, bilo aktivni state objekat mogu odlučiti da je vreme da se promeni stanje. Ovo se postiže tako što se state atributu iz objekta *Context* dodeli objekat drugog state tipa. Momentalno, svi zahtevi se prosleđuju novom state objektu koji može iskazati drugačije ponašanje od prethodnog state objekta.

11.2.2.DEMO PRIMER

Kao i Strategy patern, i State patern se oslanja na jednostavim mehanizmima agregacije i nasleđivanja. U primeru koji ćemo kreirati, dodajemo dve state klase, *NormalState* i *FastState* koje se razlikuju u odnosu na izmenu vrednosti brojača u okviru *Context* klase. Oba stanja implementiraju *IState* interfejs koji poseduje dve metode: *MoveUp* i *MoveDown*. Sama stanja odlučuju kada je vreme da se pređe na sledeće stanje. U okviru ovog primera, odluka je bazirana an poređenju numeričke vrednosti brojača i postavljenog ograničenja u klasi *Context*. Test program simulira operacije u 15 prolaza.

U cilju demonstracije kreiramo konzolnu aplikaciju *Softversko2.State*. U okviru fajla *Program.cs* dodajemo kod koji je prikazan u listingu 11.3

```
namespace Softversko2.State
{
    interface IState
    {
        int MoveUp(Context context);
        int MoveDown(Context context);
    }

    // State 1
    class NormalState : IState
    {
        public int MoveUp(Context context)
        {
            context.Counter += 2;
            return context.Counter;
        }

        public int MoveDown(Context context)
        {
            if (context.Counter < Context.limit)
            {
                context.State = new FastState();
                Console.WriteLine("|| ");
            }
            context.Counter -= 2;
            return context.Counter;
        }
    }

    // State 2
    class FastState : IState
    {
        public int MoveUp(Context context)
        {
            context.Counter += 5;
            return context.Counter;
        }
    }
}
```



```

    }

    public int MoveDown(Context context)
    {
        if (context.Counter < Context.limit)
        {
            context.State = new NormalState();
            Console.WriteLine("||");
        }
        context.Counter -= 5;
        return context.Counter;
    }
}

// Context
class Context
{
    public const int limit = 10;
    public IState State { get; set; }
    public int Counter = limit;

    public int Request(int n)
    {
        if (n == 2)
            return State.MoveUp(this);
        else
            return State.MoveDown(this);
    }
}

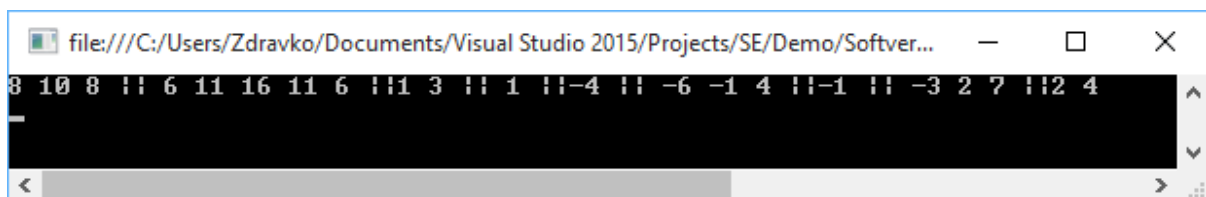
class Program
{
    static void Main(string[] args)
    {
        Context context = new Context();
        context.State = new NormalState();
        Random r = new Random(37);
        for (int i = 5; i <= 25; i++)
        {
            int command = r.Next(3);
            Console.WriteLine(context.Request(command) + " ");
        }
        Console.WriteLine();

        Console.ReadKey();
    }
}

```

Listing 11.3 - Kod u okviru fajla *Program.cs* za demonstraciju State paterna

Različita stanja mogu biti klase, ili ukoliko su vemo jednostavna, mogu biti implementirana putem delegata. Prednost State paterna jeste u tome što su prelazi između stanja jasno vidljivi. Pored toga, promena stanja se obavlja u jednom koraku (dodeljivanjem reference).



Slika 11.5 - Ispis nakon pokretanja aplikacije

11.2.3.REALNI PRIMER

U okviru primera ćemo videti kako State patern može pomoći u kreiranju vemo jednostavne igrice RPC (Run, Panic, Calm Down). U okviru ove igre, igrač može biti u jednom od četiri stanja:

- Resting
- Attacking
- Panicking
- Moving

Context objekat igrice sa sastoji od trenutnog stanja igrača. Zahtevi koji se prosleđuju u okviru igrice su pokreti koje igrač može da napravi, i upravo na njih se mora ragovati odgovarajućim stanjima.

Da bi demonstrirali ovu igricu, kreiramo konzolnu aplikaciju pod imenom Softversko2.State.RPC. U okviru fajla *Program.cs* dodajemo sledeći kod:

```
namespace Softversko2.State.RPC
{
    abstract class IState
    {
        public virtual string Move(Context context) { return ""; }
        public virtual string Attack(Context context) { return ""; }
        public virtual string Stop(Context context) { return ""; }
        public virtual string Run(Context context) { return ""; }
        public virtual string Panic(Context context) { return ""; }
        public virtual string CalmDown(Context context) { return ""; }
    }

    // There are four States
    class RestingState : IState
    {
        public override string Move(Context context)
        {
            context.State = new MovingState();
            return "You start moving";
        }
        public override string Attack(Context context)
        {
            context.State = new AttackingState();
            return "You start attacking the darkness";
        }
        public override string Stop(Context context)
        {
            return "You are already stopped!";
        }
        public override string Run(Context context)
        {
            return "You cannot run unless you are moving";
        }
        public override string Panic(Context context)
        {
            context.State = new PanickingState();
            return "You start Panicking and begin seeing things";
        }
        public override string CalmDown(Context context)
        {
            return "You are already relaxed";
        }
    }
}
```

```

class AttackingState : IState
{
    public override string Move(Context context)
    {
        return "You need to stop attacking first";
    }
    public override string Attack(Context context)
    {
        return "You attack the darkness for " +
            (new Random().Next(20) + 1) + " damage";
    }
    public override string Stop(Context context)
    {
        context.State = new RestingState();
        return "You are calm down and come to rest";
    }
    public override string Run(Context context)
    {
        context.State = new MovingState();
        return "You Run away from the fray";
    }
    public override string Panic(Context context)
    {
        context.State = new PanickingState();
        return "You start Panicking and begin seeing things";
    }
    public override string CalmDown(Context context)
    {
        context.State = new RestingState();
        return "You fall down and sleep";
    }
}

class PanickingState : IState
{
    public override string Move(Context context)
    {
        return "You move around randomly in a blind panic";
    }
    public override string Attack(Context context)
    {
        return "You start attacking the darkness, but keep on missing";
    }
    public override string Stop(Context context)
    {
        context.State = new MovingState();
        return "You are start relaxing, but keep on moving";
    }
    public override string Run(Context context)
    {
        return "You run around in your panic";
    }
    public override string Panic(Context context)
    {
        return "You are already in a panic";
    }
    public override string CalmDown(Context context)
    {
        context.State = new RestingState();
        return "You relax and calm down";
    }
}

```

```

class MovingState : IState
{
    public override string Move(Context context)
    {
        return "You move around randomly";
    }
    public override string Attack(Context context)
    {
        return "You need to stop moving first";
    }
    public override string Stop(Context context)
    {
        context.State = new RestingState();
        return "You stand still in a dark room";
    }
    public override string Run(Context context)
    {
        return "You run around in circles";
    }
    public override string Panic(Context context)
    {
        context.State = new PanickingState();
        return "You start Panicking and begin seeing things";
    }
    public override string CalmDown(Context context)
    {
        context.State = new RestingState();
        return "You stand still and relax";
    }
}

class Context
{
    public IState State { get; set; }

    public void Request(char c)
    {
        string result;
        switch (char.ToLower(c))
        {
            case 'm':
                result = State.Move(this);
                break;
            case 'a':
                result = State.Attack(this);
                break;
            case 's':
                result = State.Stop(this);
                break;
            case 'r':
                result = State.Run(this);
                break;
            case 'p':
                result = State.Panic(this);
                break;
            case 'c':
                result = State.CalmDown(this);
                break;
            case 'e':
                result = "Thank you for playing \"The RPC Game\"";
                break;
        }
    }
}

```

```

        default:
            result = "Error, try again";
            break;
    }
    Console.WriteLine(result);
}
}

class Program
{
    static void Main(string[] args)
    {
        // context.s are States
        // Decide on a starting state and hold onto the Context thus established
        Context context = new Context();
        context.State = new RestingState();
        char command = ' ';
        Console.WriteLine("Welcome to \"The State Game\"!");
        Console.WriteLine("You are standing here looking relaxed!");
        while (char.ToLower(command) != 'e')
        {
            Console.WriteLine("\nWhat would you like to do now?");
            Console.Write("    Move    Attack    Stop    Run    Panic    CalmDown\n");
            Exit the game: ==>";
            string choice;
            do
            {
                choice = Console.ReadLine();
                while (choice == null);
                command = choice[0];
                context.Request(command);
            }
        }
    }
}

```

Listing 11.4 - Kod u okviru fajla *Progam.cs* za demonstraciju State paterna u okrifu RPC igrice

Ako pogledamo detaljnije kod, možemo videti da svako od četiri stanja implementira šest akcija na potpuno drugačiji način. Sve akcije vraćaju tekst koji objašnjava šta se desilo, ali neke od njih menjaju i stanje. Metoda *Move* iz klase *RestingState* ima za cilj da promeni stanje u *MovingState* i da vrati odgovarajuću poruku.

Ako pokrenemo aplikaciju, možemo dobiti ispis koji je prikazan na slici 11.6.

```

file:///C:/Users/Zdravko/Documents/Visual Studio 2015/Projects/SE/Real/Softversko2.State.RPC/...
Welcome to "The State Game"!
You are standing here looking relaxed!

What would you like to do now?
  Move   Attack   Stop   Run   Panic   CalmDown   Exit the game: ==>m
You start moving

What would you like to do now?
  Move   Attack   Stop   Run   Panic   CalmDown   Exit the game: ==>a
You need to stop moving first

What would you like to do now?
  Move   Attack   Stop   Run   Panic   CalmDown   Exit the game: ==>r
You run around in circles

What would you like to do now?
  Move   Attack   Stop   Run   Panic   CalmDown   Exit the game: ==>p
You start Panicking and begin seeing things

What would you like to do now?
  Move   Attack   Stop   Run   Panic   CalmDown   Exit the game: ==>c
You relax and calm down

What would you like to do now?
  Move   Attack   Stop   Run   Panic   CalmDown   Exit the game: ==>a
You start attacking the darkness

What would you like to do now?
  Move   Attack   Stop   Run   Panic   CalmDown   Exit the game: ==>_

```

Slika 11.6 - Ispis nakon pokretanja aplikacije

11.3. TEMPLATE METHOD PATTERN

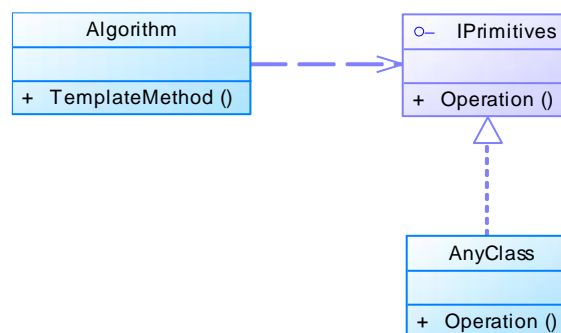
Template method patern omogućava algoritmima da prepuste određene korake podklasama. Struktura algoritma se ne menja, ali se određeni delovi operacija izvršavaju u drugim klasama.

Ovaj patern se može koristiti zajedno sa Strategy paternom. Bilo koji sistem koji želi da izvršavanje određenih operacija prepusti podklasama će imati korist od ovog paterna.

Ključna prednost kod ovog paterna jeste da može da rukuje sekvencama višestukih poziva ka metodama, gde se neke implementacije metoda prebacuju na podklase (ili druge druge klase putem Strategy paterna). Programer odlučuje koji su delovi algoritma fiksni, a koji se mogu menjati. Ne promenljivi delovi se implementiraju u baznim klasama (roditeljima), dok se promenljivim delovima ili daje podrazumevano ponašanje, ili im se uopšte ne daje implementacija. Programer odlučuje koji su zahtevani koraci u okviru algoritma i redosled u tim koracima, ali omogućava klijentu da proširi ili zameni neke od koraka.

11.3.1. UML DIJAGRAM ZA TEMPLATE METHOD

UML dijagram za Template method je prikazan na slici 11.7.



Slika 11.7 - UML dijagram za Template method patern

Pomoću UML dijagram je prikazana klasa algoritma (*Algorithm*) koja koristi *IPrimitives* interfejs kako bi se povezala sa metodama koje su prepuštenje klasama *AnyClass*. Učesnici u okviru ovog paterna su:

- *Algorithm* - klasa koja uključuje *TemplateMethod*
- *TemplateMethod* - metoda koja prepušta određene delove izvršavanja drugim klasama
- *IPrimitives* - interfejs koji definiše operacije koje *TemplateMethod* prepušta drugim klasama
- *AnyClass* - klasa koja implementira *IPrimitives* interfejs
- *Operation* - jedna od metoda koje su potrebne metodi *TemplateMethod* kako bi završila svoj rad

11.3.2.DEMO PRIMER

Kao demo primer, u okviru Visual Studio-a kreiramo konzolnu aplikaciju Softversko2.Template. Kod u okviru fajla *Program.cs* je prikazan u listingu 11.5.

```
namespace Softversko2.Template
{
    interface IPrimitives
    {
        string Operation1();
        string Operation2();
    }

    class Algorithm
    {
        public void TemplateMethod(IPrimitives a)
        {
            string s =
                a.Operation1() +
                a.Operation2();
            Console.WriteLine(s);
        }
    }

    class ClassA : IPrimitives
    {
        public string Operation1()
        {
            return "ClassA:Op1 ";
        }
        public string Operation2()
        {
            return "ClassA:Op2 ";
        }
    }

    class ClassB : IPrimitives
    {
        public string Operation1()
        {
            return "ClassB:Op1 ";
        }
        public string Operation2()
        {
            return "ClassB:Op2 ";
        }
    }
}
```

```

class Program
{
    static void Main(string[] args)
    {
        Algorithm m = new Algorithm();

        m.TemplateMethod(new ClassA());
        m.TemplateMethod(new ClassB());

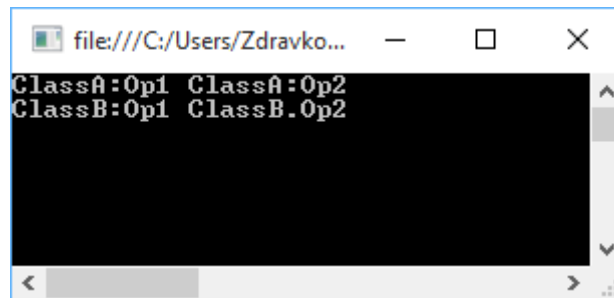
        Console.ReadKey();
    }
}

```

Listing 11.5 - Kod u okviru fajla *Program.cs* za demonstraciju Template method paterna

Prethodni primer je u skladu sa UML dijagramom sa slike 11.7. Klasa *Algorithm* sadrži metodu *TemplateMethod* i koristi *IPrimitives* interfejs kako bi pristupila klasama koje implementiraju ovaj interfejs. U okviru metode *main*, metoda *TemplateMethod* se poziva tako što joj se proslede objekti tipa *ClassA* i *ClassB*. Metoda *TemplateMethod* zna da mora pozvati *Operations* metode nad objektom tipa *IPrimitives*, ali on ne mora da zna koja će to konkretna klasa biti.

Ako pokrenemo aplikaciju, dobićemo ispis koji je prikazan na slici 11.8.



Slika 11.8 - Ispis koji se prikazuje nakon pokretanja aplikacije

11.3.3.REALNI PRIMER

Realni primer će koristiti *TemplateMethod* metodu pod imenom *ExportFormattedData* koja predstavlja okvir za pozivanje više metoda: *ReadData*, *FormatData* i *ExportData*.

U cilju demonstracije ovog paterna, kreiramo konzolnu aplikaciju u okviru Visual Studio-a pod nazivom *Softversko2.Template.DataAccess*. U okviru fajla *Program.cs* dodajemo sledeći kod:

```

namespace Softversko2.Template.DataAccess
{
    abstract class DataExporter
    {
        // This will not vary as the data is read from sql only
        public void ReadData()
        {
            Console.WriteLine("Reading the data from SqlServer");
        }

        // This will not vary as the format of report is fixed.
        public void FormatData()
        {
            Console.WriteLine("Formating the data as per requiriements.");
        }

        // This may vary based on target file type choosen
        public abstract void ExportData();
    }
}

```



```

        // This is the template method that the client will use.
        public void ExportFormattedData()
        {
            this.ReadData();
            this.FormatData();
            this.ExportData();
        }
    }

    class PDFExporter : DataExporter
    {
        public override void ExportData()
        {
            Console.WriteLine("Exporting the data to a PDF file.");
        }
    }

    class ExcelExporter : DataExporter
    {
        public override void ExportData()
        {
            Console.WriteLine("Exporting the data to an Excel file.");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            DataExporter exporter = null;

            // Lets export the data to Excel file
            exporter = new ExcelExporter();
            exporter.ExportFormattedData();

            Console.WriteLine();

            // Lets export the data to PDF file
            exporter = new PDFExporter();
            exporter.ExportFormattedData();

            // Wait for user
            Console.ReadKey();
        }
    }
}

```

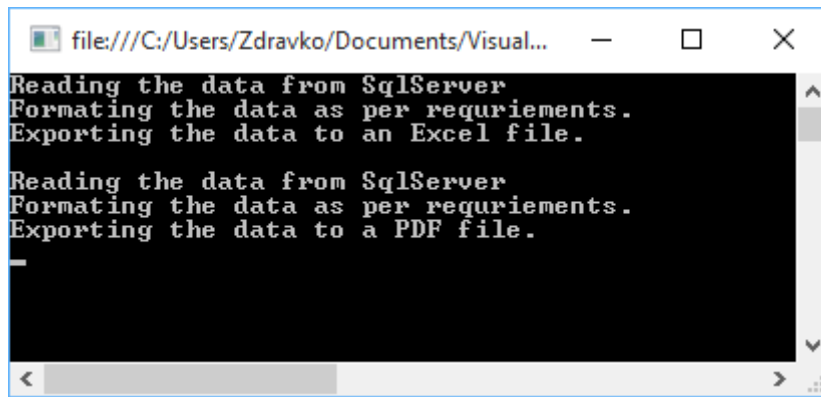
Listing 11.6 - Kod u okviru fajla *Program.cs* koji demonstrira realnu upotrebu Template method paterna

U okviru prethodnog primera, mi imamo klasu *DataExporter.cs* koja čita podatke iz baze podataka, zatim ih formatira i na kraju esportuje u željenom formatu.

U okviru klase, metode *ReadData* i *FormatData* se neće menjati pa su potpuno implementirane i ne mogu se preklapati. Jedini promenljivi deo je metoda *ExportData* čija implementacija će se menjati u zavisnosti od odabranog formata u koji eksportujemo podatke. Zbog toga, ako eksportujemo podatke u excel, potrebna nam je posebna klasa za to. Zbog toga dodajemo klase *ExcelExporter* i *PDFExporter* za esportovanje u excel i pdf format, respektivno.

Prednost kod ovog paterna možemo videti u okviru *main* metode. Klasa *DataExporter* će biti korišćena od strane aplikacije, a odgovarajuća implementacija algoritma će biti uzeta iz izvedene klase.

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 11.9.



Slika 11.9 - Ispis nakon pokretanja aplikacije

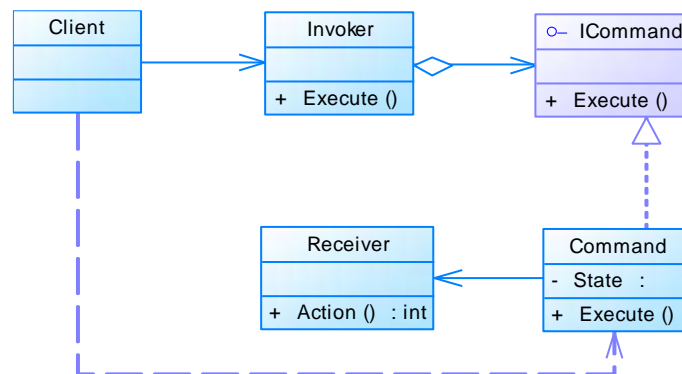
11.4. COMMAND PATTERN

Command patern kreira distancu između klijenata koji zahtevaju operacije i objekata koji ih izvršavaju. Patern je izrazito višestran. On može da podrži:

- slanje zahteva ka različitim objektima
- smeštanje zahteva u redove, logovanje i odbijanje zahteva
- kreiranje transakcija višeg nivoa od primitivnih operacija
- Redo i Undo funkcionalnosti

11.4.1.UML DIJAGRAM ZA COMMAND PATERN

UML dijagram za Command patern je prikazan na slici 11.10.



Slika 11.10 - UML dijagram za Command patern

Client poseduje određeni način na koji će reći šta je potrebno. On razmišlja u okviru komandi kao što su *Cut*, *Redo*, *Open*, itd. Objekti *Receiver*, kojih može biti više, znaju kako da obrade ove zahteve. Npr. komanda *Cut* za tekst se izvršava u okviru drugog dela sistema od komande *Cut* za slike jer su implementacije ovih naredbi drugačije.

Klasa *Client* upućuje zahtev ka metodi *Execute* u okviru klase *Invoker*, nakon čega zahtev ide do klase *Command* a zatim do metode *Action* u okviru klase *Receiver*. U programu može biti više zahteva različitih tipova koji se prosleđuju ka različitim *Receiver* objektima. Interfejs *ICommand* osigurava da svi imaju odgovarajuću standardnu formu.

U okviru UML dijagrama možemo videti više učesnika čije su uloge sledeće:

- *Client* - kreira i izvršava komande
- *ICommand* - interfejs koji navodi operacije *Execute* koje se mogu izvršiti

- *Invoker* - poziva klasu *Command* da izvrši određenu akciju
- *Command* - klasa koja implementira *Execute* operaciju tako što uključuje operacije iz klasa *Receiver*
- *Receiver* - klasa koja može da izvrši zahtevanu akciju
- *Action* - operaciju koju je potrebno izvršiti

Command patern na prvi pogled ima puno učesnika, ali se neki od njih odbacuju kada se koriste delegati, kao što će biti prikazano u demo primeru.

11.4.2.DEMO PRIMER

Da bi kreirali jednostavan primer za Command patern, kreiramo konzolnu aplikaciju u okviru Visual Studio-a pod imenom Softversko2.Command. U okviru projekta dodajemo sledeći kod u okviru fajla *Program.cs*.

```
namespace Softversko2.Command
{
    delegate void Invoker();

    class Command
    {
        public static Invoker Execute, Undo, Redo;

        public Command(Receiver receiver)
        {
            Execute = receiver.Action;
            Redo = receiver.Action;
            Undo = receiver.Reverse;
        }
    }

    public class Receiver
    {
        string build, oldbuild;
        string s = "some string ";

        public void Action()
        {
            oldbuild = build;
            build += s;
            Console.WriteLine("Receiver is adding " + build);
        }

        public void Reverse()
        {
            build = oldbuild;
            Console.WriteLine("Receiver is reverting to " + build);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Command c = new Command(new Receiver());

            Command.Execute();
            Command.Redo();
            Command.Undo();
        }
    }
}
```

```

        Command.Execute();

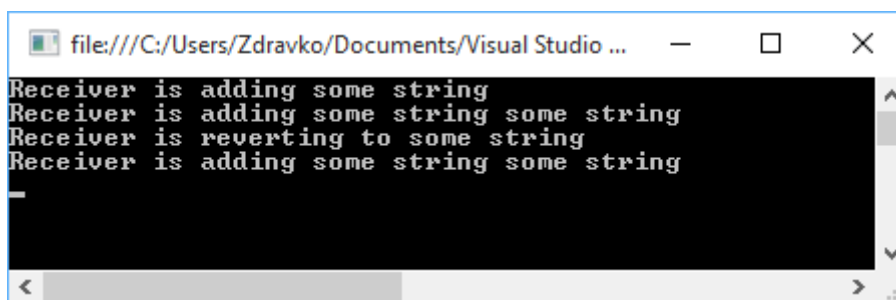
        Console.ReadKey();
    }
}

```

Listing 11.7 - Kod u okviru fajla *Program.cs* za demo primer Command paterna

Klasa *Invoker* i interfejs *ICommand* su implementirani zajedno kroz tip delegata. Imena objekata koji su tipa delegata (*Invokers*) su bazirana na onome što klijent želi. Njihova imena su *Execute*, *Redo* i *Undo*. Konstruktor u klasi *Command* povezuje objekte tipa delegata sa dve metode u okviru klase *Receiver*: *Action* i *Reverse*. Objekti *Exeute* i *Redo* su oba povezana sa metodom *Action*, a delegat *Undo* je povezan sa meotodom *Reverse*. Klasa *Receiver* vodi računa o stanju objekta i odgovorna je za ispis koji se dobija pokretanjem aplikacije.

Ako pokrenemo aplikaciju dobijamo ispis koji je prikazan na slici 11.11.



Slika 11.11 - Ispis nakon pokretanja aplikacije

11.4.3.REALNI PRIMER

U okviru ovog primera ćemo kreirati deo menija. Posmatramo samo mali deo menija sa komandama Paste i Print. Pretpostavimo da postoji javni clipboard, gde klijent može da postavi tekst. Operacije lepljenja (paste) i štampanja radi klasa receiver, koja čuva trenutnu kopiju dokumenta. Pored toga, tu su komande Undo i Redo koje se mogu, ili se ne mogu izvršiti (npr. komanda Undo posle Print komande nema smisla).

Postoji samo jedan tip za invoker objekat, ali će komande za Paste i Print implementirati Execute, Undo i Redo invoker objekte na drugačiji način. Zbog toga, ćemo ih smestiti u abstraktrnu klasu koju će command klase moći da naslede. Ovim je osigurana uniformnost za command tipove, ali i za logovanje.

Pošto su command objekti odvojeni od client i receiver objekata, oni mogu prihvatiti sistemske funkcionalnosti kao što je npr. logovanje. Jednostavna logging operacija jeste da se izbroji koliko puta je neka komanda pozvana. Da bi implementirali logovanje, definisane su proširive metode nad invoker tipom delegata kako bi se omogućilo:

- metoda *Log* koja povećava brojač
- metoda *Count* koja vraća trenutnu vrednost za brojač

Pošto je *count* polje statičko, istoj vrednosti pristupaju svi pozivi ka metodama.

U cilju demonstracije ovog primera, kreiramo konzolnu aplikaciju u okviru Visual Studio-a pod imenom Softversko2.Command.Menu. U okviru projekta dodajemo sledeći kod u fajlu *Program.cs*:

```

namespace Softversko2.Command.Menu
{
    delegate void Invoker();

    static class InvokerExtensions
    {

```

```

    static int count;

    public static int Count(this Invoker invoker)
    {
        return count;
    }

    public static void Log(this Invoker invoker)
    {
        count++;
    }
}

abstract class ICommand
{
    public Invoker Execute, Redo, Undo;
}

// Command 1
class Paste : ICommand
{
    public Paste(Document document)
    {
        Execute = delegate { Execute.Log(); document.Paste(); };
        Redo = delegate { Redo.Log(); document.Paste(); };
        Undo = delegate { Undo.Log(); document.Restore(); };
    }
}

// Command 2 - without an Undo method
class Print : ICommand
{
    public Print(Document document)
    {
        Execute = delegate { Redo.Log(); document.Print(); };
        Redo = delegate { Redo.Log(); document.Print(); };
        Undo = delegate { Redo.Log(); Console.WriteLine(
            "Cannot undo a Print "); };
    }
}

public class ClipboardSingleton
{
    public string Clipboard { get; set; }

    // Private Constructor
    ClipboardSingleton() { }

    // Private object instantiated with private constructor
    static readonly ClipboardSingleton instance = new ClipboardSingleton();

    // Public static property to get the object
    public static ClipboardSingleton UniqueInstance
    {
        get { return instance; }
    }
}

// Receiver
class Document
{
    string name;

```

```

    string oldpage, page;

    public Document(string name)
    {
        this.name = name;
    }

    public void Paste()
    {
        oldpage = page;
        page += ClipboardSingleton.UniqueInstance.Clipboard + "\n";
    }

    public void Restore()
    {
        page = oldpage;
    }

    public void Print()
    {
        Console.WriteLine(
            "File " + name + " at " + DateTime.Now + "\n" + page);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Document document = new Document("Greetings");
        Paste paste = new Paste(document);
        Print print = new Print(document);

        ClipboardSingleton.UniqueInstance.Clipboard = "Hello, everyone";
        paste.Execute();
        print.Execute();
        paste.Undo();

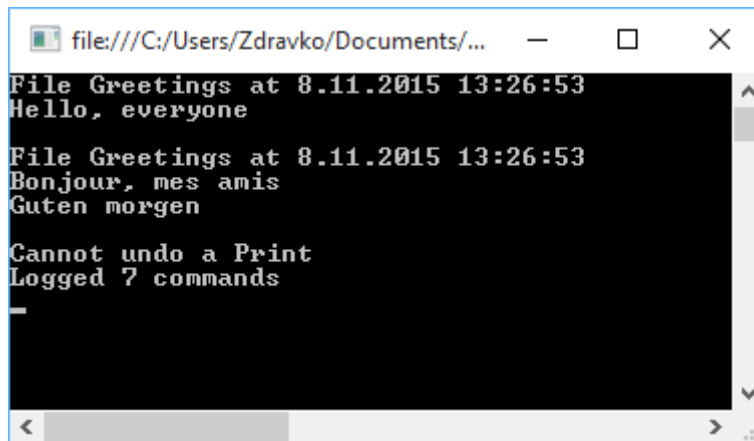
        ClipboardSingleton.UniqueInstance.Clipboard = "Bonjour, mes amis";
        paste.Execute();
        ClipboardSingleton.UniqueInstance.Clipboard = "Guten morgen";
        paste.Redo();
        print.Execute();
        print.Undo();

        Console.WriteLine("Logged " + paste.Execute.Count() + " commands");

        Console.ReadKey();
    }
}

```

Listing 11.8 - Kod u okviru fajla *Program.cs* koji demonstrira Command patern
 Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 11.12.



```
file:///C:/Users/Zdravko/Documents/...
File Greetings at 8.11.2015 13:26:53
Hello, everyone

File Greetings at 8.11.2015 13:26:53
Bonjour, mes amis
Guten morgen

Cannot undo a Print
Logged 7 commands
```

Slika 11.12 - Ispis nakon pokretanja programa za Command patern

11.5. ITERATOR PATTERN

Iterator patern omogućava sekvencijalni pristup elementima u kolekciji a da se pri tome ne mora znati njena struktura. Pored toga, patern omogućava filtriranje elemenata na različite načine.

Koncept iteratora i enumeratora postoji već duže vreme. Enumeratori su odgovorni za dobavljanje sledećeg elementa u sekvenci koju definiše određeni kriterijum. Za takvu sekvencu se kaže da je *enumerable*. Primeri mogu biti sledeći ceo broj, ili sledeći naručen proizvod sortiran po datumu. Iteratori imaju svrhu za prolazak kroz sekvencu elemenata od početka do kraja.

Iterator patern je patern koji je dobio najviše podrške iz promena u dizajnu jezika u poslednjim godinama. Iterator patern definiše osnovna razdvajanja između procesa iteracije i enumeracije. Prednost ovog paternu se može uvideti u situacijama kada je:

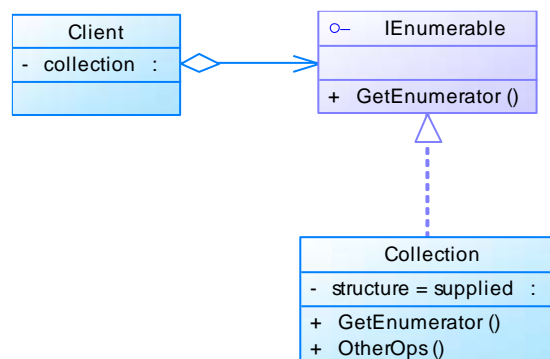
- struktura kompleksna
- potrebne se različite iteracije, i to potencijalno u isto vreme
- ista struktura i iteracije se mogu primeniti nad različitim podacima

Važno je uvideti jasne razlike između enumeratora i iteratora:

- *Enumeratori* - predstavljaju sve kolekcije, kako generičke tako i ne generičke. Podržani su od strane *IEnumerable* interfejsa i putem *yield return* naredbe
- *Iteratori* - pišu ih klijenti a podržani su pomoću *foreach* naredbe i *query* izraza

11.5.1.UML DIJAGRAM ZA ITERATOR PATTERN

UML dijagram za Iterator patern je prikazan na slici 11.13.



Slika 11.13 - UML dijagram za Iterator patern

Sa slike se vidi da je ime interfejsa *IEnumerable* a da je ime metode *GetEnumerator*. Ova imena su fiksna. Metoda *GetEnumerator* se automatski poziva kada se izvršava naredba *foreach*. Druge metode za prolazak kroz sekvencu se takođe mogu implementirati i pozivati iz klase *Client*. U dijagramu je ovo prikazano metodom *OtherOps*.

Glavni učesnici u okviru dijagrama su:

- *Client* - čuva kolekciju objekata i koristi naredbu *foreach* kako bi prolazio kroz njih
- *IEnumerable* - interfejs definisan u okviru C#-a koji sadrži metodu *GetEnumerator*
- *Collection* - tip podataka koji ima mogućnost da generiše kolekcije vrednosti
- *GetEnumerator* - metoda koja sadrži vrednosti u kolekciji
- *OtherOps* - druge metode koje vraćaju kolekcije vrednosti u drugačijim sekvencama sa primenjenim drugačijim filterima i transformacijama

11.5.2.DEMO PRIMER

Kao demo primer, kreiraćemo aplikaciju koja ispisuje sadržaj niza. Ovde je jednostavno uočiti učesnike u paternu. Client je *main* metoda. Ona skuplja kolekciju objekata i prolazi kroz nju pomoću *foreach* petlje. Naredba *foreach* koristi *GetEnumerator* metodu iz klase kolekcije. Ova metoda koristi naredbu *yield return*. Namena ove naredbe jeste da vrati samo vrednost svaki put kada se objekat kojeg vraća *GetEnumerator* poziva iz naredbe *foreach* u okviru Client koda. Naredba *foreach* iz metode *GetEnumerator* pamti gde se nalazila posle poslednjeg *yield return* iskaza i vraća sledeću vrednost.

Da bi demonstrirali ovaj patern, u okviru Visual Studio-a kreiramo konzolnu aplikaciju pod imenom Softversko2.Iterator i u okviru nje sledeći kod:

```
namespace Softversko2.Iterator
{
    class MonthCollection : IEnumerable
    {
        string[] months = {"January", "February", "March",
                           "April", "May", "June",
                           "July", "August", "September",
                           "October", "November", "December"};

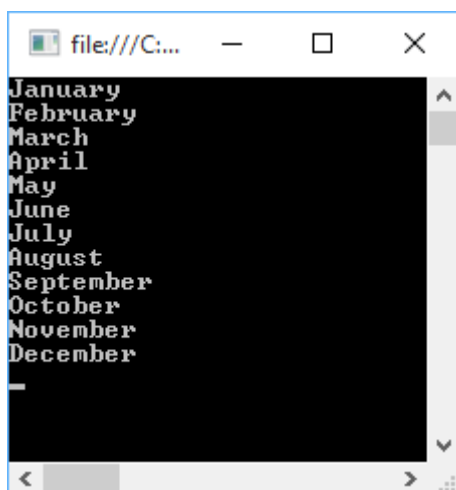
        public IEnumerator GetEnumerator()
        {
            // Generates values from the collection
            foreach (string element in months)
                yield return element;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            MonthCollection collection = new MonthCollection();
            // Consumes values generated from collection's GetEnumerator method
            foreach (string n in collection)
                Console.WriteLine(n);

            Console.ReadKey();
        }
    }
}
```

Listing 11.9 - Kod u okviru fajla *Program.cs* za demonstraciju Iterator paterna

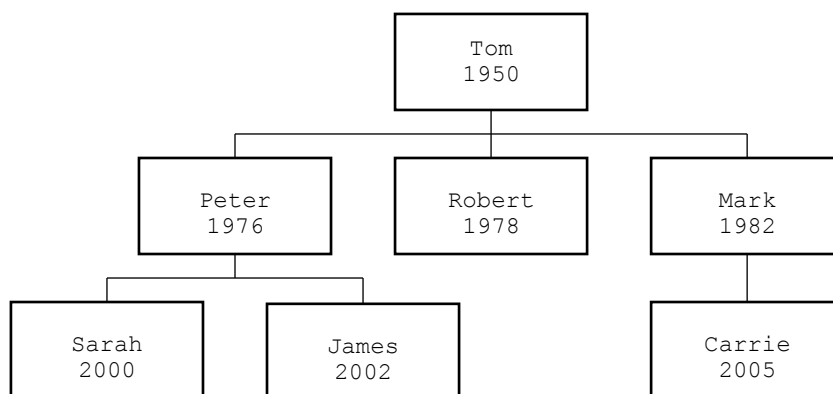
Nakon pokretanja aplikacije, dobijamo ispis koji je prikazan na slici 11.14.



Slika 11.14 - Ispis nakon pokretanja aplikacije

11.5.3.REALNI PRIMER

Pretpostavimo da imamo porodično stablo, kao što je prikazano na slici 11.15. Cilj ovog primera jeste da se napravi prolazak kroz stablo i da se odaberu ljudi koji zadovoljavaju određeni kriterijum kako bi se oni štampali.



Slika 11.15 - Primer za Iterator patern - porodično stablo

Pošto se broj veza jedne osobe sa drugim osobama ne može predvideti, porodično stablo se obično predstavlja kao binarno stablo, gde se levo povezuje prvo rođeno dete, a desno od njega sledeći potomci.

Da bi demonstrirali ovaj primer, u okviru Visual Studio-a kreiramo konzolnu aplikaciju pod imenom Softversko2.Iterator.Tree. U okviru nje dodajemo klasu *Person.cs* sa sledećim kodom:

```

namespace Softversko2.Iterator.Tree
{
    class Person
    {
        public Person() { }
        public string Name { get; set; }
        public int Birth { get; set; }

        public Person(string name, int birth)
        {
            Name = name;
            Birth = birth;
        }
    }
}

```

```

        public override string ToString()
        {
            return "[" + Name + ", " + Birth + "]";
        }
    }
}

```

Listing 11.10 - Kod u okviru klase *Person.cs*

Nakon ovoga dodajemo klasu *Node.cs* sa sledećim kodom:

```

namespace Softversko2.Iterator.Tree
{
    class Node<T>
    {
        public Node() { }
        public Node<T> Left { get; set; }
        public Node<T> Right { get; set; }
        public T Data { get; set; }

        public Node(T d, Node<T> left, Node<T> right)
        {
            Data = d;
            Left = left;
            Right = right;
        }
    }
}

```

Listing 11.11 - Kod u okviru klase *Node.cs*

Struktura drveta je implementirana kroz klasu *Tree.cs*. Klasa predstavlja kolekciju koju koristimo da pri prikazali Iterator patern. Klasa je generička, a u ovom primeru, *T* predstavlja tip *Person*.

```

namespace Softversko2.Iterator.Tree
{
    // T is the data type. The Node type is built-in
    class Tree<T>
    {
        Node<T> root;
        public Tree() { }
        public Tree(Node<T> head)
        {
            root = head;
        }

        public IEnumerable<T> Preorder
        {
            get { return ScanPreorder(root); }
        }

        // Enumerator with Filter
        public IEnumerable<T> Where(Func<T, bool> filter)
        {
            foreach (T p in ScanPreorder(root))
            {
                if (filter(p) == true)
                    yield return p;
            }
        }

        // Enumerator with T as Person
        private IEnumerable<T> ScanPreorder(Node<T> root)
        {
            yield return root.Data;
            if (root.Left != null)
                foreach (T p in ScanPreorder(root.Left))

```

```

        yield return p;
    if (root.Right != null)
        foreach (T p in ScanPreorder(root.Right))
            yield return p;
    }
}

```

Listing 11.12 - Kod u okviru klase *Tree.cs*

U okviru klase *Tree* imamo dva javna enumeratora. Prvi ima ime *Preorder*, jer će se kretati kroz stablo sa leva na desno i sa vrha na dno. Ova metoda poziva privatnu metodu *ScanPreorder* koja koristi rekurziju da bi prolazila kroz stablo na sledeći način:

- *poseti stablo*
 - *poseti koren (root)*
 - *poseti levo stablo*
 - *poseti desno stablo*

Posetiti stablo znači doći do određenog koraka i pomoću naredbe *yield* vratiti vrednost.

Drugi enumerator je metoda *Where*, koja takođe poziva metodu *ScanPreorder* ali nakon toga filtrira dobijeni *Person* objekat na osnovu dobijenog delegata. Tip delegata je *Func<T,bool>*, što znači da uzima objekat tipa *T* (u ovom slučaju to je *Person*) kao parametar u svakoj iteraciji, i vraća bool vrednost.

Poslednji korak je dodavanje izmena u klasu *Program.cs*.

```

namespace Softversko2.Iterator.Tree
{
    class Program
    {
        // Iterator Pattern for a Tree
        // Shows two enumerators using links and recursion
        static void Main()
        {
            var family = new Tree<Person>(new Node<Person>
                (new Person("Tom", 1950),
                 new Node<Person>(new Person("Peter", 1976),
                                new Node<Person>
                                    (new Person("Sarah", 2000), null,
                                     new Node<Person>
                                         (new Person("James", 2002), null,
                                          null)// no more siblings James
                                    ),
                                new Node<Person>
                                    (new Person("Robert", 1978), null,
                                     new Node<Person>
                                         (new Person("Mark", 1980),
                                          new Node<Person>
                                              (new Person("Carrie", 2005), null, null),
                                          null)// no more siblings Mark
                                    ),
                                null) // no siblings Tom
                );

            Console.WriteLine("Full family");
            foreach (Person p in family.Preorder)
                Console.Write(p + " ");
            Console.WriteLine("\n");

            var selection = family.

```

```

        Where(p => p.Birth > 1980).
        OrderBy(p => p.Name);

        Console.WriteLine("Born after 1980 in alpha order");
        foreach (Person p in selection)
            Console.Write(p + " ");
        Console.WriteLine("\n");

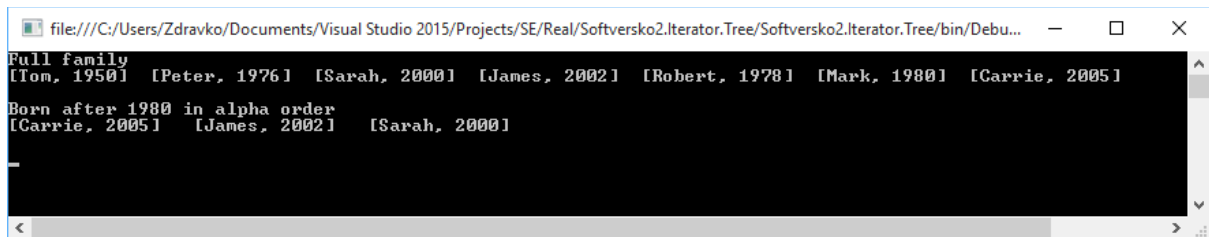
        Console.ReadKey();
    }
}

```

Listing 11.13 - Kod u okviru klase *Program.cs*

Na početku klase se vrši inicijalizacija stabla u okviru promenljive *family*.

Nakon pokretanja aplikacije dobijamo ispis koji je prikazan na slici 11.16.



Slika 11.16 - Ispis nakon pokretanja aplikacije

12. SLOJ ZA PRISTUP PODACIMA

Sloja za pristup podacima (DAL - Data Access Layer) predstavlja biblioteku koda koja omogućava pristup podacima koji se nalaze u stalnom kontejneru (bazi podataka). U slojevitom sistemu, ovom sloju se prosleđuju svi zadaci koji se odnose na čitanje ili upis u bazu.

Ideja sloja za pristup podacima je veoma jasna kada se posmatra klasični slojeviti sistem sa visokog nivoa. Iz ove perspektive, sistem poseduje sloj za prezentaciju gde se definiše korisnički interfejs, poslovni sloj gde se definišu modeli objekata i ponašanje sistema, i na kraju, sloj za pristup podacima u kojem se na bazi podataka izvršavaju upiti i ažuriranja podataka. Ovi slojevi se nalaze u jasno podeljenim oblastima, i svaki komunicira samo sa slojem koji se nalazi direktno iznad ili ispod njega. Iz toga se može zaključiti da nikakva komunikacija ne bi trebala da postoji između prezentacionog sloja i sloja za pristup podacima.

Sloj za pristup podacima poseduje precizan skup nadležnosti koje je relativno lako uočiti. Implementacija ovih nadležnosti zavisi od drugih funkcionalnih i nefunkcionalnih zahteva koji se dobijaju od naručioca. Da bi se odgovorilo ovim zahtevima, koristi se dizajn paterni u okviru faza dizajna i implementacije.

Treba napomenuti da sloj za pristup podacima živi kao poseban softverski entitet samo ako je poslovna logika organizovana korišćenjem domen model paterni. Korišćenjem domen model paterni se definiše odgovarajući objektni model. Objektni model opisuje podatke i ponašanje entiteta koji učestvuju u domenu problema. Domen model kreiran pomoću domen model paterni može biti značajno drugačiji od bilo kog relacionog modela podataka, zbog čega se pojavljuje potreba za DAL slojem koji bi implementirao trajno čuvanje podatka. Kompleksnost povezana sa odlukama gde se čuvaju određeni podaci, se obrađuje u sloju za pristup podacima.

Drugim rečima, kada se poslovna logika ne organizuje pomoću domen model paterni, metod trajnog čuvanja podataka je ugrađen u poslovnu logiku, pa se pristup podacima uspešno izvršava kroz poslovni sloj. U ovom slučaju, sloj za pristup podacima je kolekcija ugrađenih procedura i SQL iskaza.

12.1. FUNKCIONALNI ZAHTEVI

Sloj za pristup podacima predstavlja jedino mesto u sistemu gde je poznata veza sa bazom podataka i imenima tabela. Iz ovoga se može zaključiti da DAL zavisi od osobina konkretne baze podataka koja je implementirana.

Spoljašnjem posmatraču, sloj za pristup podacima bi trebao da izgleda kao crna kutija koja se dodaje u postojeći sistem i obezbeđuje čitanje i upis u određenu bazu podataka. Nezavisnost od baze podataka je čest zahtev za veliki broj poslovnih aplikacija. Kako ovo postići?

U .NET okruženju, ADO.NET predstavlja API pomoću kojeg se kreiraju konekcije, komande i transakcije nezavisne od fizičke baze podataka. U ovom slučaju, postoji samo jedan sloj za pristup podacima kreiran pomoću API-ja visokog nivoa. Iako je ovo moguć pristup, to nije nešto što se podrazumeva pod nezavisnošću baze.

Stvarna nezavisnost od baze se postiže kada se sloj za pristup podacima osmisli kao crna kutija sa pridruženim interfejsom gde se detalji o datom sloju čitaju dinamički kroz konfiguracioni fajl.

Npr. kada poslovni sloj dobije referencu na sloj za pristup podacima, on sa njim komunicira kroz metode navedene u interfejsu. Ukoliko se referenca ka stvarnoj komponenti stavi u konfiguracioni fajl, može se izvršiti promena sa Oracle baze na SQL server u dva jednostavna koraka: kreira se komponenta zavisna od DBMS koja implementira zahtevani interfejs nakon čega se prilagodi odgovarajući unos u konfiguracionom fajlu.

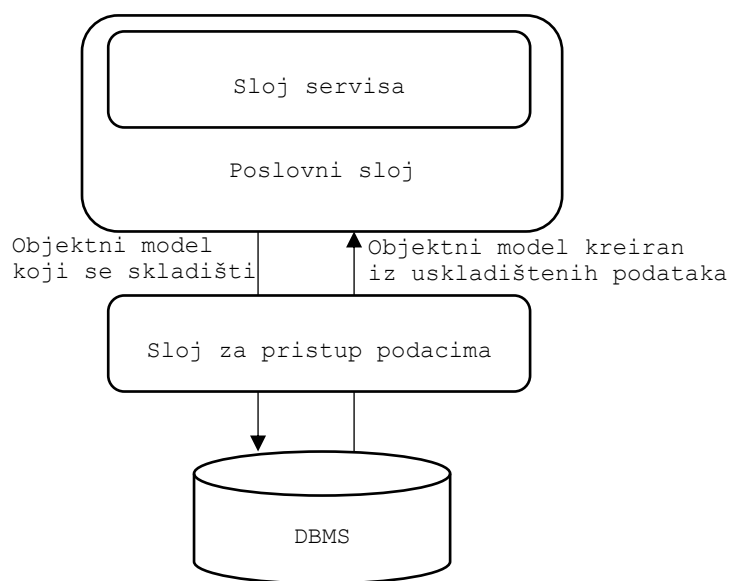
Na kraju, moguće je sistem učiniti nezavisnim od baze podataka korišćenjem alata za O/R mapiranje (O/RM). U ovom slučaju, O/RM nudi odgovarajući API koji omogućava prebacivanje na

različite baze podataka jednostavnim menjanjem konfiguracionog parametra. Primer za O/RM je Entity framework.

Sloj za pristup podacima mora biti u mogućnosti da sačuva podatke iz aplikacije bez obzira na format podataka. Podaci iz aplikacije mogu biti izraženi kroz slogove ili kroz objektni model. Ukoliko je zahtevano da se poseduje objektni model, sloj za pristup podacima mora omogućiti čuvanje modela u relacionoj organizaciji baze podataka. Jasno je da ovo kreira konstantno nepodudaranje između objektnog i relacionog pristupa. Relaciona baza skladišti torke vrednosti, dok objektni model kreira grafove objekata. Kao rezultat, mapiranje između ova dva modela postaje apsolutno neophodna i predstavlja primarni zadatak sloja za pristup podacima.

Čuvanje objektnog modela aplikacije se odnosi na sposobnost učitavanja podataka u novo kreirane instance objektnog modela i čuvanje sadržaja instanci u bazi. Trajno čuvanje se mora obezbediti bez obzira na korišćenu bazu podataka i to na način koji je nezavistan od strukture tabela.

Sledeća slika prikazuje sloj za pristup podacima koji čuva objektni model u bazu podataka i kreira instance objektnog modela od podataka iz baze.



Slika 12.1 - Sloj za pristup podacima

Poslovni sloj ne zna mnogo o primenjenoj bazi podataka. On jednostavno uspostavi komunikaciju sa slojem za pristup podacima kako bi slao ili primao podatke.

12.2. NADLEŽNOSTI SLOJA ZA PRISTUP PODACIMA

Sloj za pristup podacima ima četiri glavne nadležnosti.

- mora da obezbedi smeštanje podataka na trajno čuvanje i omogućiti CRUD (Create Read Update Delete) usluge spoljašnjim komponentama.
- ovaj sloj je nadležan da odgovori na bilo koji upit upućen prema podacima.
- on mora da omogućiti upravljanje transakcijama.
- mora na odgovarajući način odgovoriti na zahteve konkurentnosti.

Ukoliko se u aplikaciji kreira DAL, mora se dodati kod koji će implementirati četiri navedene nadležnosti. Ovo se odnosi na sve aplikacije koje poseduju određeni nivo kompleksnosti.

12.3. CRUD USLUGE

DAL je jedini sloj u sistemu koji čuva i upravlja informacijama u bazi. CRUD usluge su skup metoda koje se brinu o upisu objekata u relacione tabele, kao i o učitavanju podataka iz tabela i njihovo vraćanje u kreirane instance klasa. CRUD se izvršava nad trajnim i privremenim objektima.

Trajni objekat je instanca klase u modelu koja predstavlja određene informacije koje su izvučene iz baze podataka. Npr. ukoliko se iz baze učitava jedna narudžbina, tada se dobija trajni objekat. Ako se, umesto toga, u memoriji kreira nova narudžbina koja će kasnije biti ubačena u bazu, tada se radi sa privremenim objektom.

12.3.1. REPOSITORY PATTERN

Kakav tip koda se piše da bi implementirao CRUD?

Za svaki tip u objektnom modelu se kreira specijalna mapirajuća klasa koja implementira interfejs. Interfejs prikazuje sve operacije baze koji se mogu izvršiti nad tipom. Tu se obično koristi Repository patern. Repository se ponaša kao kolekcija u memoriji koja u potpunosti izoluje poslovne entitete od infrastrukture podataka što je vema pogodno za Domen model patern poslovne logike koji koristi POCO (Plain Old CLR Objects) objekte. Kada se koristi u projektima koji podržavaju DDD (domain-driven design) metodologiju, Repository obično postoji za svaki agregatni koren koj se nalazi u okviru domena. Primer Repository interfejsa je dat u nastavku:

```
public interface IRepository<T>
{
    IEnumerable<T> FindAll();
    IEnumerable<T> FindAll(int index, int count);

    IEnumerable<T> FindBy(Query query);
    IEnumerable<T> FindBy(Query query, int index, int count);

    T FindBy(Guid Id);

    void Add(T entity);
    void Save(T entity);
    void Remove(T entity);
}
```

Listing 12.1 - Kod za repository interfejs sa Query paternom

Kao što se može videti, interfejs pruža standardne metode za čuvanje poslovnih entiteta, ali je dobavljanje entiteta urađeno na malo drugačiji način. Objekat *Query* vrši upit nad Repository-jem na način nezavistan od podataka, čime razdvaja poslovne module od implementacije skladišta podataka i šeme podataka.

Uključivanjem prednosti koje donosi LINQ (Language Integrated Query) i model odloženog izvršavanja, Repository može da sadrži *IQueryable FindAll* metod koji omogućava poslovnom sloju da vrši direktno upite nad Repository-jem:

```
public interface IRepository<T>
{
    IQueryable<T> FindAll();

    T FindBy(Guid Id);

    void Add(T entity);
    void Save(T entity);
    void Remove(T entity);
}
```

Listing 12.2 - Kod za Repository interfejs

12.4. UPITI

U većini slučajeva, korisnik DAL usluga ima potrebu za ad hok upitima nad podacima. Međutim, nisu svi upiti isti, niti zahtevaju isti tretman. U nekim situacijama se može uočiti nekoliko uobičajenih upita koji se pozivaju sa više mesta iz poslovnog sloja ili iz sloja servisa. Ukoliko su i komanda i rezultat isti, tada se vrši hard-kodiranje upita u određenu metodu koja se po potrebi poziva. Npr. posmatramo upit kao što je *GetOrdersByCountry*. Šta se od njega očekuje? To, npr., može biti metod koji izvršava sledeću komandu:

```
SELECT * FROM orders WHERE countryID=@countryID
```

U datom slučaju se uvek dobija kolekcija *Order* objekata bez bilo kakve informacije o stavkama narudžbine i proizvodima.

U nekoj drugoj situaciji, isti upit *Get-Orders-By-Country* može dati drugačije podatke, npr. kolekciju narudžbina zajedno sa podacima o stavkama narudžbine. Šta uraditi u takvoj situaciji? Da li omogućiti dve neznatno različite *GetOrdersByCountry* metode u sloju za pristup podacima? Šta se dešava ako korisnik može da zatraži od sistema da mu da podatke o narudžbinama, stavkama u narudžbini i proizvodima po zemljama? Da li tada omogućiti još jednu sličnu metodu u sloju za pristup podacima?

Bolji pristup je definisanje posebne klase u kojoj će biti definisani svi hard-kodirani upiti koji se koriste kroz poslovni sloj. Pored toga, definiše se i objekat sa upitima opšte namene kojeg je moguće programski konfigurisati i koji, kada se izvrši, kreira ad hok SQL kod.

Glavna svrha takvog objekta sa upitima jeste da prikuplja ulazne podatke, odnosno kriterijume, kroz skup osobina i metoda i da generiše dinamički SQL za bazu podataka. Dati objekat sa upitima je idealan alat za kreiranje dinamičkih upita izvan sadržaja UI formi.

Kreiranje objekata sa upitima ne predstavlja zamenu za SQL; umesto toga on predstavlja način koji je više objektno orijentisan i fleksibilan a koji omogućava dinamičko kreiranje SQL koda za bazu. On predstavlja instancu Query Object paterna. Prema ovom paternu, rezultujući objekat sa upitima se kreira kao klasa koja sadrži skup kriterijuma. Skup kriterijuma predstavlja jednostavnu klasu koja sadrži ime osobine, vrednost i logički operator koji stavlja ime osobine i vrednost u odgovarajuću relaciju.

12.4.1. QUERY OBJECT PATTERN

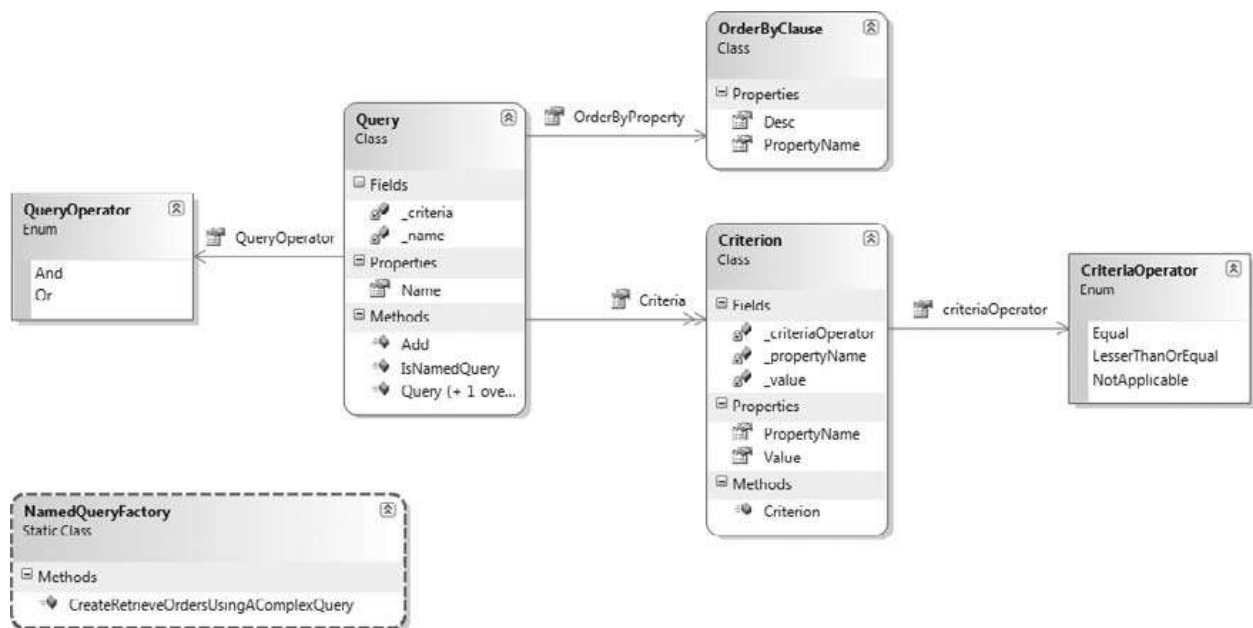
U okviru CRUD usluga kreiran je primer u kom interfejs za Repository definiše metodu koja uzima Query Object. Query Object predstavlja upit napisan pomoću jezika domena i zapravo je implementacija Query Object paterna. Query Object patern je objekat koji predstavlja upit nad bazom podataka. Glavna prednost Query Object paterna je u tome da u potpunosti abstrahuje jezik upita korišćene baze i time čini da se briga oko smeštanja i dobavljanja podataka ukloni iz sloja poslovne logike. U određenoj tački u aplikaciji se, međutim, mora kreirati jezik upita koji koristi baza podataka. Ovo se postiže korišćenjem *QueryTranslator* objekta koji uzima Query objekte i pretvara ih u jezik baze.

Da bi kreirali implementaciju za Query Object patern, kreiraćemo rešenje pod imenom *Softversko2.DB.Query* i u njega dodati sledeće class library projekte:

- *Softversko2.DB.Query.Infrastructure*
- *Softversko2.DB.Query.Model*
- *Softversko2.DB.Query.Repository*

U okviru projekta *Model* ćemo dodati referencu na projekat *Infrastructure*. U okviru projekta *Repository* ćemo dodati reference na projekte *Model* i *Infrastructure*.

Klasni dijagram za primer koji će biti prikazan je dat na slici 12.2.



Slika 12.2 – Klasni dijagram aplikacije

U projekat *Infrastructure* dodajemo folder pod imenom *Query* i dodajemo enumeraciju *CriteriaOperator* sa sledećim kodom:

```

namespace Sofrversko2.DB.QueryObj.Infrastructure.Query
{
    public enum CriteriaOperator
    {
        Equal,
        LesserThanOrEqualTo,
        NotApplicable
    }
}

```

Listing 12.3 - Kod u okviru enumeracije *CriteriaOperator.cs*

U ovom primeru ćemo koristiti samo tri kriterijum operatore koja su data u prethodnom listingu. Za punu implementaciju, treba dodati i preostale operatore.

Sledeći korak je dodavanje klase koja će predstavljati kriterijum pod imenom *Criterion*. Ova klasa predstavlja deo filtera koji formira upit, navodeći properti entiteta, vrednost sa kojom treba porediti, i način na koji treba izvršiti poređenje. Ovu klasu takođe dodajemo u folder *Query* projekta *Infrastructure*. Kod za ovu klasu je:

```

namespace Sofrversko2.DB.QueryObj.Infrastructure.Query
{
    public class Criterion
    {
        private string _propertyName;
        private object _value;
        private CriteriaOperator _criteriaOperator;

        public Criterion(string propertyName, object value,
            CriteriaOperator criteriaOperator)
        {
            _propertyName = propertyName;
            _value = value;
            _criteriaOperator = criteriaOperator;
        }

        public string PropertyName
        {

```

```

        get { return _propertyName; }
    }

    public object Value
    {
        get { return _value; }
    }

    public CriteriaOperator criteriaOperator
    {
        get { return _criteriaOperator; }
    }
}

```

Listing 12.4 - Kod u okviru klase *Criterion.cs*

Sledeća klasa koju dodajemo će predstavljati redosled kojim će upit vraćati podatke. Ime klase je *OrderByClause* sa sledećim kodom:

```

namespace Sofrversko2.DB.QueryObj.Infrastructure.Query
{
    public class OrderByClause
    {
        public string PropertyName { get; set; }
        public bool Desc { get; set; }
    }
}

```

Listing 12.5 - Kod u okviru klas *OrderByClause.cs*

Dodajemo još jednu enumeraciju da bi odredili kako će se *Criterion* objekti proveravati jedan u odnosu na drugi. Ime enumeracije je *QueryOperator* sa sledećim kodom:

```

namespace Sofrversko2.DB.QueryObj.Infrastructure.Query
{
    public enum QueryOperator
    {
        And,
        Or
    }
}

```

Listing 12.6 - Kod za enumeraciju *QueryOperator.cs*

Ponekad je teško kreirati kompleksne upite. U ovim slučajevima, mogu se koristiti imenovani upiti koji upućuju na pogled ili na ugrađenu proceduru u bazi podataka. Ovi imenovani upiti su dodati kao enumeracija *QueryName* sa sledećim kodom:

```

namespace Sofrversko2.DB.QueryObj.Infrastructure.Query
{
    public enum QueryName
    {
        Dynamic = 0,
        RetrieveOrdersUsingAComplexQuery = 1
    }
}

```

Listing 12.7 - Kod u okviru enumeracije *QueryName.cs*

U listing je uključena vrednost *Dynamic* koja se koristi ukoliko upit nije imenovan već se kreira u okviru poslovnog sloja.

Klasa koja sklapa Query Object patern je klasa *Query* sa sledećim kodom:

```

namespace Sofrversko2.DB.QueryObj.Infrastructure.Query
{
    public class Query

```

```

{
    private QueryName _name;
    private IList<Criterion> _criteria;

    public Query()
        : this(QueryName.Dynamic, new List<Criterion>()) { }

    public Query(QueryName name, IList<Criterion> criteria)
    {
        _name = name;
        _criteria = criteria;
    }

    public QueryName Name
    {
        get { return _name; }
    }

    public bool IsNamedQuery()
    {
        return Name != QueryName.Dynamic;
    }

    public IEnumerable<Criterion> Criteria
    {
        get { return _criteria; }
    }

    public void Add(Criterion criterion)
    {
        if (!IsNamedQuery())
            _criteria.Add(criterion);
        else
            throw new ApplicationException(
                "You cannot add additional criteria to named queries");
    }

    public QueryOperator QueryOperator { get; set; }
    public OrderByClause OrderByProperty { get; set; }
}
}

```

Listing 12.8 - Kod u okviru klase *Query.cs*

Klasa sadrži kolekciju *Criterion* objekata, *OrderBy* vrednost i *Oprator* vrednost. Ona takođe sadrži metodu *IsNamedQuery* koja govori da li je upit dinamički generisan ili se odnosi na ranije kreirani upid sa Repository-ja.

Poslednja klasa koju treba kreirati u okviru foldera *Query* u projektu *Infrastructure* je *NamedQueryFactory* sa sledećim kodom:

```

namespace Sofrversko2.DB.QueryObj.Infrastructure.Query
{
    public static class NamedQueryFactory
    {
        public static Query CreateRetrieveOrdersUsingAComplexQuery(Guid CustomerId)
        {
            IList<Criterion> criteria = new List<Criterion>();
            Query query =
                new Query(QueryName.RetrieveOrdersUsingAComplexQuery, criteria);
            criteria.Add(new Criterion("CustomerId", CustomerId,
                CriteriaOperator.NotApplicable));
            return query;
        }
    }
}

```

```
}
}
```

Listing 12.9 - Kod u okviru klase *NamedQueryFactory.cs*

Klasa kreira Query Object za imenovani upit. *QueryTranslator* može ispitati Query Object da bi utvrdio da li je to imenovani upit koji koristi *Criteria*s kao vrednosti za upit ugrađen u bazi. Ovim je završena implementacija Query Object paterna.

Sledeći korak je kreiranje domen modela kako bi se demonstrirala upotreba Query Object implementacije. U projekat *Model* dodajemo klasu *Order*:

```
namespace Softversko2.DB.QueryObj.Model
{
    public class Order
    {
        public Guid Id { get; set; }
        public bool HasShipped { get; set; }
        public DateTime OrderDate { get; set; }
        public Guid CustomerId { get; set; }
    }
}
```

Listing 12.10 - Kod u okviru klase *Order.cs*

Treba dodati i interfejs za dobavljanje narudžbina iz Repository-ja u okviru *Model* projekta. Pošto nas zanimaju samo funkcije Query Object paterna, treba uključiti samo jednu metodu zaduženu za dobavljanje *Order* entiteta iz Repository-ja korišćenjem Query Object paterna. Dodajemo interfejs *IOrderRepository* sa sledećim kodom:

```
namespace Softversko2.DB.QueryObj.Model
{
    public interface IOrderRepository
    {
        IEnumerable<Order> FindBy(Query query);
    }
}
```

Listing 12.11 - Kod u okviru interfejsa *IOrderRepository.cs*

Poslednja klasa koju dodajemo u projekat *Model* je domen servis klasa koja će koristiti Query Object implementaciju da vrši upite nad Repository-jem. Dodajemo klasu *OrderService* sa sledećim kodom:

```
namespace Softversko2.DB.QueryObj.Model
{
    public class OrderService
    {
        private IOrderRepository _orderRepository;

        public OrderService(IOrderRepository orderRepository)
        {
            _orderRepository = orderRepository;
        }

        public IEnumerable<Order> FindAllCustomersOrdersBy(Guid customerId)
        {
            IEnumerable<Order> customerOrders = new List<Order>();
            Query query = new Query();
            query.Add(new Criterion(
                "CustomerId", customerId, CriteriaOperator.Equal));
            query.OrderByProperty = new OrderByClause
            {
                PropertyName = "CustomerId",
                Desc = true
            };
        }
    }
}
```

```

        customerOrders = _orderRepository.FindBy(query);
        return customerOrders;
    }

    public IEnumerable<Order> FindAllCustomersOrdersWithInOrderDateBy(
        Guid customerId, DateTime orderDate)
    {
        IEnumerable<Order> customerOrders = new List<Order>();
        Query query = new Query();
        query.Add(new Criterion(
            "CustomerId", customerId, CriteriaOperator.Equal));
        query.QueryOperator = QueryOperator.And;
        query.Add(new Criterion(
            "OrderDate", orderDate, CriteriaOperator.LessThanOrEqual));
        query.OrderByProperty = new OrderByClause
        {
            PropertyName = "OrderDate",
            Desc = true
        };
        customerOrders = _orderRepository.FindBy(query);
        return customerOrders;
    }

    public IEnumerable<Order> FindAllCustomersOrdersUsingAComplexQueryWith(
        Guid customerId)
    {
        IEnumerable<Order> customerOrders = new List<Order>();
        Query query = NamedQueryFactory.
            CreateRetrieveOrdersUsingAComplexQuery(customerId);
        customerOrders = _orderRepository.FindBy(query);
        return customerOrders;
    }
}

```

Listing 12.12 - Kod u okviru klase *OrderService.cs*

Klasa *OrderService* sadrži tri metode koje kreiraju upite koji se zatim prosleđuju Repository-ju. *FindAllCustomersOrdersBy* i *FindAllCustomersOrdersWithInOrderDateBy* kreiraju dinamičke upite dodavanjem *Criteria*s i *OrderByClause*. Metoda *FindAllCustomersOrdersUsingComplexQueryWith* predstavlja imenovani upit koji koristi *NameQueryFactory* da kreira Query Object koji će biti prosleđen Repository-ju.

Nakon kreiranja domen modela i servisa, možemo implementirati *IOrderRepository* i kreirati *QueryTranslator* kako bi konvertovali Query Object u jezik koji baza može da razume.

U projekat *Repository* dodajemo klasu *OrderQueryTranslator*. Ova klasa će sadržati proširive metode koje daju Query Object-u mogućnost da se pretvori u SQL komandu koja se može izvršiti nad bazom podataka.

```

{
    public static class OrderQueryTranslator
    {
        private static string baseSelectQuery = "SELECT * FROM Orders ";

        public static void TranslateInto(this Query query, SqlCommand command)
        {
            if (query.IsNamedQuery())
            {
                command.CommandType = CommandType.StoredProcedure;
                command.CommandText = query.Name.ToString();
                foreach (Criterion criterion in query.Criteria)
                {

```

```

        command.Parameters.Add(
            new SqlParameter("@ " + criterion.PropertyName,
                criterion.Value));
    }
}
else
{
    StringBuilder sqlQuery = new StringBuilder();
    sqlQuery.Append(baseSelectQuery);
    bool _isNotfirstFilterClause = false;
    if (query.Criteria.Count() > 0)
        sqlQuery.Append("WHERE ");
    foreach (Criterion criterion in query.Criteria)
    {
        if (_isNotfirstFilterClause)
            sqlQuery.Append(GetQueryOperator(query));
        sqlQuery.Append(AddFilterClauseFrom(criterion));
        command.Parameters.Add(
            new SqlParameter("@ " + criterion.PropertyName,
                criterion.Value));
        _isNotfirstFilterClause = true;
    }
    sqlQuery.Append(GenerateOrderByClauseFrom(query.OrderByProperty));
    command.CommandType = CommandType.Text;
    command.CommandText = sqlQuery.ToString();
}

private static string GenerateOrderByClauseFrom(
    OrderByClause orderByClause)
{
    return String.Format("ORDER BY {0} {1}",
        FindTableColumnFor(orderByClause.PropertyName),
        orderByClause.Desc ? "DESC" : "ASC");
}

private static string GetQueryOperator(Query query)
{
    if (query.QueryOperator == QueryOperator.And)
        return "AND ";
    else
        return "OR ";
}

private static string AddFilterClauseFrom(Criterion criterion)
{
    return string.Format("{0} {1} @{2} ",
        FindTableColumnFor(criterion.PropertyName),
        FindSQLOperatorFor(criterion.criteriaOperator),
        criterion.PropertyName);
}

private static string FindSQLOperatorFor(
    CriteriaOperator criteriaOperator)
{
    switch (criteriaOperator)
    {
        case CriteriaOperator.Equal:
            return "=";
        case CriteriaOperator.LessThanOrEqual:
            return "<=";
        default:

```

```

        throw new ApplicationException("No operator defined.");
    }
}

private static string FindTableColumnFor(string propertyName)
{
    switch (propertyName)
    {
        case "CustomerId":
            return "CustomerId";
        case "OrderDate":
            return "OrderDate";
        default:
            throw new ApplicationException(
                "No column defined for this property.");
    }
}
}
}
}

```

Listing 12.13 - Kod u okviru klase *OrderQueryTranslator.cs*

Metod *TranslateInto* uzima ADO.NET komandu i popunjava je sa upitom za bazu podataka. Prva stvar koju *TranslateInto* radi jeste da proveri da li je Query Object imenovan upit. Ukoliko jeste, komanda je postavljena da čita imenovanu proceduru, čije je ime zapravo ime enumeracije upita. *Criteria*s i *Query* obezbeđuju sve parametre koje ugrađena procedura očekuje.

Ukoliko je *Query* dinamički kreiran, metoda iterira kroz svaki od *Criteria*s-a i kreira SQL iskaz, koristeći metode koje konvertuju imena svojstava *Order* entiteta u imena kolona *Order* tabele.

Druga i poslednja klasa koja nam je potrebna u *Repository* projektu je implementacija *IOrderRepository* interfejsa koji je definisan u okviru *Model* projekta. Dodajemo novu klasu *OrderRepository* u *Repository* projekat sa sledećim kodom:

```

namespace Softversko2.DB.QueryObj.Repository
{
    public class OrderRepository : IOrderRepository
    {
        private string _connectionString;

        public OrderRepository(string connectionString)
        {
            _connectionString = connectionString;
        }

        public IEnumerable<Order> FindBy(Query query)
        {
            IList<Order> orders = new List<Order>();
            using (SqlConnection connection =
                new SqlConnection(_connectionString))
            {
                SqlCommand command = connection.CreateCommand();
                query.TranslateInto(command);
                connection.Open();
                using (SqlDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        orders.Add(new Order
                        {
                            CustomerId = new Guid(reader["CustomerId"].ToString()),
                            OrderDate = DateTime.Parse(
                                reader["OrderDate"].ToString()),
                            Id = new Guid(reader["Id"].ToString())
                        }
                    }
                }
            }
        }
    }
}

```

```
}  
}  
}  
    }  
        }  
            return orders;  
        }  
    }  
};
```

Listing 12.14 - Kod u okviru klase *OrderRepository.cs*

OrderRepository poziva *TranslateInto* extension metod nad Query Object-om kako bi popunio ADO.NET komandni objekat. Kada je komandni objekat popunjen sa SQL iskazom, komanda je izvršena, a kolekcija narudžbina je generisana i vraćena pozivaocu.

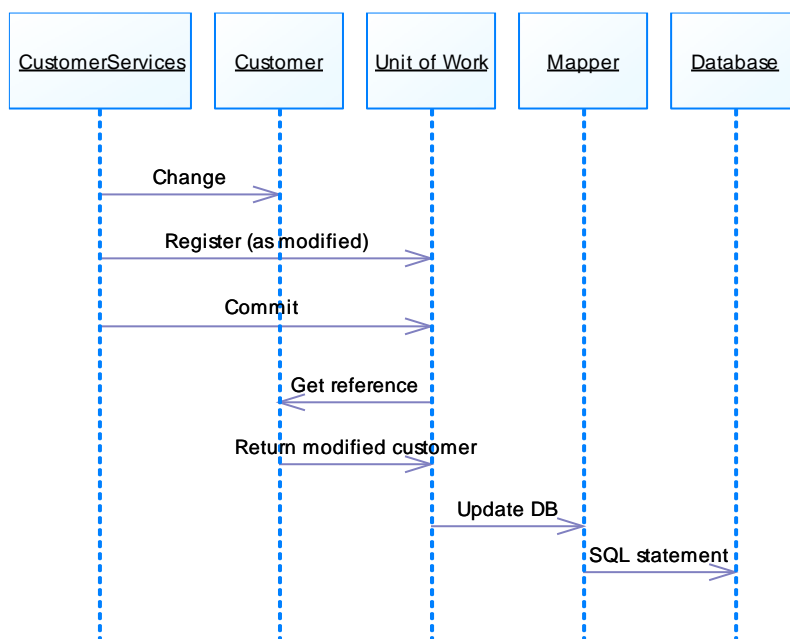
12.5. UPRAVLJANJE TRANSAKCIJAMA

U sloju za pristup bazi poslovne aplikacije, nije poželjno obraćati se bazi za svaku promenu u podacima aplikacije i odgovarajućem domen ili objektnom modelu. Očigledno je da bi takav pristup doneo veoma veliki saobraćaj ka bazi podataka i veliki šum oko svakog poziva baze (kao što su otvaranje konekcije, priprema paketa, zatvaranje konekcije, ...). Sa druge strane, smanjivanje saobraćaja ka bazi predstavlja veoma važno pravilo u arhitekturi i kreiranju aplikacije, kao i u administriranju baze.

Dobro kreirani DAL bi trebao da omogući model koji će pratiti značajne promene u podacima aplikacije koji se dešavaju tokom njenog izvršavanja. U takvom režimu rada, date promene bi se mogle preneti u bazu u jednom koraku (ne moraju se odmah učiniti promene već se to može uraditi i kasnije). Jedinica rada (Unit of Work) predstavlja logičku transakciju koja grupiše veći broj poziva ka bazi.

Uvođenjem notacije jedinice rada u sloju za pristup podacima, u suštini se kreira klasa koja odražava listu objekata domena nad kojima su se desile promene. Ova klasa takođe izražava i transakcionu semantiku (kao što je begin, commit, roll back) koja čini trajne promene nad podacima u bazi. Jedinica rada završava svoje postojanje pozivanjem objekta koji mapira podatke kako bi se izvršile CRUD aktivnosti u bazi. Ukoliko se pozove klasa jedinice rada kroz transakcioni interfejs, pristup bazi podataka se dešava transakciono.

Sledeće slika prikazuje ulogu klase jedinice rada i njenu interakciju sa entitetima u domenu ili objektnom modelu koji koriste usluge DAL-a i bazom podataka.



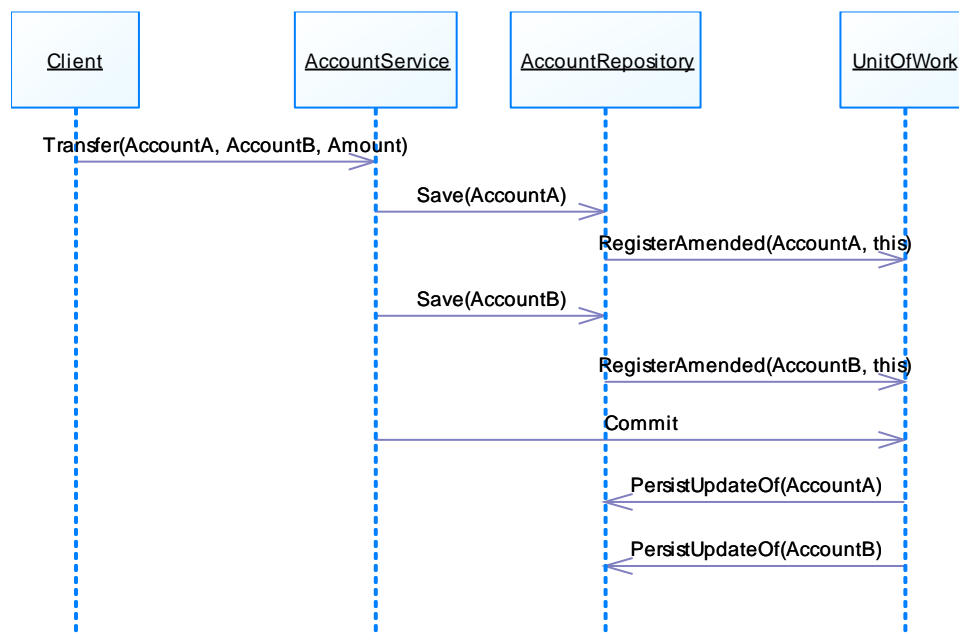
Slika 12.3 - Opšta šema jedinice rada

U suštini, kada DAL korisnik želi da ažurira klijenta, klijent se registruje pomoću jedinice rada kao izmenjeni objekat. Kasnije, kada korisnik odluči da je poslovna transakcija završena, on nalaže jedinici rada da potvrdi načinjene izmene. Nakon toga, klasa jedinice rada ide kroz listu izmenjenih objekata i izvršava zakazane akcije (insert, delete, update) nad bazom podataka. Klasa jedinice rada se odnosi na Unit of Work pattern (UoW).

12.5.1. UNIT OF WORK PATTERN

Unit of Work pattern je dizajniran da vrši održavanje za listu poslovnih objekata koji su promenjeni poslovnom transakcijom, bez obzira da li je to dodavanje, uklanjanje ili ažuriranje. Unit of Work zatim koordinira skladištenje izmena i rešavanje svih naznačenih problema kada je u pitanju konkurentnost. Prednost uvođenja ovog patterna se ogleda u tome da uvek imamo ispravne podatke.

Kako bi se demonstrirao Unit of Work pattern, koristiće se jednostavan bankarski domen koji će modelovati transfer sredstava između dva računa. Slika 12.XX prikazuje interakciju između sloja servisa i repository sloja koji koristi Unit of Work pattern kako bi osigurao da se transfer potvrđuje kao jedan atomski Unit of Work.



Slika 12.4 – Sekvencijalni dijagram aplikacije

Kreiramo novi projekat *Softversko2.DB.UoW* i u njega dodajemo sledeće class library projekte:

- *Softversko2.DB.UoW.Infrastructure*,
- *Softversko2.DB.UoW.Model*,
- *Softversko2.DB.UoW.Repository*.

U *Softversko2.DB.UoW.Model* dodajemo referencu na *Softversko2.DB.UoW.Infrastructure*. U *Softversko2.DB.UoW.Repository* treba dodati reference na projekte *Softversko2.Db.Uow.Infrastructure* i *Softversko2.DB.UoW.Model*.

Prvo ćemo dodavati kod u okviru *Infrastructure* projekta kako bi podržali Unit of Work pattern. U ovaj projekat dodajemo interfejs *IAggregateRoot* sa sledećim kodom:

```

namespace Softversko2.DB.UoW.Infrastructure
{
    public interface IAggregateRoot
    {
    }
}
  
```

```
}
```

Listing 12.15 - Kod u okviru interfejsa *IAggregateRoot.cs*

Interfejs *IAggregateRoot* predstavlja patern koji se zove marker interfejs patern. On govori da će biti skladišteni samo oni poslovni objekti koji implementiraju *IAggregateRoot* interfejs. Unit of Work implementacija će koristiti ovaj interfejs kako bi referencirala bilo koji poslovni entitet koji učestvuje u atomskoj transakciji.

U projekat *Infrastructure* dodajemo još jedan interfejs *IUnitOfWorkRepository* sa sledećim kodom:

```
namespace Softversko2.DB.UoW.Infrastructure
{
    public interface IUnitOfWorkRepository
    {
        void PersistCreationOf(IAggregateRoot entity);
        void PersistUpdateOf(IAggregateRoot entity);
        void PersistDeletionOf(IAggregateRoot entity);
    }
}
```

Listing 12.16 - Kod u okviru interfejsa *IUnitOfWorkRepository.cs*

IUnitOfWorkRepository je drugi interfejs koji će morati da implementiraju svi repository ukoliko nameravaju da budu korišćeni u okviru Unit of Work paterna.

Na kraju dodajemo treći interfejs u *Infrastructure* projekat pod imenom *IUnitOfWork*:

```
namespace Softversko2.DB.UoW.Infrastructure
{
    public interface IUnitOfWork
    {
        void RegisterAmended(IAggregateRoot entity,
            IUnitOfWorkRepository unitOfWorkRepository);
        void RegisterNew(IAggregateRoot entity,
            IUnitOfWorkRepository unitOfWorkRepository);
        void RegisterRemoved(IAggregateRoot entity,
            IUnitOfWorkRepository unitOfWorkRepository);
        void Commit();
    }
}
```

Listing 12.17 - Kod u okviru interfejsa *IUnitOfWork.cs*

Interfejs *IUnitOfWork* zahteva *IUnitOfWorkRepository* kada registruje izmenu/dodavanje /brisanje tako da prilikom komitovanja, Unit Of Work može da delegira rad stvarne metode za skladištenje na odgovarajuću konkretnu implementaciju.

U projekat *Infrastructure* zatim dodajemo klasu *UnitOfWork* sa sledećim kodom:

```
namespace Softversko2.DB.UoW.Infrastructure
{
    public class UnitOfWork : IUnitOfWork
    {
        private Dictionary<IAggregateRoot, IUnitOfWorkRepository> addedEntities;
        private Dictionary<IAggregateRoot, IUnitOfWorkRepository> changedEntities;
        private Dictionary<IAggregateRoot, IUnitOfWorkRepository> deletedEntities;

        public UnitOfWork()
        {
            addedEntities =
                new Dictionary<IAggregateRoot, IUnitOfWorkRepository>();
            changedEntities =
                new Dictionary<IAggregateRoot, IUnitOfWorkRepository>();
            deletedEntities =
                new Dictionary<IAggregateRoot, IUnitOfWorkRepository>();
        }
    }
}
```

```

    }

    public void RegisterAmended(IAggregateRoot entity,
        IUnitOfWorkRepository unitOfWorkRepository)
    {
        if (!changedEntities.ContainsKey(entity))
        {
            changedEntities.Add(entity, unitOfWorkRepository);
        }
    }

    public void RegisterNew(IAggregateRoot entity,
        IUnitOfWorkRepository unitOfWorkRepository)
    {
        if (!addedEntities.ContainsKey(entity))
        {
            addedEntities.Add(entity, unitOfWorkRepository);
        };
    }

    public void RegisterRemoved(IAggregateRoot entity,
        IUnitOfWorkRepository unitOfWorkRepository)
    {
        if (!deletedEntities.ContainsKey(entity))
        {
            deletedEntities.Add(entity, unitOfWorkRepository);
        }
    }

    public void Commit()
    {
        using (TransactionScope scope = new TransactionScope())
        {
            foreach (IAggregateRoot entity in this.addedEntities.Keys)
            {
                this.addedEntities[entity].PersistCreationOf(entity);
            }
            foreach (IAggregateRoot entity in this.changedEntities.Keys)
            {
                this.changedEntities[entity].PersistUpdateOf(entity);
            }
            foreach (IAggregateRoot entity in this.deletedEntities.Keys)
            {
                this.deletedEntities[entity].PersistDeletionOf(entity);
            }
            scope.Complete();
        }
    }
}

```

Listing 12.18 - Kod u okviru klase *UnitOfWork.cs*

Nakon dodavanja koda, neophodno je dodati referencu na *System.Transactions* kako bi se mogla koristiti *TransactionScope* klasa koja osigurava da će se skladištenje komitovati u okviru atomske transakcije. Klasa *UnitOfWork* koristi tri dictionary kolekcije kako bi pratila promene koje treba da se izvrše nad poslovnim entitetima. Prvi dictionary sadrži entitete koji treba da budu dodati u skladište podataka. Drugi dictionary prati entitete koji treba da budu ažurirani, dok se treći odnosi na entitete koji treba da budu uklonjeni. Odgovarajući *IUnitOfWorkRepository* se skladišti naspram ključa u okviru dictionary-ja i koristi se u okviru *Commit* metode kako bi pozvao Repository koji sadrži kod za rad sa stvarnim skladištem. Metoda *Commit* vrši iteraciju kroz sve elemente dictionary-ja i poziva odgovarajuću *IUnitOfWorkRepository* metodu prosleđujući referencu na dati entitet. Sve aktivnosti u

okviru *Commit* metode su upakovane u okviru *TransactionScope*-a korišćenjem bloka. Ovo osigurava da ni jedna aktivnost neće biti izvršena dok se ne pozove metoda *Complete* iz *TransactionScope*-a. Ukoliko se desi izuzetak dok se izvršava neka aktivnost u okviru *IUnitOfWorkRepository*-ja, sve se poništava (roll back), a skladište ostaje u originalnom stanju.

Kako bi demonstrirali Unit of Work patern u praksi, kreiraćemo domen bankarskih računa koji omogućavaju prenos sredstava između dva računa. U projekat *Model* dodajemo novu klasu *Account* koja poseduje jedan properti koji čuva iznos sredstava na računu:

```
namespace Softversko2.DB.UoW.Model
{
    public class Account : IAggregateRoot
    {
        public decimal balance { get; set; }
    }
}
```

Listing 12.19 - Kod u okviru klase *Account.cs*

Kako bi omogućili čuvanje *Account* objekata, dodaćemo smanjenu verziju Repository interfejsa koji sadrži metode koje su potrebne u okviru primera. U okviru modela dodajemo novi interfejs *IAccountRepository*:

```
namespace Softversko2.DB.UoW.Model
{
    public interface IAccountRepository
    {
        void Save(Account account);
        void Add(Account account);
        void Remove(Account account);
    }
}
```

Listing 12.20 - Kod u okviru interfejsa *IAccountRepository.cs*

U prethodnom interfejsu se ne nalaze metode za dobavljanje podataka jer ih u okviru aplikacije nećemo koristiti.

Da bi kompletirali model, dodaćemo klasu servisa koja će vršiti koordinaciju prenosa novčanih sredstava između dva računa. Dodajemo klasu *AccountService* sa sledećim kodom:

```
namespace Softversko2.DB.UoW.Model
{
    public class AccountService
    {
        private IAccountRepository _accountRepository;
        private IUnitOfWork _unitOfWork;

        public AccountService(IAccountRepository accountRepository,
                               IUnitOfWork unitOfWork)
        {
            _accountRepository = accountRepository;
            _unitOfWork = unitOfWork;
        }

        public void Transfer(Account from, Account to, decimal amount)
        {
            if (from.balance >= amount)
            {
                from.balance -= amount;
                to.balance += amount;
                _accountRepository.Save(from);
                _accountRepository.Save(to);
                _unitOfWork.Commit();
            }
        }
    }
}
```

```

    }
}

```

Listing 12.21 - Kod u okviru klase *AccountService.rs*

Klasa *AccountService* zahteva implementaciju *IAccountRepository* i *IUnitOfWork* interfejsa kroz svoj konstruktor. Metoda *Transfer* proverava da li je moguće izvršiti prenos sredstava, i u koliko jeste, menja iznose na računima. Nakon toga se poziva *Repository* za račune kako bi se izvršilo čuvanje podataka. Na kraju, poziva se *Commit* metoda *UnitOfWork* instance kako bi se osiguralo da je transakcija izvršena kao atomska. Da bi se videla interakcija između *Repository*-ja i *Unit of Work* paterna, potrebno je dodati implementaciju *Repository*-ja za račun.

Dodajemo novu klasu *AccountRepository* u okviru *Repository* projekta sa kodom:

```

namespace Softversko2.DB.UoW.Repository
{
    public class AccountRepository : IAccountRepository, IUnitOfWorkRepository
    {
        private IUnitOfWork _unitOfWork;

        public AccountRepository(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
        }

        public void Save(Account account)
        {
            _unitOfWork.RegisterAmended(account, this);
        }

        public void Add(Account account)
        {
            _unitOfWork.RegisterNew(account, this);
        }

        public void Remove(Account account)
        {
            _unitOfWork.RegisterRemoved(account, this);
        }

        public void PersistUpdateOf(IAggregateRoot entity)
        {
            // ADO.NET code to update the entity...
        }

        public void PersistCreationOf(IAggregateRoot entity)
        {
            // ADO.NET code to add the entity...
        }

        public void PersistDeletionOf(IAggregateRoot entity)
        {
            // ADO.NET code to delete the entity...
        }
    }
}

```

Listing 12.22 - Kod u okviru klase *AccountRepository.cs*

Klasa *AccountRepository* implementira i *IAccountRepository* i *IUnitOfWorkRepository* interfejs. Implementacija *IAccountRepository* metoda delegira aktivnost ka *Unit of Work*, pri čemu prosleđuje entitet koji treba da bude uskladišten zajedno sa referencom na *Repository*, koji implementira

IUnitOfWorkRepository. Kada se pozove *Commit* metoda, Unit of Work se obraća implementaciji *IUnitOfWorkRepository* interfejsa kako bi ispunio zahteve upućene ka skladištu.

Zbog jednostavnosti, iz prethodne klase je izbačen deo koji sadrži ADO.NET kod za čuvanje *Account* entiteta.

12.6. OBRADA KONKURENTNOSTI

Kao što je već rečeno, obraćanje bazi zbog svake promene koja se desila tokom poslovnih transakcija jednostavno nije praktično. U suštini, izbegavanje ovakvog ponašanja predstavlja prioritet u bilo kom sistemu koji poseduje bar srednji nivo kompleksnosti. Zbog toga je UoW patern veoma važan jer pruža rešenje za dati problem. Kada korisnici rade sa bazom u offline modu, javljaju se dva veoma važna problema. Jedan je upravljanje transakcijama, za koji rešenje predstavlja UoW. Drugi je konkurentnost.

U višekorisničkom okruženju, rad u offline režimu predstavlja problem sa strane integriteta. Korisnik A učita sopstvenu kopiju proizvoda 1234 i ažurira njegov opis. Pošto je operacija sastavni deo duže poslovne transakcije, dešava se da promene ne budu odmah prosleđene bazi podataka. U isto vreme korisnik B učita drugu kopiju istog proizvoda 1234 i izmeni link koji vodi ka slici proizvoda. Pretpostavimo da korisnik B izvrši momentalnu izmenu u bazi podataka. Šta se desi kada korisnik A pokuša da potvrdi izmene koje je učinio u opisu proizvoda?

Ovakva situacija se nekada može i tolerisati jer su korisnici izmenili različita polja vezana za isti proizvod. Međutim šta bi se desilo kada bi oba korisnika uneli različit opis istog proizvoda? Postoji opravdan rizik da bi se opis proizvoda koji je učinio prvi korisnik izgubio. Ovo se često označava sa "poslednji pisac pobeđuje".

Rešenje datog problema predstavlja kreiranje polise koja bi rukovala konkurentnim pristupom korisnika ka istom polju u bazi. Šansa da se desi konflikt u pristupu podacima nije ista u svim sistemima; samim tim ni cena izbijanja konflikta nije ista. Optimistički princip u rešavanju konkurentnosti se smatra najboljom polaznom tačkom za sve sisteme. Ovaj princip znači da korisnik u svojoj sesiji može slobodno pokušati da ažurira bilo koji slog u bazi podataka. Uspešnost ažuriranja se ne garantuje. U suštini, DAL ne izvršava izmene ukoliko detektuje da je slog u međuvremenu izmenjen.

Obrada konkurentnosti optimističkim metodom zahteva ad hoc mapper podataka koji kreira specijalnu proksi verziju datog entiteta (*CustomerProxy* umesto *Customer*). Kakva je razlika između *CustomerProxy* i *Customer* entiteta? Prvi sadrži originalne vrednosti sloga koje su pročitane u vreme kreiranja, dok drugi sadrži trenutnu vrednost sloga. Mapper podataka bi u WHERE iskazu SQL naredbe trebao da proveri da li je u međuvremenu došlo do promene u bazi. Ideja koja se nalazi u pozadini optimističke konkurentnosti je implementirana kroz Optimistic Offline Lock (OOL) patern.

12.6.1. DATA CONCURRENCY CONTROL

Data Concurrency Control predstavlja sistem koji daje rešenje za situaciju kada se istovremeno zatraži više modifikacija poslovnog objekta.

Kada više korisnika menjaju stanja poslovnih objekata i pokušavaju da konkurentno uskladište promene u bazi podataka, potrebno je kreirati mehanizam koji će osigurati da modifikacije jednog korisnika neće negativno uticati na stanje transakcije drugog konkurentnog korisnika.

Postoje dve forme kontrole konkurentnosti: optimistička i pesimistička. Optimistička konkurentnost pretpostavlja da ne postoje problemi kada više korisnika istovremeno vrši izmene na stanju poslovnih objekata. Ova forma je poznata i pod nazivom *poslednja promena pobeđuje*. Ovakav pristup je odgovarajući za neke sisteme. Međutim, ukoliko je neophodno da stanje poslovnih objekata bude konzistentno sa stanjem dobijenim iz beze, mora se primeniti pesimistički pristup.

Pesimistički pristup dolazi u više oblika, od zaključavanja tabela od kada se podatak dobavi, do čuvanja kopije originalnog sadržaja poslovnog objekta i njegove provere sa verzijom u skladištu pre ažuriranja podataka kako bi se osigralo da nije bilo promena u okviru datih podataka tokom trajanja

transakcije. U ovom primeru ćemo koristiti broj verzije da bi proverili da li je poslovni entitet promenjen od poslednjeg učitavanja iz baze. Nakon ažuriranja, broj verzije poslovnog entiteta će biti upoređen sa brojem verzije koji se nalazi u bazi podataka pre komitovanja promene. Ovim se osigurava da poslovni entitet nije bio modifikovan od trenutka kada je bio dobavljen.

Da bi demonstrirali pesimistički patern konkurentnosti, kreiraće se aplikacija koja čuva podatke i osigurava da je integritet podataka održan između učitavanja i ažuriranja entiteta.

Kreiramo novo rešenje *Softversko2.DB.Concurrency* i u okviru njega dodajemo sledeće C# class library projekte:

- *Softversko2.DB.Concurrency.Model*
- *Softversko2.DB.Concurrency.Repository*

U okviru projekta *Repository* treba dodati referencu na projekat *Model*.

U projekat *Model* dodajemo novu klasu *Person* sa sledećim kodom:

```
namespace Softversko2.DB.Concurrency.Model
{
    public class Person
    {
        public Guid Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int Version { get; set; }
    }
}
```

Listing 12.23 - Kod u okviru klase *Person.cs*

Property *Version* će biti podešen kada se entitet *Person* dobavi iz baze podataka.

Da bi kompletirali jednostavan domen model, dodajemo novi interfejs u *Model* pod imenom *IPersonRepository* sa sledećim kodom:

```
namespace Softversko2.DB.Concurrency.Model
{
    public interface IPersonRepository
    {
        void Add(Person person);
        void Save(Person person);
        Person FindBy(Guid Id);
    }
}
```

Listing 12.24 - Kod u okviru interfejsa *IPersonRepository.cs*

Da bi se kreirao projekat *Repository* u okviru njega treba dodati klasu *PersonRepository* koja implementira *IPersonRepository* interfejs:

```
namespace Softversko2.DB.Concurrency.Repository
{
    public class PersonRepository : IPersonRepository
    {
        private string _connectionString;
        private string _findByIdSQL =
            "SELECT * FROM People WHERE PersonId = @PersonId";
        private string _insertSQL =
            "INSERT People (FirstName, LastName, PersonId, Version) VALUES " +
            "(@FirstName, @LastName, @PersonId, @Version)";
        private string _updateSQL =
            "UPDATE People SET FirstName = " +
            "@FirstName, LastName = @LastName, Version = " +
            "@Version + 1 WHERE PersonId = @PersonId AND Version = @Version;";
    }
}
```

```

public PersonRepository(string connectionString)
{
    _connectionString = connectionString;
}

public void Add(Person person)
{
    using (SqlConnection connection = new SqlConnection(_connectionString))
    {
        SqlCommand command = connection.CreateCommand();
        command.CommandText = _insertSQL;
        command.Parameters.Add
            (new SqlParameter("@PersonId", person.Id));
        command.Parameters.Add
            (new SqlParameter("@Version", person.Version));
        command.Parameters.Add
            (new SqlParameter("@FirstName", person.FirstName));
        command.Parameters.Add
            (new SqlParameter("@LastName", person.LastName));
        connection.Open();
        command.ExecuteNonQuery();
    }
}

public void Save(Person person)
{
    int numberOfRecordsAffected = 0;
    using (SqlConnection connection = new SqlConnection(_connectionString))
    {
        SqlCommand command = connection.CreateCommand();
        command.CommandText = _updateSQL;
        command.Parameters.Add
            (new SqlParameter("@PersonId", person.Id));
        command.Parameters.Add
            (new SqlParameter("@Version", person.Version));
        command.Parameters.Add
            (new SqlParameter("@FirstName", person.FirstName));
        command.Parameters.Add
            (new SqlParameter("@LastName", person.LastName));
        connection.Open();
        numberOfRecordsAffected = command.ExecuteNonQuery();
    }
    if (numberOfRecordsAffected == 0)
        throw new ApplicationException(
            @"No changes were made to Person Id (" + person.Id + "), " +
            "this was due to another process updating the data.");
    else
        person.Version++;
}

public Person FindBy(Guid Id)
{
    Person person = default(Person);
    using (SqlConnection connection = new SqlConnection(_connectionString))
    {
        SqlCommand command = connection.CreateCommand();
        command.CommandText = _findByIdSQL;
        command.Parameters.Add(new SqlParameter("@PersonId", Id));
        connection.Open();
        using (SqlDataReader reader = command.ExecuteReader())
        {
            if (reader.Read())

```



```

        {
            person = new Person
            {
                FirstName = reader["FirstName"].ToString(),
                LastName = reader["LastName"].ToString(),
                Id = new Guid(reader["PersonId"].ToString()),
                Version = int.Parse(reader["Version"].ToString())
            };
        }
    }
    return person;
}
}
}

```

Listing 12.25 - Kod u okviru klase *PersonRepository.cs*

Metode *FindBy* i *Add* sadrže ADO.NET kod koji omogućava da se popuni jedan entitet *Person* iz baze podataka, odnosno da se doda novi *Person* entitet u bazu podataka. Metod *Save* sadrži logiku koja proverava integritet podataka *Person* entiteta. Kada je *Person* entitet sačuvan, verzija promenjenog entiteta je uključena u *where* iskaz. Ukoliko se verzija ne podudara, ne izvršava se ažuriranje, a metoda *ExecuteNonQuery* vraća da je nula objekata pogodeno promenom. U ovoj tački bi mogao biti ispaljen određeni izuzetak koji bi govorio da je *Person* entitet promenjen ili obrisan od trenutka kada je bio dobavljen i da ažuriranje nije uspelo.

12.6.2.IDENTITY MAP

Ovaj patern osigurava da se svaki objekat učitava samo jednom tako što drži učitane objekte u okviru mape i traži objekte u okviru te mape kada se referencira na njih. Kada se suočavamo sa konkurentnošću podataka, važno je imati strategiju za više korisnika koji deluju nad istim poslovnim entitetom. Ovaj patern je veoma važan i za pojedinačno korisnike nekog sistema kako bi koristili konzistentne verzije poslovnih entiteta tokom aplikacija koje se dugo izvršavaju ili su kompleksne. Identity map omogućava ove funkcionalnosti tako što čuva verzije svih poslovnih objekata koji se koriste u okviru transakcije. Ovaj patern znači da ako se npr. isti *Employee* entitet zatraži dva puta, ista instanca se vraća.

U cilju demonstracije ovog patern, kreiraće se aplikacija koja dobavlja podatke o zaposlenima (*Employee*) iz Repository-ja. Kreiraćemo novo rešenje pod nazivom *Softversko2.DB.IM*. U okviru rešenja dodajemo dva nova class library projekta:

- *Softversko2.DB.IM.Model*
- *Softversko2.DB.IM.Repository*

U okviru projekta *Repository* dodajemo referencu na projekat *Model*.

U projekat *Model* dodajemo novu klasu *Employee* sa sledećim kodom:

```

namespace Softversko2.DB.IM.Model
{
    public class Employee
    {
        public Guid Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}

```

Listing 12.26 - Kod u okviru klase *Employee.cs*

Da bi završili model, potrebno je dodati još interfejs za pristup skladištu koji sadrži jednu metodu za vraćanje objekata tipa *Employee* na osnovu njegovog id-a. Ime interfejsa je *IEmployeeRepository*:

```

namespace Softversko2.DB.IM.Model
{
    public interface IEmployeeRepository
    {
        Employee FindBy(Guid Id);
    }
}

```

Listing 12.27 - Kod u okviru interfejsa *IEmployee.cs*

Nakon kreiranja modela, krećemo sa kreiranjem *Repository* projekta. U projekat dodajemo novu klasu *IdentityMap*. Ova klasa će koristiti generičke tipove kako bi omogućila Identity Map implementaciju za jedinstvene *Employee* entitete tokom poslovnih transakcija. Kod za ovu klasu je dat u nastavku:

```

namespace Softversko2.DB.IM.Repository
{
    public class IdentityMap<T>
    {
        Hashtable entities = new Hashtable();

        public T GetById(Guid Id)
        {
            if (entities.ContainsKey(Id))
                return (T)entities[Id];
            else
                return default(T);
        }

        public void Store(T entity, Guid key)
        {
            if (!entities.Contains(key))
                entities.Add(key, entity);
        }
    }
}

```

Listing 12.28 - Kod u okviru klase *IdentityMap.cs*

IdentityMap sadrži heš tabelu kako bi čuvala poslovne entitete koji se koriste u okviru transakcija, i omogućava jednostavan interfejs za skladištenje i dobavljanje entiteta.

Klasu *IdentityMap* ćemo koristiti u okviru implementacije za *IEmployeeRepository*. Dodajemo novu klasu u *Repository* projekat pod imenom *EmployeeRepository* pri čemu će ona da implementira *IEmployeeRepository* interfejs koji se nalazi u okviru *Model* projekta:

```

{
    public class EmployeeRepository : IEmployeeRepository
    {
        private IdentityMap<Employee> _employeeMap;

        public EmployeeRepository()
        {
            _employeeMap = new IdentityMap<Employee>();
        }

        public Employee FindBy(Guid Id)
        {
            Employee employee = _employeeMap.GetById(Id);
            if (employee == null)
            {
                employee = DatastoreFindBy(Id);
                if (employee != null)
                    _employeeMap.Store(employee, employee.Id);
            }
        }
    }
}

```

```

        return employee;
    }

    private Employee DatastoreFindBy(Guid Id)
    {
        Employee employee = default(Employee);
        // Code to hydrate employee from datastore...
        return employee;
    }
}

```

Listing 12.29 - Kod u okviru klase *EmployeeRepository.cs*

Kada se pozove metoda *FindBy*, *EmployeeRepository* provo proverava *IdentityMap* kako bi odredio da li je *Employee* entitet ranije bio dobavljan. Ukoliko jeste, on se vraća pozivaocu. Ukoliko nije, vrši se upit nad skladištem podataka kako bi se dobila *Employee* instanca korišćenjem njenog identifikatora, a zatim se ona doaje u *IdentityMap*.

13. LITERATURA

- [1] Microsoft .NET Architecting Applications for the Enterprise, Dino Esposito, Andrea Saltarello, Microsoft Press, October 2008
- [2] <http://www.dofactory.com/net/singleton-design-pattern>
- [3] http://www.tutorialspoint.com/design_pattern/factory_pattern.htm
- [4] http://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm
- [5] <http://www.dofactory.com/net/decorator-design-pattern>
- [6] <http://www.codeproject.com/Articles/482196/Understanding-and-Implementing-Template-Method-Des>

14. ISPITNA PITANJA

1. Arhitektura sistema, ciljevi i principi
2. Dizajn principi i paterni, uobičajeni dizajn principi
3. S.O.L.I.D. principi, Single Responsibility princip (navesti primer koda)
4. S.O.L.I.D. principi, Open close princip (navesti primer koda)
5. S.O.L.I.D. principi, Liskov substitution princip (navesti primer koda)
6. S.O.L.I.D. principi, Interface segregation princip (navesti primer koda)
7. S.O.L.I.D. principi, Dependency inversion princip (navesti primer koda)
8. Višeslojna arhitektura
9. Prezantacioni sloj i MVC patern (navesti primer koda)
10. Sloj servisa, primer aplikacije koja koristi sloj servisa
11. Sloj servisa, Idempotent patern
12. Poslovni sloj aplikacije
13. Paterni za kreiranje poslovnog sloja i grupe dizajn paterni
14. Adapter patern, UML dijagram i demo primer
15. Composite patern, UML dijagram i demo primer
16. Decorator patern, UML dijagram i demo primer
17. Facade patern, UML dijagram i demo primer
18. Proxy patern, UML dijagram i demo primer
19. Bridge patern, UML dijagram i demo primer
20. Prototype patern, UML dijagram i demo primer
21. Factory method patern, UML dijagram i demo primer
22. Singleton patern, UML dijagram i demo primer
23. Abstract Factory patern, UML dijagram i demo primer
24. Strategy patern, UML dijagram i demo primer
25. State patern, UML dijagram i demo primer
26. Template method patern, UML dijagram i demo primer
27. Command patern, UML dijagram i demo primer
28. Iterator patern, UML dijagram i demo primer
29. Sloj za pristup podacima, funkcionalni zahtevi i nadležnost
30. CRUD usluge i Repository patern
31. Upravljanje transakcijama i Unit of Work patern
32. Obrada konkurentnosti i Data concurrency control
33. Identity map