

# Objektno orijentirana paradigma kroz C#

---

Seminarski rad iz predmeta Programske paradigme i jezici

Melita Mihaljević, Ana Vignjević



## SADRŽAJ

|  |    |
|--|----|
| 1.UVOD U OOP.....  | 5  |
| 1.1. Programske paradigme .....                                  | 5  |
| 1.2. Osnovni koncepti objektno orijentiranog programiranja.....  | 6  |
| 1.3. Razlika objektno orijentirane i proceduralne paradigme..... | 8  |
| 1.4. Zanimljivi linkovi.....                                     | 8  |
| 2.APSTRAKCIJA .....  | 9  |
| 2.1. Definicija .....  | 9  |
| 2.2. Općenito o apstrakciji.....                                 | 9  |
| 2.3. Primjeri apstrakcije.....                                   | 9  |
| 2.4. Zanimljivi linkovi.....                                     | 11 |
| 3.ENKAPSULACIJA.....   | 12 |
| 3.1. Definicija.....   | 12 |
| 3.2. Općenito o enkapsulaciji.....                               | 12 |
| 3.3. Enkapsulacija i skrivanje informacija.....                  | 12 |
| 3.4. Odnos enkapsulacije i apstrakcije.....                      | 13 |
| 3.5. Zanimljivi linkovi.....                                     | 14 |
| 4. NASLJEĐIVANJE.....  | 15 |
| 4.1 Definicija.....  | 15 |
| 4.2 Općenito o nasljeđivanju.....                                | 15 |
| 4.3 Nasljeđivanje u C#.....                                      | 18 |
| 4.3.1. Nasljeđivanje razreda.....                                | 18 |
| 4.3.1.1. Razred Object.....                                      | 20 |
| 4.3.2. Nasljeđivanje i konstruktori.....                         | 20 |
| 4.4. Zanimljivi linkovi.....                                     | 21 |
| 5.POLIMORFIZAM.....  | 22 |
| 5.1. Definicija.....   | 22 |
| 5.2. Općenito o polimorfizmu.....                                | 22 |
| 5.3. Polimorfizam u C#.....                                      | 23 |
| 5.4. Zanimljivi linkovi.....                                     | 26 |
| 6.SUČELJA.....   | 27 |
| 6.1. Definicija.....   | 27 |
| 6.2. Općenito o sučeljima.....                                   | 27 |
| 6.3. Sučelja u C#.....   | 27 |
| 6.3.1. Razlika sučelja i apstraktnih razreda.....                | 29 |
| 6.3.2. Razlika sučelja i razreda.....                            | 30 |
| 6.3.3. Nasljeđivanje i kombiniranje sučelja.....                 | 31 |
| 6.3.4. Korištenje metoda sučelja.....                            | 31 |
| 6.4. Eksplicitna implementacija sučelja.....                     | 33 |
| 6.5. Zanimljivi linkovi.....                                     | 34 |
| 7.UZORCI OBJEKTNO ORIJENTIRANOG DIZAJNA.....                     | 35 |
| 7.1. Definicija.....   | 35 |
| 7.2. Općenito o uzorcima.....                                    | 35 |
| 7.3. Što čini uzorak.....  | 35 |

|   |    |
|---|----|
| 7.3.1. Ime uzorka.....                                      | 35 |
| 7.3.2. Problem.....   | 36 |
| 7.3.3. Rješenje.....  | 36 |
| 7.3.4. Posljedice.....                                      | 36 |
| 7.4. Razlozi uporabe uzoraka.....                           | 36 |
| 7.5. Klasifikacija i dokumentacija uzoraka.....             | 37 |
| 7.5.1. Klasifikacija uzoraka.....                           | 37 |
| 7.5.1.1. Tvorbeni uzorci.....                               | 38 |
| 7.5.1.2. Strukturni uzorci.....                             | 38 |
| 7.5.1.3. Ponašajni uzorci.....                              | 39 |
| 7.5.1.4. Klasifikacija uzoraka.....                         | 39 |
| 7.5.2. Dokumentacija.....                                   | 39 |
| 7.6. Primjena uzoraka.....                                  | 40 |
| 7.6.1. Factory Method.....                                  | 40 |
| 7.6.2. Singleton.....                                       | 42 |
| 7.6.3. Strategy.....  | 43 |
| 7.7. Kako odabrati i koristiti uzorak dizajna.....          | 45 |
| 7.8. Antiuzorci oo dizajna.....                             | 46 |
| 7.9. Zanimljivi linkovi.....                                | 46 |
| 8. OOP vs OOdizajn.....                                     | 47 |
| 8.1. Proces dizajniranja objektno orijentiranog modela..... | 47 |
| 8.2. Zanimljivi linkovi.....                                | 52 |
| 9. BIBLIOGRAFIJA.....                                       | 53 |

## 1. UVOD U OOP

### 1.1 Programske paradigme

Danas su u programiranju dominantne dvije programske paradigme. Prva od njih i povijesno gledajući najstarija je **imperativna** (ili proceduralna) paradigma. Druga koja je danas vjerojatno najzastupljenija, te koja se najčešće spominje je **objektno orijentirana** programska paradigma. Uz dvije gore navedene paradigme poprilično su zastupljene i funkcijska paradigma, te danas sve popularnija konceptualno orijentirana paradigma (eng. Concept oriented paradigm).

**Imperativna** (tj. proceduralna) programska paradigma je jednostavna – program se sastoji od niza deklaracija, naredbi pridruživanja te poziva predefiniranih kao i vlastitih potprograma. Potprogrami omogućuju smisleno strukturiranje programa. Problem sa ovakvim pristupom programiranju je sve veća kompleksnost današnjih programa: što je program veći - teže ga je održavati.

Temelj **funkcijskog programiranja** te funkcijske programske paradigme leži u izrazima i evaluacijama. Ta dva pojma se mogu smatrati centralnim objektima promatranja u funkcijskom programiranju. Program se sastoji od niza izraza i evaluacija, a nakon svake evaluacije smo bliže rješenju (potrebno je obaviti još evaluacija), ili smo došli do konačnog rješenja. Prilikom evaluacije se primjenjuju razne tehnike poput rekurzije i ugnježđivanja funkcija. Ovakav programski model je pogodan za efikasan opis raznih matematičkih izraza, te algoritima. Pošto većina takvih izraza koristi liste i skupove, funkcijski jezici su najčešće orijentirani na rad sa listama. Jedan od najpoznatijih te najstarijih programskih jezika koji podržavaju funkcijsko programiranje je LISP. LISP je funkcijski jezik, ali u njemu je moguće koristiti i drukčije programske paradigme, na primjer objektnu. Neki od ostalih poznatih programskih jezika koji su funkcijski ili podržavaju funkcijsko programiranje su: ML, Erlang, Logo, Python, itd.

**Konceptualno orijentirana** paradigma generalizira objektno-orijentirano programiranje i ostale programske tehnike. Temelji se na skupu svojstava i koristi novu programsku strukturu – koncept. Koncepti su u konceptualno orijentiranom programiranju ekvivalentni objektima u objektno orijentiranom programiranju. Konceptualno orijentirana paradigma je nova programska paradigma tek u nastajanju te još ne postoji programski jezik koji podržava ovu programsku paradigmu, ali se pretpostavlja da će izgrađeni programski jezik biti nasljednik programskog jezika Java U daljem tekstu nećemo se dodatno osvrnati na ovu paradigmu jer ona nije predmet našeg proučavanja.

Kod **objektno orijentiranog** programiranja (OOP) program se sastoji od skupa objekata koji međusobno komuniciraju. **Objekt** je

**programski entitet** koji je sposoban primiti, obrađivati i slati poruke (poruke su programski najčešće realizirane običnim funkcijskim pozivima i callback funkcijama). Objektno orijentirani model je izuzetno pogodan **za opisivanje današnjih sustava koji se najčešće sastoje od jasno odvojenih, samostalnih cjelina** koje međusobno komuniciraju i razmjenjuju obrađene podatke. Jedan od takvih primjera koji se najčešće navodi je grafičko korisničko sučelje (GUI). Primjeri objektno orijentiranih jezika su **C++, C#, Eiffel, Java**, itd.

Povijesni razvoj objektno orijentiranog programiranja seže u 60-te godine prošlog stoljeća. Objekti kao entiteti programa predstavljeni su prvi put u programskom jeziku **Simula 67**, kojeg su dizajnirali Ole-Johan Dahl i Kristen Nygaard na Norveškom računarskom centru u Oslu. Programskim jezikom Simula 67 uvedeni su pojmovi objekata, razreda, podrazreda, virtualnih metoda, itd. 70-tih godina razvijen je programski jezik **Smalltalk** pod utjecajem Simula-e ali dizajniran kao potpuno dinamički sustav u kojemu je razrede moguće kreirati i modificirati dinamički za razliku od statičkog načina u Simulli 67. Popularnost objektno orijentirane paradigme porasla je naglo s razvojem grafičkih korisničkih sučelja, te razvojem programskog jezika **C++**. Dodatan rast popularnosti dogodio se u 90-tima pri razvoju računalnih igara. Primjeri igara su svima dobro poznate Doom III, Starcraft, Diablo, Warcraft III i World of Warcraft. Rast popularnosti objektno orijentirane paradigme nastavio se sve do danas.

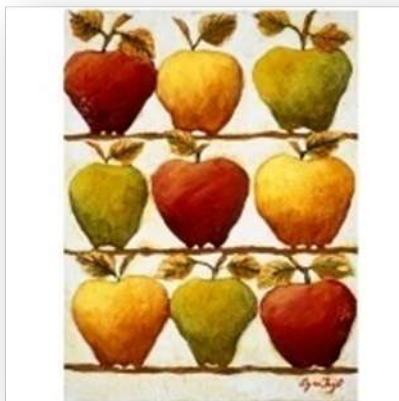
## 1.2 Osnovni koncepti objektno orijentiranog programiranja

Osnovni temelji objektno orijentirane paradigme su prije svega **objekt** i **razred**, te **metode**, **nasljeđivanje**, **enkapsulacija**, **polimorfizam**, **apstrakcija** i **sučelja**.

Konceptom objektnog programiranja uvodi se mogućnosti definiranja novih korisničkih tipova – razreda. **Razredi su temelj objektno orijentiranog programiranja.**

Ideja uvođenja objekata u programiranje došla je analiziranjem načina na koji funkcionira stvarni svijet. Promatanjem svijeta oko sebe doći ćemo do zaključka da se mnoge stvari mogu vrlo jednostavno modelirati pomoću objekata.

Sjetimo se slikara koji izrađuje svoje umjetničko djelo. On prvo gleda objekte koje će staviti na platno, zatim kako će se oni ponašati u skladu s drugim objektima. Razmislimo što je slikar koji je htio naslikati sliku s jabukama morao napraviti (osim što je morao kupiti kist, boje i platno). Prvo je razmišljao o osnovnim svojstvima jabuka (boja, oblik, težina) i na temelju osnovnih svojstava naslikao svaku od jabuka – napravio je objekte na temelju razmišljanja, stvarne objekte sa zadovoljavajućim svojstvima. Svatko tko pogleda prepoznat će da je to jabuka, točnije 9 jabuka (9 objekata). **Postoji 9 objekata razreda jabuka.** Kao što vidimo razred jabuka je samo opis onih objekata koje možemo dobiti (vidjeti u našem slučaju).



Kada smo na primjeru iz svijeta oko nas objasnili razliku između objekta i razreda definirajmo ih formalno.

**Objekt** je naziv za skup svojstava koja možemo objediniti u smislenu cjelinu.

**Razred** je skup pravila koja propisuju od čega je pojedini objekt sagrađen, te kakva su njegova svojstva.

Vrlo je važno uočiti razliku između razreda i objekta: **razred je samo opis, a objekt je stvarna, konkretna realizacija napravljena na temelju razreda.**

Objekti međusobno razmjenjuju informacije i traže jedan od drugoga usluge. Pritom okolina objekta ništa ne mora znati o njegovom unutarnjem ustrojstvu. Svaki objekt ima javno sućenje kojim se definira njegova suradnja s okolinom. Ono određuje koje informacije objekt može dati, te u kojem formatu. Također su definirane i sve usluge koje objekt može pružiti.

Interno se objekt sastoji od niza drugih objekata i interakcija među njima. Način na koji se reprezentacija objekta ostvaruje jest implementacija objekta. Ona je najčešće skrivena od okoline kako bi se osigurala konzistentnost objekta. Razred se, dakle, sastoji od opisa javnog sučelja i od implementacije.

Kada je razred jednom definiran može se pomoću njega konstruirati neograničen broj objekata (sjetimo se slike s jabukama – razred jabuka (zamišljena svojstva jabuka) i stvarnih jabuka na slici - objekti). Kada se objekt stvori potrebno mu je dati početni oblik. Postupak koji opisuje kako će se to napraviti dan je u specifikaciji razreda funkcijom koja inicijalizira objekt i koja se naziva konstruktorom. Različiti OO programski jezici imaju različite mehanizme oslobađanja memorije kada se objekti više ne koriste. U C++ tome služe destruktori, u Javi o tome vodi brigu Garbage Collector, a u C# postoji oboje [1].

U objektno orijentiranom svijetu izbjegava se korištenje globalnih varijabli, nego **svaki objekt ima svoje varijable**, koje još zovemo i **podatkovni članovi klase**. Isto tako nema više ni samostojećih funkcija, nego **funkcije pripadaju klasi**, bilo da vanjskom svijetu

nešto kazuju o stanju objekta, ili mijenjaju stanje objekta, a zovemo ih **funkcijski članovi klase ili metode**.

Osnovni koncepti objektno orijentirane paradigme su detaljno objašnjeni u sljedećim poglavljima pa ih ovdje nećemo definirati.

Da bismo bolje razumjeli koncept objektno orijentiranog programiranja usporedimo ga s proceduralnom paradigmom koja je obrađivana u nekoliko prethodnih kolegija.

Poseban naglasak je na izvedbi koncepata objektno orijentirane paradigme u programskom jeziku C#, te su svi primjeri napisani u njemu

### **1.3. Razlika objektno orijentirane i proceduralne paradigme**

Objasnimo razliku objektno orijentirane i proceduralne paradigme na konkretnom primjeru koji je obrađen u sklopu učenja proceduralne paradigme i programskog jezika C.

Sjetimo se implementacije vezane liste u programskom jeziku C gdje je bilo potrebno implementirati posebnu strukturu za čvor liste, posebne funkcije za dodavanje u listu, brisanje iz liste, sortiranje liste.

Sve je to bilo vrlo neelegantno riješeno jer je svaka dodatna funkcionalnost koju je bilo potrebno implementirati zahtijevala vrlo velike promjene u programskom kodu i programskoj logici, što najčešće ostavlja veliki trag na ostatku programa i programima koji koriste te funkcije (tj. prestajali su raditi). S druge strane objektno orijentirani pristup omogućava vrlo elegantno rješenje, jer omogućava da se podaci i metode koje rukuju podacima liste nalaze na istom mjestu – u objektu.

U nastavku su detaljno objašnjeni koncepti objektno orijentiranog programiranja, počevši s apstrakcijom.

### **1.4. Zanimljivi linkovi:**

1. Programske paradigme:  
[http://en.wikipedia.org/wiki/Programming\\_paradigms](http://en.wikipedia.org/wiki/Programming_paradigms) (1.2.2007)
2. Konceptualno orijentirana paradigma  
<http://conceptoriented.com/> (2.2.2007)
3. Osnovni koncepti konceptualno orijentirane paradigme:  
[http://conceptoriented.com/savinov/publicat/imi-report\\_07.pdf](http://conceptoriented.com/savinov/publicat/imi-report_07.pdf)  
(24.3.2007)



## **2. APSTRAKCIJA**

### **2.1. Definicija**

Apstrakcija je određivanje bitnih osobina i funkcionalnosti bez uvođenja nepotrebnih detalja ili objašnjenja. Pomoću nje modeliraju se najvažniji dijelovi nekog razreda i njegovo ponašanje.

### **2.2. Općenito o apstrakciji**

Apstrakcija se koristi kao proces ali kao i entitet. U kontekstu procesa, apstrakcija je izlučivanje bitnih svojstava nekog elementa, ili skupine elemenata dok se nebitna svojstva zanemaruju. Kao entitet, apstrakcija je konkretan prikaz nekog elementa, njegovih sastavnih dijelova i ponašanja. Cilj apstrakcije jest smanjenje kompleksnosti i povećanje učinkovitosti koda.

Ono što ostane kao rezultat izostavljanja svih nebitnih i perifernih detalja i identificiranjem ključnih ponašanja nekog objekta može se smatrati našom definicijom tog entiteta, sastavljenom od probranih osobina i ponašanja. U terminima programiranja apstrakcija je određivanje suštine nekog entiteta.

Postoji više razina apstrakcije. Količina informacija je obrnuto proporcionalna razini na kojoj se nalazimo. Na višim razinama apstrakcije naš fokus je na bitnijim i općenitijim informacijama (iz naše perspektive) kojih neće biti puno. Kako se spuštamo na niže razine otkrivamo specifičnije detalje čija brojnost se povećava. Implementacija se može smatrati najnižom razinom apstrakcije, pošto se tu radi o najvećoj količini informacija, a sve su važne (od varijabli, kontrole toka, metoda, itd.).

Apstrakcija ovisi o perspektivi korisnika, te je iterativni process. Da bi se pronašao najbolji i najtočniji opis objekta kao rješenje za zadani problem, obično je potrebno proći nekoliko koraka.

Postoji nekoliko vrsta apstrakcije, npr.: funkcionalna (engl. functional abstraction), apstrakcija podataka (engl. data abstraction), apstrakcija procesa (engl. process abstraction) i apstrakcija objekata (engl. object abstraction).

Apstrakcija neke informacije proglašava bitnijima od drugih, no ona ne definira mehanizam kojim bi se to moglo odrediti niti kako postupati sa informacijama koje su nevažne. To u cijelosti ovisi o samom developeru i kontekstu.

### **2.3. Primjeri apstrakcije**

Zamislamo da nam je zadatak napraviti aplikaciju koja će se koristiti u knjižnicama za posuđivanje knjiga. Što su sve elementi jedne knjige?

Knjiga {naslov, autori, izdavač, isbn broj, žanr, kategorija, tematika, broj stranica, vrsta uveza, vrsta papira, itd. }

Da bismo mogli modelirati objekt tipa Knjiga, potrebno je odrediti njegove bitne značajke.

Ako gledamo iz perspektive knjižničarke, ona bi kao glavne značajke knjige mogla izdvojiti naslov, autore, isbn broj i kategoriju kojoj knjiga pripada. Naime, njena perspektiva je određena njenim poslom – knjižnicu treba organizirati po kategorijama ili po abecednom redoslijedu autora ili naslova, a možda i po isbn brojevima. Model knjige bi tada mogao biti:

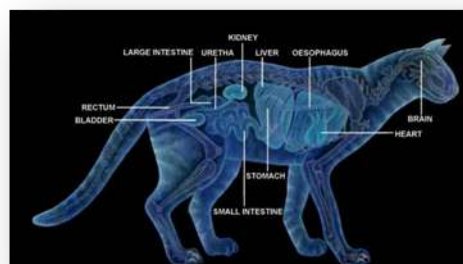
Knjiga {naslov, autori, isbn broj, kategorija}

Iz perspektive srednjoškolskog učenika, glavne značajke knjige bi bile naslov, autori i nažalost, broj stranica (čak ne nužno tim redoslijedom).

Knjiga {broj stranica, naslov, autori}

Iz perspektive grafičkog urednika čija je zadaća osmisлити naslovnu stranu knjige, glavna značajka bi vjerojatno bila vrsta uveza (pošto je to jedini dio knjige s kojim će u svom poslu imati doticaja) i tematika (na temelju koje treba osmisлити prikladan izgled omota). Očito je naša percepcija modela ključna u procesu apstrakcije.

Drugi primjer možemo uzeti iz životinjskog svijeta. Ako mačku promatra dijete, ono će vidjeti samo kućnog ljubimca za kojeg zna da može prestići, penjati se po drveću i hvatati miševе (ponašanje), te ima glatku dlaku različitih boja i dug rep (osobine). Veterinar će na mačku gledati kao na životinju, s određenim rasporedom unutrašnjih organa, koja može imati neke specifične bolesti (te informacije su bitne u njegovom poslu).



Apstrakcija nam definira što nešto jest najjednostavnije i najtočnije moguće i ne više no što je nužno potrebno.

Također, apstrakcija nam omogućava da problem adresiramo u njegovoj vlastitoj domeni, a ne u domeni programskog jezika u kojem modeliramo rješenje. Primjerice, ako radimo aplikaciju koja će se brinuti o studentima i njihovim položenim ispitima tada nam apstrakcija daje jasnu strukturu modela Student i Ispit. Kod razreda Student nama neće biti važno kojeg je student spola, visine, težine itd., već njegov jedinstveni matični broj akademskog građanina, godina upisa fakulteta, godina na kojoj se nalazi te eventualno prosjek svih položenih ispita. Ispit će sadržavati datum polaganja, šifru kolegija iz kojeg se ispit polagao i ocjenu. Pri modeliranju tih razreda nećemo se voditi mogućnostima i specifičnostima programskog jezika u kojem radimo. Nakon što odredimo kako naši modeli izgledaju, odabir prikladnih struktura i tipova podataka je zapravo trivijalan.

#### **2.4. Zanimljivi linkovi o temi**

1. Wikipedia, URL: <http://en.wikipedia.org/wiki/Abstraction>, (2.1.2007.)
2. Edward V. Berard, Abstraction, Encapsulation, and Information Hiding ,  
URL: <http://www.toa.com/pub/abstraction.txt> , (10.1.2007.)

### 3. ENKAPSULACIJA

#### 3.1. Definicija

Enkapsulacija je povezivanje bitnih informacija i elemenata nekog objekta te njegovog ponašanja (metoda) u povezane, izolirane i nepovredive cjeline na takav način da unutrašnji ustroj nije vidljiv vanjskom svijetu.

#### 3.2. Općenito o enkapsulaciji

Enkapsulacija tj. učahurivanje je koncept za koji postoji nekoliko vrlo sličnih, a opet različitih definicija. Najčešće su one koje poistovjećuju učahurivanje sa skrivanjem informacija (engl. information hiding [1]). U sljedećem poglavlju ćemo pokazati da to nije sasvim točno. Naime, kad bi enkapsulacija bila isto što i skrivanje informacija, tada bi sve što je enkapsulirano nužno bilo i skriveno.

Enkapsulacija, kao i apstrakcija može biti proces i entitet. Kao proces ona opisuje ograđivanje informacija nekom fizičkom ili logičkom barijerom. Kao entitet enkapsulacija je paket ili posuda (engl. container) koja sadrži jedan ili više elemenata. Sam mehanizam definiranja barijere koja odvaja cjelinu od vanjskog svijeta ovdje se ničime ne definira.

Podaci su u objektno orijentiranom programiranju kritičan element u razvoju aplikacija. Puno pažnje se posvećuje organizaciji i granularizaciji podataka. Isto tako važno je zaštititi ih od tipičnih grešaka koje se događaju prilikom programiranja, kao i smanjiti ili potpuno ukloniti povezanost implementacije od sučelja objekta.

Pretpostavimo da imamo dva razreda, Kvadrat i Printer. Kvadrat ima metodu SadrziTocku(int x, int y) koja za koordinate (x,y) vraća „true“ ako je točka unutar samog kvadrata, inače „false“. Razred Printer zna nacrtati sliku nekog kvadrata, a to čini pozivajući metodu SadrziTocku za svaku koordinatu svog platna. Iz perspektive Printera sve što je važno jest da objekt nad kojim zove metodu zna reći da li sadrži predane koordinate ili ne.

Sama metoda može se ostvariti na više načina, npr.: za svaku koordinatu se može provjeriti da li se nalaze između najviše ( krajnje lijeve) i najniže(krajnje desne) točke. Ako u nekom trenutku smislimo bolju, kraću ili elegantniju implementaciju metode moramo osigurati da razred Printer toga ne bude svjestan. Naime, njemu nije važno kako metoda provjerava pripadnost koordinata liku, već samo da to zna učiniti.

*«Niti jedan dio složenog sustava ne bi smio ovisiti o (oslanjati se na) unutrašnje detalje nekog drugog sustava».*

Jednostavnije rečeno, želi se postići da sam način na koji je izvedena funkcionalnost nekog objekta 'A' ne utječe niti na jedan

drugi objekt 'B' koji ga koristi. To se postiže dobrim definiranjem sučelja razreda 'A'. Ako drugim objektima izložimo samo sučelje objekta 'A', tada smo sigurni da bilo kakva buduća promjena implementacije neće nikako utjecati na objekte koji ga koriste.

### 3.3. Enkapsulacija i skrivanje informacija

Enkapsulacija se može postići ograničavanjem pristupa unutrašnjim podacima nekog objekta. Prava pristupa pojedinim metodama i elementima razreda (tj. objekta) prilikom deklaracije mogu se definirati korištenjem ključnih riječi:

1. **public**: elementi su dio javnog sučelja razreda i dostupnim svim drugim razredima
2. **private**: elementi su dostupni samo razredu u kojem su deklarirani i prijateljima (engl. friends)
3. **protected**: elementi su dostupni razredu u kojem su deklarirani, podrazredima i prijateljima

Postoje još i ključne riječi **internal** i **protected internal** [2].

Različiti programski jezici ključnim riječima definiraju različite razine prava pristupa elementima. Ovisno o odabranoj ključnoj riječi, možemo postići učinkovitu kontrolu podataka, pošto je pristup istima djelomično ili potpuno filtriran. Ako elementima objekta pridodamo ključnu riječ **private**, niti jedan drugi objekt izvana im neće moći direktno pristupiti i ti podaci sami za sebe doista jesu skriveni. Na cijeli objekt možemo gledati kao na crnu kutiju čija su interna stanja zaštićena od bilo kakve korupcije izvana.

Ključnom riječi **public** pristup podacima je omogućen svima, i oni nisu ni na koji način skriveni od vanjskog svijeta. Neovisno o tome, ti podaci su i dalje povezani u smislenu cjelinu koja je izolirana od okoline nekom barijerom.

Točnije bi bilo reći da je skrivanje implementacije podataka jedan od mehanizama kojima se enkapsulacija ostvaruje, a ne nužno njezin sinonim. Cilj nije sakriti same podatke. Skriva se njihova implementacija, a može im se i dalje pristupiti i njima raspolagati, no samo kroz vrlo pažljivo definirano sučelje objekta.

### 3.4. Odnos enkapsulacije i apstrakcije

Jedna od čestih pogreška je poistovjećivanje enkapsulacije sa apstrakcijom. Ova dva koncepta objektno orijentirane paradigme su usko povezana, no treba imati na umu da su oni komplementarni a ne identični. Apstrakcijom se identificiraju i definiraju bitne vrijednosti i temeljno ponašanja nekog objekta. Enkapsulacijom se ti

najbitniji dijelovi povezuju u ograđene cjeline, a otkrivaju samo oni bitni i na strogo kontroliran način.

### 3.5. Zanimljivi linkovi

1. Wikipedia, URL: [http://en.wikipedia.org/wiki/Information\\_hiding](http://en.wikipedia.org/wiki/Information_hiding), (02.02.2007.)
2. MSDN, URL: <http://msdn2.microsoft.com/en-us/library/ms173121.aspx> , (02.02.2007.)
3. JavaWorld, URL: <http://www.javaworld.com/javaworld/jw-05-2001/jw-0518-encapsulation.html>, (05.02.2007.)
4. Liberty Associates, URL: <http://www.libertyassociates.com/pages/column2.htm>, (03.02.2007.)
5. TOA, URL: <http://www.toa.com/pub/abstraction.txt>, (08.02.2007.)
6. Webopedia, URL: <http://www.webopedia.com/TERM/e/encapsulation.html> , (02.02.2007.)
7. Wikipedia, URL: [http://en.wikipedia.org/wiki/Separation\\_of\\_concerns](http://en.wikipedia.org/wiki/Separation_of_concerns) , (02.02.2007.)
8. Microsoft TechNet, URL: [http://www.microsoft.com/technet/archive/wfw/7\\_agloss.msp?mfr=true](http://www.microsoft.com/technet/archive/wfw/7_agloss.msp?mfr=true) , (10.02.2007.)

## 4. NASLJEĐIVANJE

U ovom poglavlju objašnjen je vrlo važan koncept objektno orijentirane paradigme – nasljeđivanje, implementacija nasljeđivanja u programskom jeziku C#, dan skup primjera kojim je dodatno objašnjen sam koncept nasljeđivanja.

### 4.1. Definicija

**Nasljeđivanje** (eng. inheritance) je tehnika kojom se definiranje nekog razreda vrši korištenjem definicije postojećeg razreda (koji se naziva **bazni razred**). Tako dobiveni razred naziva se **izvedeni razred**. Kada se pomoću izvedenih razreda deklarira neki objekt, on nasljeđuje funkcije i varijable kreirane u baznom razredu. Dakle sve funkcije i varijable kreirane u baznom razredu mogu se koristiti u našem izvedenom razredu.

### 4.2. Općenito o nasljeđivanju

Kako je nasljeđivanje jedan od temeljnih koncepata objektno orijentiranog programiranja, a sama definicija pomalo zbunjujuća, objasnimo je na jednom opće poznatom primjeru.

Sjetimo se biologije i učenja o sisavcima. Definirali smo razred sisavaca sa nekim općim svojstvima: ženke proizvode mlijeko, koža je pokrivena dlakama, toplokrvni su. To su njihova opća svojstva, no u razredu sisavaca postoje različite vrste životinja koje imaju različite osobine, a ipak i neka zajednička svojstva svih sisavaca. Kažemo da sve vrste sisavaca (npr. mačka, pas, majmun, dupin) nasljeđuju opće osobine, te dodaju neke svoje osobine (koncept polimorfizma, iako ne mora biti nužno jer se pojedine osobine svojstvene za neku životinju mogu jednostavno nadodati u izvedeni razred, ali o tome će biti više govora u sljedećem poglavlju.)).

Kao što možemo vidjeti na slikama postoje velike razlike među pojedinim vrstama sisavaca, a isto tako i velike sličnosti među njima.



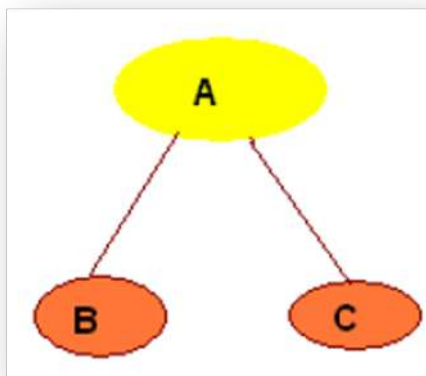


Po uzoru na svijet oko nas izgrađen je koncept objektno orijentiranog programiranja, pa tako i koncept samog nasljeđivanja.

Iako i u stvarnom svijetu koristimo koncept nasljeđivanja da bismo odredili hijerarhijske odnose i tako si olakšali život, kod objektno orijentiranih jezika to je puno izraženije.

Kod objektno orijentiranog programiranja nasljeđivanje pomaže učinkovitom korištenju već postojećeg koda s minimalnim modifikacijama. Prednost nasljeđivanja je u tome što moduli dovoljno sličnih sučelja mogu dijeliti kôd na taj način smanjujući kompleksnost programa. Na primjer, definiramo razred *auto* te razrede *Mercedes* i *Fićo*. Svaki *Mercedes* je *auto* sa specifičnim karakteristikama *auta* (ima 4 kotača, volan, troši gorivo, vozi, ima kočnicu, itd.), a jednake karakteristike ima i *Fićo*. Dakle razredi *Mercedes* i *Fićo* su izvedeni razredi, a razred *Auto* je bazni razred. Kada govorimo o nasljeđivanju, možemo to promatrati i s aspekta poopćenja, konkretno u našem primjeru *Auto* je poopćenje *Mercedesa* i *Fiće*, iako se mnogi ne bi s time složili.

Osnovna ideja nasljeđivanja je u tome da se prilikom razvoja programa identificiraju razredi koji imaju sličnu funkcionalnost, te se u izvedenim razredima samo redefiniraju specifična svojstva, dok se preostala svojstva nasljeđuju u nepromijenjenom obliku. Na slici je prikazan princip nasljeđivanja:



Podrazredi B i C imaju pristup izvedenim varijablama i metodama razreda A kao da su deklarirani u njima. To znači da je metodu izvedenu iz razreda A moguće dohvatiti u B ili C putem njenog imena jednako kao da je definirana u B ili C. Razred A još zovemo **nadrazredom** (engl. superclass) razreda B i C. Potrebno je naglasiti **da razred A mora biti definiran prije razreda B i C**. Možemo reći da je osnovna svrha nasljeđivanja upravo omogućavanje korištenja varijabli i metoda definiranim u nekom drugom razredu.

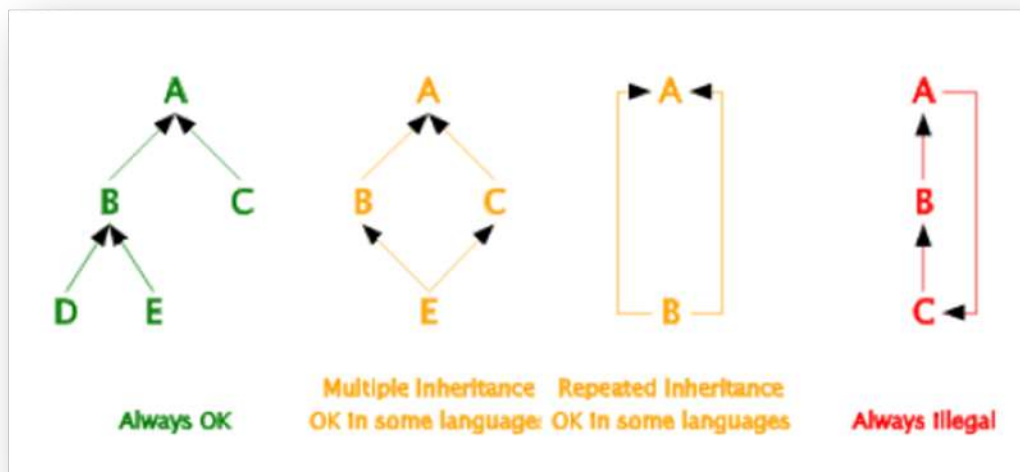
Zamislimo na trenutak što bi sve bilo potrebno napraviti da ne postoji mogućnost nasljeđivanja. Morali bi "kopirati" varijable i metode iz razreda A i "zalijepiti" ih u razrede B i C. Zamislite da ste otkrili grešku u nekoj od metoda. Te greške je potrebno ispraviti na sva-

kom od mjesta gdje smo je u prethodnom postupku kopirali, što do-  
vodi do većih mogućnosti pogreške. Naime, možda ne uspijemo is-  
praviti grešku na svim mjestima jer smo npr. zaboravili gdje je sve  
kopiran kod. Suvišno je reći da na kraju imamo nekoliko puta više  
koda koji je teško čitljiv.

Iako se na prvi pogled nasljeđivanje čini vrlo praktičnom i korisnom  
metodom rješavanja svih mogućih problema, ima i svoje mane od  
kojih je najizraženija činjenica da se promjena metode u baznom  
razredu reflektira na promjenu te iste metode u svim izvedenim raz-  
redima. Zbog toga postoji mogućnost da program neće radi ono što  
se od njega očekuje. Stoga je potrebno vrlo pažljivo koristiti kon-  
cept nasljeđivanja. Kada koristiti nasljeđivanje, a kada jednostavno  
"kopirati" dijelove koda naučit ćete s iskustvom. Iskusni programeri  
koriste nasljeđivanje isključivo u slučajevima kad je to nužno po-  
trebno.

U ovom trenutku važno je naučiti kako implementirati nasljeđivanje,  
a s vremenom kako i kada ga najbolje koristiti, te se ovim proble-  
mom nećemo dodatno zamarati. Za više informacija o tome kada  
koristiti nasljeđivanje, a kada ne pogledajte na linku na kraju po-  
glavlja.

Postoji nekoliko vrsta nasljeđivanja. U programskim jezicima koji  
poržavaju objektno orijentiranu paradigmu nisu implementirani svi  
dolje navedeni oblici nasljeđivanja.. Ako ih prikazemo grafovima do-  
bijemo slijedeću sliku:



Prvi graf opisuje "normalno" **jednostruko nasljeđivanje** (engl. single-inheritance) gdje je moguće nasljeđivanje isključivo iz jednog baznog razreda. Kod tog oblika nasljeđivanja hijerarhijska struktura uvijek je stablo. **Svi objektno orijentirani jezici imaju mogućnost barem jednostrukog nasljeđivanja.**

Drugi graf opisuje **višestruko nasljeđivanje** (engl. multiple-inheritance) koje podržavaju programski jezici kao što su C++ i Eiffel.

**Višestruko nasljeđivanje u programskom jeziku C# nije implementirano.** Kod višestrukog nasljeđivanja izvedeni razredi nasljeđuju svojstva iz barem dva bazna razreda.

**Ponavljajuće nasljeđivanje** (eng. repeated inheritance) koristi se rijetko i podržano je programskim jezikom Eiffel.

Posljednji graf prikazuje ono što nikada ne smijemo imati – cikličko nasljeđivanje. Nije moguće nasljeđivati iz razreda koji je izveden iz razreda koji mislite izvoditi.

Sada kada smo uveli osnovne pojmove vezane uz nasljeđivanje i saznali ideju koja nas treba voditi, objasnimo kako se implementira nasljeđivanje u C#.

### 4.3. Nasljeđivanje u C#

#### 4.3.1 Nasljeđivanje razreda

U trenutku kreiranja, uz ime razreda kojeg kreiramo moguće je naznačiti razred čija svojstva želimo naslijediti u novo kreiranom razredu. To činimo na sljedeći način:

```
class A {}  
class B: A{}
```

Ukoliko ne navedemo ime baznog razreda od kojeg smo naslijedili svojstva i atribut podrazumijeva se da smo napisali:

```
class B: Object
```

Svi razredi kojima nije eksplicitno navedeno iz kojeg su razreda izvedeni, izvedeni su iz razreda Object. O razredu Object govorit ćemo nešto kasnije.

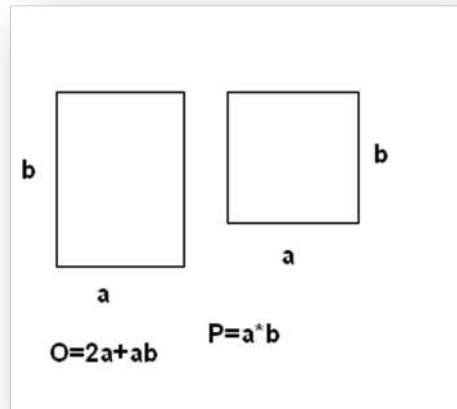
U nastavku je dan "recept" za implementaciju nasljeđivanja:

```
public class A  
{  
    public A() { }  
}  
public class B : A  
{  
    public B() { }  
}
```

Razred B - izvedeni razred stječe sve podatke koji nisu privatni i ponašanje baznog razreda kao dodatak vlastitim podacima i ponašanju. Nakon što smo dali "recept" kako implementirati nasljeđivanje, navedimo nekoliko općenitih primjera.

U sljedećem primjeru prikazano je jednostavno nasljeđivanje metoda i varijabli iz baznog razreda. Najprije kreirajmo razred Pravokutnik, te nakon njega razred Kvadrat koji nasljeđuje svojstva razreda

Pravokutnik : površina i opseg. Razred Kvadrat nasljeđuje svojstva površine i opsega:



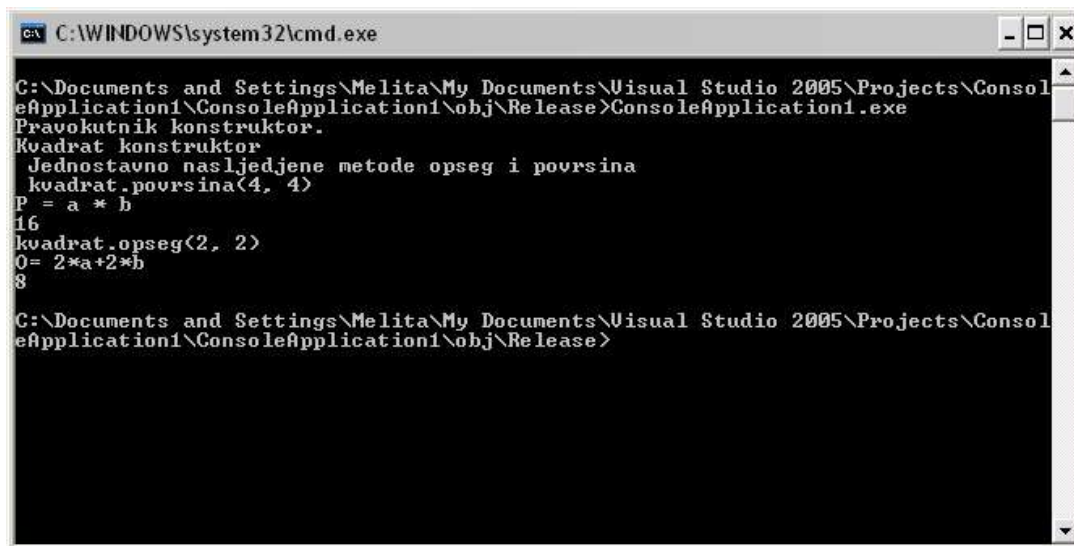
```
using System;
public class Pravokutnik
{
    public Pravokutnik()
    {
        Console.WriteLine("Pravokutnik konstruktor.");
    }
    public void površina(float a, float b)
    {
        float P = a * b;
        Console.WriteLine("P = a * b");
        Console.WriteLine(P);
    }
    public void opseg(float a, float b)
    {
        float O= 2*a+2*b;
        Console.WriteLine("O= 2*a+2*b");
        Console.WriteLine(O);
    }
}

public class Kvadrat : Pravokutnik
{
    public Kvadrat()
    {
        Console.WriteLine("Kvadrat konstruktor");
    }
}

public static void Main()
{
    Kvadrat kvadrat = new Kvadrat();
    Console.WriteLine("Jednostavno nasljedjene metode opseg i površina");
    Console.WriteLine(" kvadrat.povrsina(4, 4)");
    kvadrat.povrsina(4, 4);
    Console.WriteLine(" kvadrat.opseg(2, 2)");
    kvadrat.opseg(2, 2);
}
```

```
}  
}
```

Ispis programa:



```
C:\WINDOWS\system32\cmd.exe  
C:\Documents and Settings\Melita\My Documents\Visual Studio 2005\Projects\ConsoleApplication1\ConsoleApplication1\obj\Release>ConsoleApplication1.exe  
Pravokutnik konstruktor.  
Kvadrat konstruktor  
Jednostavno nasljedjene metode opseg i površina  
kvadrat.povrsina(4, 4)  
P = a * b  
16  
kvadrat.opseg(2, 2)  
O= 2*a+2*b  
8  
C:\Documents and Settings\Melita\My Documents\Visual Studio 2005\Projects\ConsoleApplication1\ConsoleApplication1\obj\Release>
```

Kao što možemo vidjeti u ispisu prvo se poziva konstruktor razreda Pravokutnik, a zatim konstruktor izvedenog razreda Kvadrat koji nasljeđuje nepromijenjene metode za izračunavanje opsega i površine. Instancirali smo objekt razreda Kvadrat a zatim pozvali metode koje smo naslijedili.

Promotrimo što smo dobili ovim primjerom. Razred Kvadrat može koristiti sve što ima razred Pravokutnik, nije potrebno ponovno pisati metode opseg i površina nego ih je moguće ponovno iskoristiti.

Isto tako izvedeno razredi mogu dodavati i neke svoje posebnosti koje bazni razred NE vidi. To je malo teže prikazati na primjeru razreda Pravokutnik i Kvadrat, ali ako definiramo bazni razred Telefon i izvedeni razred Mobitel koji nasljeđuje svojstvo, a dodaje na primjer svojstvo jacinaSignala koja je svojstvena za objekte razreda Mobitel, dok su nasljeđena svojstva iz razreda Telefon na primjer Pozivaje i prekidanjePoziva.

Objekte instancirane iz razreda izvedenih iz baznih razreda moguće je pretvoriti **cast** operatorom u objekte baznih razreda. Time se odbacuju posebnosti koje je uveo izvedeni razred, a zadržava se sve što je nasljeđeno iz baznog razreda.

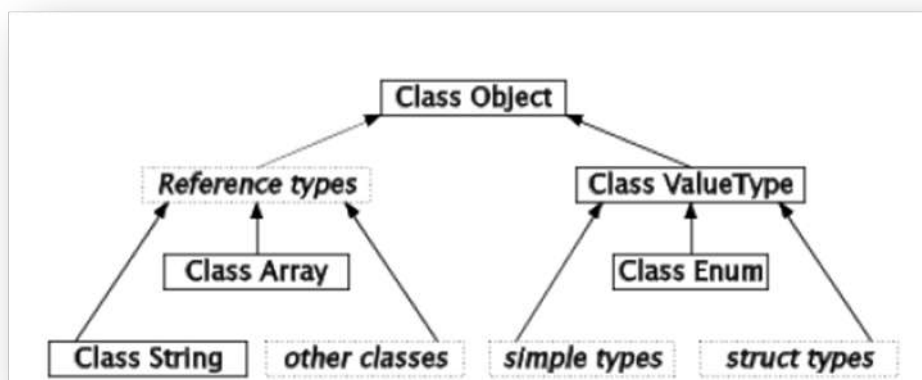
To možemo napraviti na sljedeći način. Neka imamo definiran razred Telefon i izvedeni razred Mobitel kao u gornjem primjeru te želimo objekt razreda Mobitel pretvoriti cast operatorom u objekt razreda Telefon.:

```
Mobitel mobitel = new Mobitel();  
Telefon telefon;  
telefon = (Mobitel)mobitel;
```

Time više nije moguće koristiti svojstvo `jacinaSignala`. Castanje objekta baznog razreda u izvedeni razred nije moguće te kompajler javlja pogrešku.

#### 4.3.1.1. Razred Object

Objasnimo razred `Object`. Ovaj razred nalazi se na vrhu hijerarhijskog stabla svih razreda. Metode razreda `Object` dostupne su u svim tipovima u C#, uključujući i vrijednosne tipove.



Lijeva grana odgovara **referentnim tipovima** podataka u C#, a desna strana odgovara **vrijednosnim tipovima**. Svi već kreirani razredi i oni razredi koje sami definiramo su referentnog tipa, kao što su stringovi (razred `String`) i polja (razred `Array`). U stvarnosti ne postoji razred `Reference types`, ali je dodan na sliku radi boljeg razumijevanja. Razred `ValueType` je glavni razred svih vrijednosnih tipova. Ovdje nećemo dublje ulaziti u samu strukturu razreda `Object`. Dovoljno je zapamtiti da je on glavni razred na vrhu hijerarhijske ljestvice od koje su nasljeđeni svi ostali razredi. Ukoliko vas ova tematika više zanima proučite sljedeći [link](#).

#### 4.3.2. Nasljeđivanje i konstruktor

Konstruktor služi da bismo kreirali objekt nekog razreda. Jedina metoda koju nije moguće naslijediti je konstruktor razreda. određena pravila za konstruktor razreda i nasljeđivanje:

- svaki razred bi trebao imati svoj konstruktor
- konstruktor izvedenog razreda trebao bi surađivati s konstruktorom nadređenog razreda izvedenom razredu da bi inicijalizirao objekt izvedenog razreda.
- konstruktor izvedenog razreda će uvijek pozivati konstruktor nadređenog razreda

Pri inicijalizaciji objekta izvedenog razreda redoslijed inicijalizacije varijabli i pozivanja konstruktora je slijedeći:

1. primjerci varijabli u izvedenom razredu su inicijalizirani
2. primjerci varijabli u baznom razredu su inicijalizirani
3. konstruktor baznog razreda je pozvan
4. konstruktor izvedenog razreda je pozvan

Ukoliko želimo koristiti parametre u konstruktorima potrebno je iz nasljeđenih razreda pozvati konstruktor baznog razreda pomoću `base`.

#### **4.4 Zanimljivi linkovi:**

1. Razred Object ,URL:

<http://msdn2.microsoft.com/en-us/library/system.object.aspx>  
(2.2.2007.)

2. Kada koristiti nasljeđivanje, a kada ne, URL:

<http://forum.java.sun.com/thread.jspa?threadID=224280&messageID=789257> (2.2.2007)

3. Definicija i općenito o nasljeđivanju :

<http://en.wikipedia.org/wiki/Inheritance> (24.3.2007)

4. Nasljeđivanje implementirano u C#:

<http://www.c-sharpcorner.com/UploadFile/grusso/ImplInherit12032005000508AM/ImplInherit.aspx> (24.3.2007)

<http://www.exforsys.com/content/view/1748/360/> (24.3.2007)

5. Nasljeđivanje i polimorfizam u C#:

<http://www.codeproject.com/csharp/csharpintro01.asp>  
(24.3.2007)

## 5. POLIMORFIZAM

U ovom poglavlju objasniti ćemo koncept polimorfizma, te kako implementirati polimorfizam u programskom jeziku C#. Za početak definirajmo što je to polimorfizam.

### 5.1. Definicija

**Polimorfizam** (engl. Polymorphism) je svojstvo objektno orijentiranog sustava da se izvedeni razredi iz nekog baznog razreda mogu referencirati preko tipa tog baznog razreda.

### 5.2. Općenito o polimorfizmu

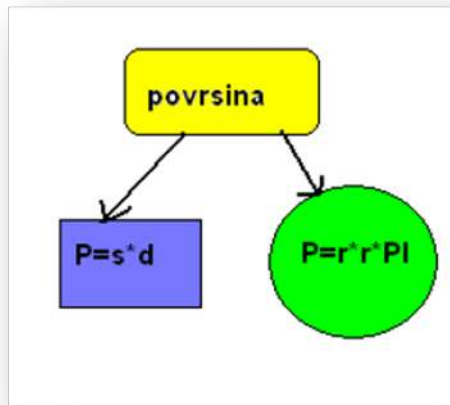
Polimorfizam je, uz enkapsulaciju, apstrakciju i nasljeđivanje treće važno svojstvo koje objektno orijentiran jezik mora podržavati. Bez nasljeđivanja nema polimorfizma, a nasljeđivanje bez polimorfizma gotovo da i nema smisla.

Kod objektno orijentiranog programiranja, pozivanje metode se može smatrati slanjem poruke objektu na koje objekt odgovara izvršavanjem prikladne metode. Budući da objekt zna koje je vrste, onda zna i kako reagirati na tu poruku. Polimorfnost znači samo da različiti objekti mogu različito reagirati na istu poruku. Ako razmislimo o tome što nam sama riječ polimorfizam govori možemo riješiti u startu nekoliko nedoumica: poli – mnogo, form – oblik = više oblika. Neka nam taj izraz više oblika bude misao vodilja u shvaćanju ovog pojma.

Pokušajmo polimorfizam kao i u prethodnom poglavlju, objasniti na jednom primjeru iz svakodnevnog života. Na primjer, mogli bismo sve ljude klasificirati u hijerarhiju razreda koja počinje s podjelom ljudi po boji kože (jest da bi tada bili prozvani rasistima, ali zanemarimo tu činjenicu na trenutak). Svaki od tih ljudi će na pitanje "Kako se zoveš" odgovoriti na jednak način, bez obzira na to da li na njega gledamo kao azijata lieuropljana. Odgovor na to pitanje nije određen objektom nego okolinom.

Drugi općepoznati primjer na kojeg ćete najčešće naići u literaturi je primjer s geometrijskim likovima. Neka je bazni razred Geometrijski\_lik, a izvedeni razredi su Krug i Pravokutnik. Svaki Geometrijski\_lik ima svoju površinu, ali se ona za Krug izračunava na drugačiji način nego za Pravokutnik. Za krug je  $\text{radijus} \times \text{radijus} \times \text{PI}$ , a za pravokutnik je  $\text{duljina} \times \text{sirina}$ , ali općenito svojstvo svakog geometrijskog lika je njegova površina kao što je prikazano na slici.





Svojstva svih geometrijskih likova su npr. površina, a svaki od geometrijskih likova tumači to svojstvo na sebi prikladan način. Dakle pravokutnik površinu tumači kao:  $P=s*d$ , dok krug površinu tumači kao:  $P= r*r*PI$ . Kada smo objasnili koncept polimorfizma, naučimo kako implementirati polimorfizam u programskom jeziku C#

### 5.3. Polimorfizam u C#:

Prje svega navedimo jedno svojstvo kojeg se moramo uvijek pridržavati: metodu iz baznog razreda moguće je nadjačati jedino ako je ona u baznom razredu deklarirana kao virtualna metoda. U izvedenom razredu je za metodu koju nadjačavamo potrebno koristiti ključnu riječ `override`.

U nastavku je dan općeniti "recept" za implementaciju polimorfizma:

```

using System;
namespace Polymorphism
{
    class A
    {
        public virtual void Foo()
        { Console.WriteLine("A::Foo()"); }
    }
    class B : A
    {
        public override void Foo()
        { Console.WriteLine("B::Foo()"); }
    }
}
  
```

Kao što možemo vidjeti u "receptu", prvo je potrebno kreirati bazni razred A() koji u sebi sadrži neku općenitu metodu Foo koja je virtualna metoda. Izvedeni razred B() iz razreda A() sadrži metodu Foo() za koju je navedeno da je ona nadjačala metodu iz razreda A (ključna riječ `override`).

Osim nadjačanih metoda postoje i skrivene metode (*eng. hiding method*). Kažemo da ukoliko metoda ne nadjačava izvedenu metodu, ona je sakriva. Skrivenu metodu potrebno je deklarirati ključnom riječi `new`. "Recept" je dan u nastavku:

```

using System;
namespace Polymorphism
{
class A
{
public void Foo() { Console.WriteLine("A::Foo()"); }
}
class B : A
{
public new void Foo()
{ Console.WriteLine("B::Foo()"); }
}
}

```

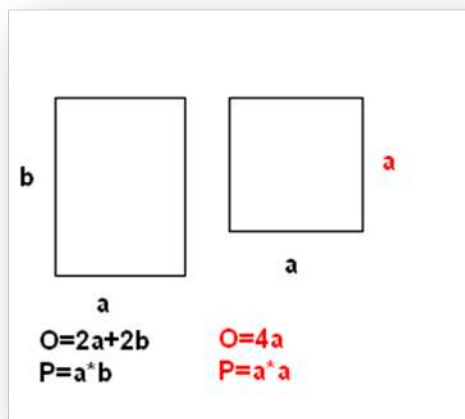
Ukoliko pozovemo metode:

```

a = new A();
b = new B();
a.Foo(); // output --> "A::Foo()"
b.Foo(); // output --> "B::Foo()"
a = new B();
a.Foo(); // output --> "A::Foo()"

```

Kako je recept vrlo općenit i ne govori nam puno o ponašanju samih razreda i metoda, u nastavku je dato nekoliko primjera implementacije polimorfizma. Podsjetimo se primjera baznog razreda Pravokutnik i izvedenog razreda Kvadrat u poglavlju o nasljeđivanju. Promijenimo metode opseg i površina na sljedeći način:



Postoji još jedna mogućnost kada metoda može biti i virtualna i izvedena kao skrivena metoda, kao u sljedećem primjeru:

```

class A
{
public void Foo() {}
}
class B : A
{

```

```
public virtual new void Foo() {}
}
```

Ako je metoda ujedno i virtualna i skrivena potrebno je to naznačiti navođenjem ključnih riječi virtual i new. Takvu metodu koristimo u izvedenom razredu C() na sljedeći način:

```
class C : B
{
    public override void Foo() {}
    // ili
    public new void Foo() {}
}
```

Kao što vidimo metodu Foo u izvedenom razredu možemo koristiti ili kao nadjačanu ili kao skrivenu metodu što u dosta slučajeva može biti korisno.

Dakle potrebno je zapamtiti:

- isključivo metode koje su u baznom razredu navedene kao virtualne moguće je u izvedenim razredima prepravljati
- Kada se te metode u izvedenim razredima prepravljaju potrebno je navesti ključnu riječ override

Za više primjera pogledati linkove u dijelu examples.

## 5.4 Zanimljivi linkovi

1. Wikipedia, URL: [http://en.wikipedia.org/wiki/Polymorphism\\_%28computer\\_science%29](http://en.wikipedia.org/wiki/Polymorphism_%28computer_science%29) (7.2.2007)
2. Coodeproject, URL: <http://www.codeproject.com/csharp/csharpintro01.asp> (7.2.2007.)
3. csharp-station, URL: <http://www.csharp-station.com/Tutorials/Lesson09.aspx> (7.2.2007.)
4. Examples:
  - <http://msdn2.microsoft.com/en-us/library/ms173152.aspx> (24.3.2007.)
  - <http://www.softsteel.co.uk/tutorials/cSharp/lesson14.html> (24.3.2007)
  - <http://nazish.blog.com/1396268/> (24.3.2007)
  - [http://www.codersource.net/csharp\\_tutorial\\_oops.html](http://www.codersource.net/csharp_tutorial_oops.html) (23.4.2007)
  - <http://www.devarticles.com/c/a/C-Sharp/Inheritance-and-Polymorphism/> (23.4.2007)
  - <http://www.functionx.com/csharp/classes2/Lesson02.htm> (23.4.2007)
  -

## 6. SUČELJA

### 6.1. Definicija

U najopćenitijem obliku, sučelje je skup (popis) metoda koje nisu implementirane, tj. skup potpisa metoda (engl. method signatures).

Sintaksa:

```
[atributi][modifikator pristupa ] interface ime_sučelja [: base-lista]{  
    ...  
    tijelo_sučelja  
    ...  
}
```

gdje su

atributi: [\[1\]](#)

modifikatori pristupa: public, private, protected, internal i protected internal.

base-lista: popis sučelja koja se nasljeđuju.

## 6.2. Općenito o sučeljima

Objekt definira svoju interakciju s vanjskim svijetom kroz funkcije koje podržava. Npr. gumbi na televizoru čine njegovo sučelje koje odvaja elektroniku i način na koji je izvedena od krajnjeg korisnika (nas), koji želi gledati neki program. Bez obzira na proizvođača televizora i način na koji je elektronika organizirana, očekujemo da ćemo svaki televizor moći upaliti, ugasiti, promijeniti mu kanal, smanjiti ili povećati glasnoću itd. Sučelje je ugovor koji garantira ponašanje i osobine nekog razreda. Svaki razred koji ga implementira obavezuje se podržavati sve metode i elemente koje sučelje sadrži. Uobičajeno je ime sučelja započeti velikim slovom «I», premda ovo nije strogo pravilo.

## 6.3. Sučelja u C#

Pretpostavimo kako bi moglo izgledati sučelje za televizor:

```
public interface ITv {  
    bool Power {get; set;} //bez implementacije  
    svojstva, samo deklaracija  
    void On();  
    void Off();  
    void VolumeUp();  
    void VolumeDown();  
    void ChannelInc();  
}
```

```

        void ChannelDec();
    }

```

Ovo bi bila osnovna funkcionalnost koju očekujemo od svakog televizora. Metode nismo implementirali već samo deklarirali. Naime, svaki razred (recimo razred Tv) koji implementira to sučelje, neovisno o proizvođaču i modelu, metode može implementirati na način koji je njemu najprikladniji. Npr.: ako se radi o prijenosnom televizoru radit će na baterije, dok će ostali pomoću kabela biti spojeni u utičnicu. Sve što korisnike bilo kojeg od tih televizora zanima jest da su navedene metode podržane. Isto tako deklaracija metoda sučelja izostavlja modifikator pristupa. Sve metode su implicitno javne pošto je sučelje ugovor namijenjen korištenju drugim razredima.

```

public class TV : ITV {
    private int ScreenWidth;
    private int ScreenHeight;
    private static const int ChannelCount = 100;
    private int CurrentChannel = 0;
    private boolean PowerButton = false;
    public TV(){
    }
    public void On(){
        this.PowerButton = true;
    }
    public void Off(){
        this.PowerButton = false;
    }
    public void ChannelInc(){
        this.CurrentChannel = (this.CurrentChannel+1)%
        ChannelCount;
    }
    public void ChannelDec(){
        ...implementacija!
    }
    public void VolumeUp(){
        ...implementacija!
    }
    public void VolumeDown(){
        ...implementacija!
    }
}

```

### 6.3.1. Razlika sučelja i apstraktnih razreda

Sučelja su slična apstraktnim razredima, no za razliku od apstraktnog razreda koji služi kao temelj za skup podrazreda koji će ga naslijediti, sučelja se mogu koristiti u više različitih struktura nasljeđivanja. Također, samo sučelje ne može sadržavati implementaciju u metodama, dok apstraktni razredi mogu.

Zamislamo da postoje razredi Cow, Sheep i Goat. Iako imaju neke svoje zajedničke značajke, sve tri su potpuno različite životinje. Pogledajmo sljedeće sučelje:

```
interface IMilkable {  
    void GetMilk();  
}
```

Sasvim je legalno napisati sljedeće:

```
public class Cow : IMilkable {  
    //implementacija  
}  
  
public class Goat : IMilkable {  
    //implementacija  
}  
  
public class Sheep : IMilkable {  
    //implementacija  
}
```

Dakle, bitno je uočiti da se isto sučelje može implementirati u više različitih razreda. Još jedna važna činjenica je da razredi koji implementiraju sučelje čak ne moraju biti u istim strukturama nasljeđivanja:

```
interface IKontroliraj {  
    void Upali();  
    void Ugasi();  
}  
  
public class Auto : IKontroliraj {  
    //implementacija Upali() okreni ključ  
    //implementacija Ugasi() okreni ključ u drugom smjeru  
}
```

```

public class Zarulja : IKontroliraj {
    //implementacija Upali()   pritisni prekidač
    //implementacija Ugasi()   pritisni prekidač
}

public class Video : IKontroliraj {
    //implementacija Upali()   stisni gumb na daljinskom
    //implementacija Ugasi()   opet stisni gumb na
    daljinskom
}

```

Jasno je da razredi Video, Zarulja i Auto nemaju apsolutno ništa zajedničko, osim metoda koje služe njihovom upravljanju. A premda se metode u razredima jednako zovu, njihova implementacija je vrlo različita.

### 6.3.2. Razlika sučelja i razreda

Za sučelja ne možemo reći da su razredi. Već smo utvrdili da višestruko naslijeđivanje razreda u C# nije podržano (razred može imati samo jednog "roditelja"), no razred može implementirati više od jednog sučelja i time ostvariti veću funkcionalnost. Definirajmo dva sučelja, IReadable s metodom read() i IWriteable s metodom write(), te razred Letter koji implementira ta dva sučelja.

```

interface IReadable {
    void Read();
}

interface IWriteable {
    void Write(string s);
}

public class Letter : IReadable, IWriteable {
    private string letter;
    public Letter () {} //konstruktor
    public void Read() {
        System.Console.WriteLine("Letter:{0}",letter);
    }
    public void Write(string s) {
        //implementacija
    }
    ...
}

```

```
}
```

Razred `Letter` sam odlučuje kako će implementirati ugovorene metode. Tekst bi se mogao pisati na standardni izlaz, u datoteku, bazu podataka, a isto vrijedi i za čitanje.

### 6.3.3. Nasljeđivanje i kombiniranje sučelja

Sučelje može naslijediti neko drugo sučelje, a moguće je čak kombinirati više njih kako bi se stvorilo neko novo sučelje proširene funkcionalnosti.

Uz manje preinake, prošli primjer bi mogao izgledati ovako:

```
interface IModifiable : IReadable, IWriteable{
    // kombiniranje sučelja
    int Version(get; set;); // dodatno svojstvo
    novokreiranog sučelja
}
```

Ovo sučelje sadrži popis svih metoda sučelja `IReadable` i `IWriteable` kao i novo svojstvo `version`.

```
public class Letter : IModifiable {
    //implementacija svega sadržanog u sučelju
    IModifiable
}
```

### 6.3.4. Korištenje metoda sučelja

Ako želimo pristupiti metodama sučelja, to možemo učiniti na dva načina. Prvi je kroz razred koji ga implementira:

```
Letter pismo = new Letter();
pismo.Write("Pozdrav!");
pismo.Version = 0;
```

Drugi način je korištenjem cast operatora kako bismo na objekt `Letter` gledali kroz sučelje `IModifiable`:

```
IModifiable jePismo = (IModifiable) pismo;
jePismo.Read();
jePismo.Version = 1;
```

Može se koristiti i kraći zapis:

```
IModifiable jePismo = (IModifiable) new Letter();
```

Sučelja nije moguće direktno instancirati, kao u sljedećoj liniji koda:



```
IModifiable jePismo = new IModifiable();//ono nije  
ispravno!!!!
```

Prije castanja treba provjeriti je li ono dozvoljeno. Ako želimo saznati da li razred implementira neko proizvoljno sučelje, to možemo pitati na sljedeći način:

*izraz **is** tip*

```
if (letter is IModifiable){  
    IModifiable jePismo = (IModifiable) pismo;  
    jePismo.Read();  
}
```

Bolja provjera bi bila sa operatorom **as**. On je učinkovitiji od operatora **is**, a ukoliko pretvorba nije dozvoljena jer tip sučelja nije implementiran u razredu, vratit će null referencu.

*izraz **as** tip*

```
Letter pismo = new Letter();  
IModifiable jePismo = pismo as IModifiable;  
if(jePismo != null) {  
    jePismo.Read();  
} else {  
    System.Console.WriteLine("IModifiable nije  
    podržano u razredu Letter");  
}
```

Kada koristiti koji operator? Ako želimo samo provjeriti implementira li razred neko sučelje, a ne namjeravamo ga pretvoriti u njega, tada je "is" bolji izbor. Ako znamo da ćemo objekt pretvoriti jer želimo pristupiti metodama sučelja, ako su podržane, tada je "as" operator učinkovitiji izbor.

Općenito, bolje je koristiti metode sučelja preko reference sučelja, jer nam to nudi prednosti polimorfizma. Ako više različitih razreda implementira sučelje a mi tim razredima pristupamo kroz samo sučelje, možemo ih koristiti neovisno o njihovom tipu. Dobar primjer za ovaj pristup se može vidjeti kod kolekcija. Uzmimo sučelje `IList`. `IList` nasljeđuje sučelje `ICollection` i služi kao osnovno sučelje za sve negeneričke liste. Implementacije `IList` sučelja se dijele na nekoliko kategorija: isključivo čitanje, promjenjiva i fiksna veličina. Lista predviđena samo za čitanje ne može se mijenjati. Liste fiksne veličine ne dopuštaju brisanja niti dodavanja novih elemenata, ali se postojeći elementi mogu modificirati, dok lista promjenjive veličine dopušta sve navedene radnje. Korisnik u nekom trenutku može zaključiti da je jedna od implementacija idealna za zahtjeve njegovog programskog zadatka, ovisno o složenosti operacija koje

dominiraju (iteriranje po listi, dodavanje elemenata, brisanje, itd.). Za nekoliko mjeseci možda neka druga operacija postane dominantna, a prvotni izbor postane neučinkovit. Mijenjanje deklaracija kroz kod bi moglo biti dugotrajno, zamorno i nepregledno, pogotovo ako se radi o nekom ozbiljnijem i većem projektu. Umjesto deklaracija poput:

```
ArrayList aList = new ArrayList();
```

možemo napisati i ovo:

```
IList aList = new ArrayList();
```

Ukoliko se pokaže da nismo odabrali dobru implementaciju, npr.; moramo imati osiguran poredak elemenata a znamo da ArrayList razred to ne garantira, dovoljno je samo odabrati drugu implementaciju na početku bez da razmišljamo gdje sve i kako koristimo objekt.

```
IList aList = new SortedList();
```

Osnovna funkcionalnost se nije promijenila jer su sve metode koje sučelje IList propisuje i dalje dostupne, a povećali smo učinkovitost bez da je ostatak koda svjestan promjene.

#### **6.4. Eksplicitna implementacija sučelja**

Što se događa kada jedan razred pokuša implementirati dva sučelja koja imaju metode s istim potpisom? Konflikt koji nastaje nalaže da razred koristi eksplicitnu implementaciju za bar jednu od problematičnih metoda. Pretpostavimo da razred Letter, osim sučelja IModifiable, želi implementirati i sučelje IAppendable s metodom write(). Tada prilikom implementacije moramo napisati sljedeće:

```
void IAppendable.write(string s) {...} //  
implementacija metode IAppendable sučelja  
  
public void write(string s); // jasno je da ova  
metoda pripada sučelju IModifiable
```

Metoda sučelja IAppendable ne može imati modifikator pristupa i po definiciji je javna. Također postoji samo jedan način korištenja eksplicitno implementirane metode:

```
Letter pismo = new Letter();  
IAppendable jePismo = pismo as IAppendable;  
if(jePismo != null) {  
    jePismo.write("Neki tekst");  
}
```

Ukoliko se ne bi koristilo castanje, korištenje eksplicitno implementirane metode ne bi bilo moguće.

## 6.5. Korisni i zanimljivi linkovi

1. Wikipedia, URL: [http://en.wikipedia.org/wiki/Interface\\_%28computer\\_science%29](http://en.wikipedia.org/wiki/Interface_%28computer_science%29) , (2.1.2007.)
2. Java, What Is an Interface,  
URL: <http://java.sun.com/docs/books/tutorial/java/concepts/interface.html> , (10.12.2006.)
3. MSDN, Interface (C# Reference),  
URL: <http://msdn2.microsoft.com/en-us/library/87d83y5b.aspx> , (10.12.2006.)
4. MSDN, System.Collections Namespace, URL: <http://msdn2.microsoft.com/en-us/library/system.collections.aspx> , (10.12.2006.)
5. MSDN, ICollection Interface, URL: <http://msdn2.microsoft.com/en-us/library/system.collections.ICollection.aspx> , (10.12.2006.)
6. MSDN, Generic Interfaces (C# Programming Guide),  
URL: <http://msdn2.microsoft.com/en-us/library/kwtft8ak.aspx> , (10.12.2006.)
7. C# Online.NET, Attributes and Reflection - Intrinsic Attributes,  
URL: [http://en.csharp-online.net/Attributes\\_and\\_Reflection%E2%80%9494Intrinsic\\_Attributes](http://en.csharp-online.net/Attributes_and_Reflection%E2%80%9494Intrinsic_Attributes) , (17.01.2007.)

## **7. UZORCI OBJEKTNO ORIJENTIRANOG DIZAJNA**

### **7.1. Definicija**

Uzorci oo dizajna su opisi objekata i razreda koji međusobno komuniciraju. Ti opisi su prilagođeni rješavanju općenitog problema dizajna unutar nekog specifičnog konteksta.

### **7.2. Općenito o uzorcima**

Kroz puno godina primijećeno je da se neki problemi u procesu razvoja aplikacija ili u arhitekturi aplikacije mogu riješiti na određene načine, koji se onda mogu primijeniti i u razvoju drugih aplikacija. Ta rješenja, opisana i dokumentirana, nazivaju se uzorci oo dizajna (engl, design patterns). Pažnja koja se posveti proučavanju zavisnosti i suradnje objekata neke aplikacije ili sustava može rezultirati manjim, jednostavnijim i puno razumljivijim rješenjem, kao i pomoći pri održavanju i dokumentaciji. Većina iskusnih programera tvrdi da dobro rješenje leži u proučavanju potreba i prepreka koje neki problem postavlja, te višestrukog korištenja onih metoda i prethodnih rješenja koja su se pokazala efikasnim. Za razliku od traženja potpuno novog rješenja "od nule", iskorištavanje postojećih uzoraka čini objektno orijentirani dizajn fleksibilnim, elegantnim i višestruko iskoristivim. Prilikom dizajna mogu se predvidjeti i neke buduće potrebe (koje u sadašnjosti još nisu izražene) i osigurati adekvatna podrška za njih i trenutak kada se pokaže potreba za doradom ili proširenjem postojećih sustava ili algoritama.

Sama definicija i shvaćanje uzorka je podložno stajalištu osobe koja ga koristi; ono što jedan programer doživljava kao uzorak, drugome može biti primitivan element korišten u svakodnevnom programiranju. Svrha uzorka jest da otkrije ključne značajke neke općenite strukture dizajna i raspodijeli uloge i odgovornosti elemenata koje će ponuditi kvalitetno i višestruko iskoristivo rješenje.

### **7.3. Što čini uzorak**

Uzorak dizajna je rješenje problema u nekom kontekstu, i svaki se uzorak sastoji od četiri osnovna elementa: imena, problema, rješenja i posljedica.

#### **7.3.1 Ime uzorka**

Ime koristimo da bismo u nekoliko riječi opisali problem dizajna, potencijalna rješenja i eventualne posljedice. Ovo nam omogućava dizajn na višoj razini apstrakcije. Uzorak može imati više od jednog imena, a poznavanje istih pojednostavljuje sporazumijevanje sa drugima kad želimo razgovarati o našem kodu, a isto je važno pri pisanju dokumentacije.

### 7.3.2. Problem

Problem upućuje na to kada treba koristiti uzorak, tj. daje objašnjenje i njegov kontekst. Može dati opis nekih specifičnih problema oo dizajna. Ponekad problem uključuje i popis uvjeta koji moraju biti zadovoljeni da bi primjena uzorka uopće imala smisla.

### 7.3.3. Rješenje

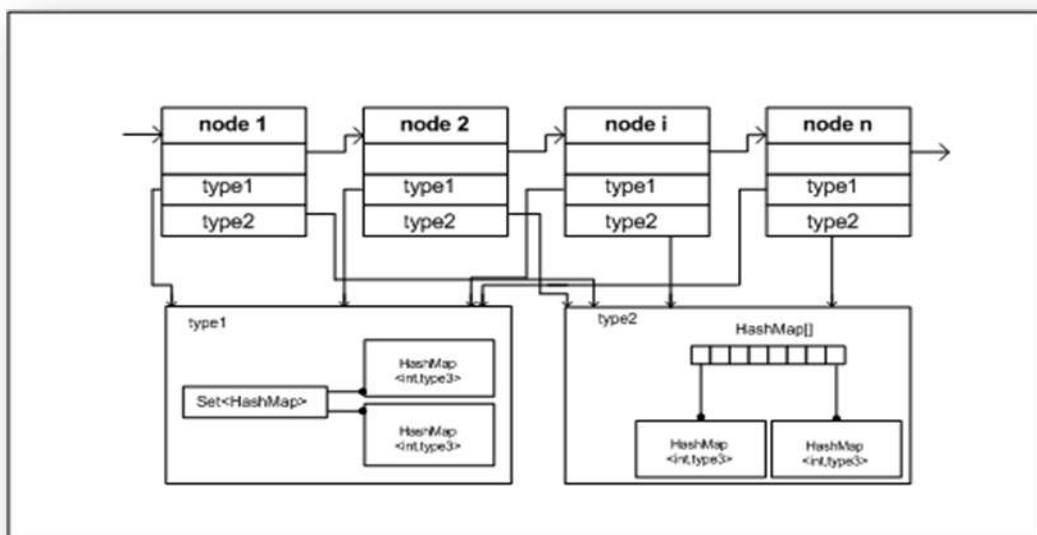
Rješenje daje jedan općenit opis sastavnih elemenata koji čine dizajn, predoduje njihove međusobne odnose, odgovornosti i suradnju, te način na koji ti elementi rješavaju problem. Ovo naravno ne uključuje konkretnu implementaciju pošto je uzorak predložak koji treba biti korišten u različitim situacijama.

### 7.3.4. Posljedice

Posljedice su rezultati i kompromisi korištenja nekog uzorka. Iako se rijetko spominju tijekom donošenja odluka, ključne su za procjenu cijene i koristi primjene nekog uzorka, kao i traženje drugih uzoraka koji bi mogli bili prikladniji za određeni problem. Većinom se odnose na prostornu i vremensku složenost rješenja, no mogu biti vezane i za sam programski jezik, kao i implementaciju.

## 7.4. Razlozi uporabe uzoraka

Pretpostavimo da tim programera raspravlja o nekoj aplikaciji koju moraju realizirati. U nekom dijelu raspravlja se o obilasku i ispisu elemenata unutar svih sadržanih kolekcija.



Jedan od opisa potencijalog rješenja bi mogao uključivati nešto poput: `foreach` petljom treba učitati svaki od postojećih čvorova, zatim uzeti prvi element čvora pa drugi. U novoj `foreach` ili `while` petlji učitavati tablice raspršenog adresiranja koje su elementi skupa te pomoću iteratora svaki njezin element ispisati u konzoli. Drugi element čvora bi se mogao sastojati od polja tablica raspršenog adresiranja koje bi mogle čuvati bilo kakve vrijednosti.

Sasvim je jasno da se u ovako nepreciznom i neprikladnom opisu i više nego lako izgubiti. Alternativa gornjem pristupu rješavanja problema obilaska kompleksne strukture jest: koristiti ćemo Visitor uzorak.

Važniji razlozi uporabe uzoraka su  **smanjenje kompleksnosti i povećanje razumljivosti**  rada sustava. Pomoću apstraktnijeg opisa lakše je dočarati kako je sustav koncipiran, te zavisnosti njegovih sastavnih dijelova.

Iskustvo stečeno kroz godine programiranja i rezultati različitih metoda smanjuju broj čestih grešaka na minimum. Već unaprijed možemo procijeniti koje rješenje je najisplativije za problem koji je pred nama i samo slijedimo dobro definiran postupak rješavanja istog, čime  **skraćujemo i sam proces razvoja** . Poznavanje i razumijevanje određenog broja uzoraka nam daje  **mogućnost bržeg snalaženja**  u situacijama kad iskrsne neki novi problem, za koji ne postoji specifičan uzorak kojeg možemo upotrijebiti. U takvim situacijama na raspolaganju su nam dijelovi postojećih uzoraka koje možemo iskoristiti kako bismo osmislili novo, dovoljno jednostavno i učinkovito rješenje.

Uzorci, između ostalog, uvelike  **olakšavaju rad** . Rasprave o aplikacijama, sustavima i procesima u njima se odvijaju kroz prizmu uzoraka, te su na višoj razini od rasprava o jednostavnim uvjetima i elementima poput `for`-petlji i tablica raspršenog adresiranja.

## 7.5. Klasifikacija i dokumentacija uzoraka

Uzorci se klasificiraju na temelju problema kojeg rješavaju, a organiziraju se u skupine povezanih uzoraka na temelju postojećih kriterija. Dokumentacija uzorka treba sadržavati sve bitne informacije o problemu kojeg uzorak rješava, kontekstu u kojem se koristi i samom rješenju.

### 7.5.1. Klasifikacija uzoraka

Dva kriterija koja se koriste za klasifikaciju su *domet djelovanja* i *namjena*.

*Domest djelovanja* definira da li se uzorak prvenstveno odnosi na razrede ili na objekte.

*Uzorci razreda* se bave odnosima razreda i podrazreda. Ti odnosi su utvrđeni kroz nasljeđivanje i statični su.

*Uzorci objekata* se bave odnosima među objektima i dinamički su. Ovi uzorci su brojniji.

*Namjena* kod klasifikacije govori upravo to – koja je namjena uzorka, tj. što on točno radi. Osnovna podjela uzoraka po namjeni jest: *tvorbeni* (engl. *creational*), *strukturni* (engl. *structural*) i *ponašajni* (engl. *behavioral*). No pored njih se u literaturi još mogu pronaći i sljedeće vrste: *arhitekturni* (engl. *architectural*), *temeljni* (engl. *fundamental*) i *istovremeni* (engl. *concurrency*).

#### **7.5.1.1. Tvorbeni uzorci**

Bave se mehanizmima stvaranja objekata:

Factory Method – definiranje sučelja za stvaranje objekata, s mogućnošću prosljeđivanja samog stvaranja podrazredima;

Abstract Factory – definiranje sučelja za stvaranje objekata, bez definiranja kojem razredu će ti objekti pripadati;

Singleton – osigurati da neki razred može imati samo jedan primjerak, te omogućiti pristup tom primjerku preko neke globalne točke;

Prototype – definira se primjerak prototipa koji se koristi za stvaranje novih objekata kopiranjem;

Builder – rastavljanje stvaranja složenog objekta od njegovog prikaza, s ciljem omogućavanja različitih prikaza.

#### **7.5.1.2. Strukturni uzorci**

Bave se slaganjem objekata i razreda u veće strukture:

Adapter – konvertiranje sučelja u oblik prihvatljiv sa strane nekog klijenta;

Bridge – razdvajanje apstrakcije od implementacije kako bi se omogućilo njihovo nezavisno mijenjanje;

Composite – organizacija objekata u stablovite strukture radi predstavljanja polovično definiranih hijerarhija;

Decorator – dinamičko dodjeljivanje dodatnih odgovornosti nekom objektu (zamjena za ostvarivanje dodatne funkcionalnosti kreiranjem podrazreda);

Façade – definiranje nadsučelja nekom skupu sučelja podsustava radi olakšanog korištenja podsustava;

Flyweight – korištenje dijeljenja objekata da bi se osigurala efikasna podrška za velik broj manjih objekata (zbog visoke cijene pohrane istih);

Proxy – definiranje sučelja radi kontrole pristupa nekom drugom objektu ili sustavu.

### 7.5.1.3. Ponašajni uzorci

Bave se algoritmima, dodjelom odgovornosti objektima i njihovom međusobnom komunikacijom:

Chain of responsibility – razdvajanje pošiljatelja zahtjeva od primatelja, kako bi više objekata moglo obraditi sam zahtjev;

Command – prikaz zahtjeva kao objekata da bi se omogućila parametrizacija klijenata i osigurala podrška za nepodržane akcije;

Interpreter – opis definiranja gramatike za neki jednostavan jezik, reprezentacije rečenica u njemu i njihove interpretacije.

Iterator – omogućiti slijedan pristup skupu elemenata bez otkrivanja njegovog unutarnjeg prikaza;

Mediator – definiranje objekta koji sadrži znanje o načinu funkcioniranja skupa razreda, metodama koje svaki razred posjeduje, te pružanje posredništva u prosljeđivanju zahtjeva među njima;

Memento – mogućnost vraćanja prvotnog stanja nekog objekta;

Observer – motrenje stanja nekog objekta i obavješćavanje zavisnih objekata o promjeni stanja kad se dogodi;

State – promjenom unutarnjeg stanja nekog objekta može se promijeniti i njegovo ponašanje, a djelomično i njegov tip;

Strategy – koristan kod dinamičkog mijenjanja algoritama u aplikacijama;

Template method – prosljeđivanje definiranja nekih koraka algoritma podrazredima, bez mijenjanja same strukture algoritma;

Visitor – odvajanje algoritma od strukture objekta kako dodavanje novih operacija ne bi utjecalo na strukturu objekta.

### 7.5.2. Dokumentacija

Dokumentacija ili opis uzorka koristi dobro definirani format koji uključuje cijeli proces dizajna, odlučivanja, drugih razmatranih mogućnosti i kompromisa, kao i krajnji rezultat sa primjerima uporabe.

Svaki uzorak sastoji se od sljedećih elemenata:

1. ime uzorka i klasifikacija; svaki uzorak treba imati jedinstveno i opisno ime zbog lakšeg referenciranja na njega i prepoznavanja prilika za njegovu uporabu.
2. namjena; kratki opis što uzorak radi
3. poznat kao; uzorak može imati više imena, te ako ih ima treba ih dokumentirati
4. motivacija; scenarij u kojem se pojavljuje problem i kontekst u kojem uzorak može biti korišten
5. primjenjivost; popis situacija u kojima je uzorak koristan



6. struktura; grafički prikaz uzorka (dijagrami razreda i interakcija)
7. sudionici; lista razreda i objekata uzorka te njihova uloga
8. suradnja; opisuje na koji način sudionici uzorka međusobno komuniciraju
9. posljedice; rezultati uporabe ovog uzorka (kompromisi, cijene...)
10. implementacija; opis implementacije (tehnike i savjeti), te prikaz rješenja
11. primjer koda; ilustracija primjene uzorka u nekom programskom jeziku
12. poznati korisnici; primjeri uporabe uzorka u nekim postojećim sustavima
13. srodni uzorci; popis uzoraka koji su na neki način povezani, koji mogu biti korišteni uz ili umjesto trenutnog, te postojeće razlike

Postoje tri često korištena prikaza dijagrama za ilustraciju uzoraka: dijagram razreda (struktura i odnos razreda), dijagram objekata (struktura objekata) i dijagram međudjelovanja (tok zahtjeva među objektima). Služe kao dodatna ilustracija uzoraka, a većina dokumentacija uzoraka sadrži barem jedan od tih dijagrama.

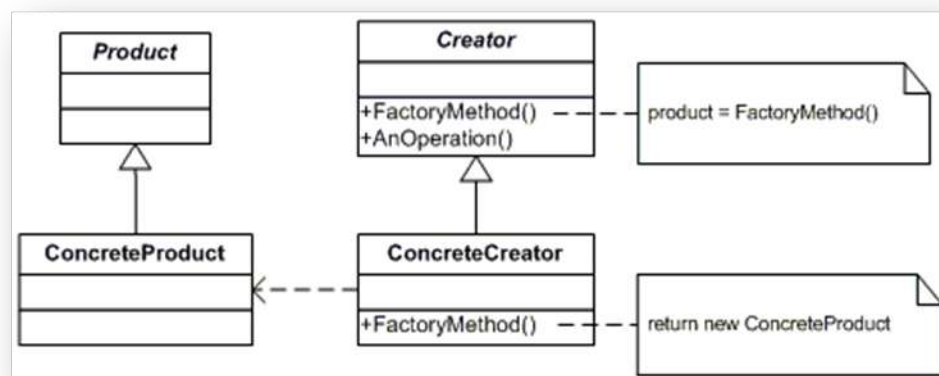
## 7.6. Primjena uzoraka

U nastavku je malo opširniji opis nekolicine uzoraka, te jednostavnija implementacija istih u programskom jeziku C#.

### 7.6.1. Factory Method

Po namjeni i dometu, Factory Method je tvorbeni uzorak razreda.

Struktura uzorka je prikazana sljedećim UML dijagramom razreda:



Ovaj uzorak čine:

Product - definira funkcionalnost preko sučelja objekata (ili apstraktnog razreda) koje factory method stvara.

Concrete Product - implementacija Product sučelja

Creator - deklarira factory method.

ConcreteCreator – reimplementacija factory method kako bi vraćala primjerak ConcreteProduct.

Zamislamo da nam je potreban program koji treba raditi s više različitih tipova slika, za svaki od kojih zna prepoznati format i stvoriti prikladan čitač.

```
using System;
namespace Primjeri.Uzorci.FactoryMethodPrimjer {

// Product
abstract class Image{ ... }

//ConcreteProduct
class BMPImage : Image { ... }
//ConcreteProduct
class JPGImage : Image { ... }
//ConcreteProduct
class PNGImage : Image { ... }
//ConcreteProduct
class GIFImage : Image { ... }

// Creator
abstract class ImageLoader {
    Image loadImage(string fileName);
}

//ConcreteCreator
class JPGImageLoader : ImageLoader{
    public override Image loadImage(string
        fileName) {
        //iz predanog imena datoteke učitava se jpg
        Image i = internalLoadJPG(fileName);
        return i;
    }
    internalLoadJPG(string fileName) {
        //zna učitati jpg iz datoteke
    }
}

//ConcreteCreator
class BMPImageLoader : ImageLoader{
    public override Image loadImage(string
        fileName) {
        //iz predanog imena datoteke učitava se bmp
        Image i = internalLoadBMP(fileName);
        return i;
    }
    internalLoadBMP(string fileName) {
        //zna učitati bmp iz datoteke
    }
}

//ConcreteCreator
class PNGImageLoader : ImageLoader{
    public override Image loadImage(string
        fileName) {
        //iz predanog imena datoteke učitava se png
```

```

        Image i = internalLoadPNG(fileName);
        return i;
    }
    internalLoadPNG(string fileName) {
        //zna učitati png iz datoteke
    }
}

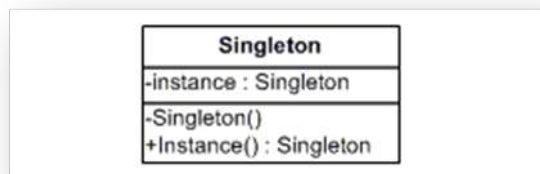
public class ImageLoaderFactory {
    public static ImageLoader getImageLoader(string
fileName) {
        int imageType = getImageType(filename);
        switch(imageType) {
            case ImageLoaderFactory.BMP:
                return new BMPLoader(fileName);
            case ImageLoaderFactory.JPG:
                return new JPGLoader(fileName);
            // itd.
        }
    }
}
}

```

### 7.6.2. Singleton

Po namjeni i dometu, Singleton je tvorbeni uzorak objekata.

Struktura uzorka prikazana je sljedećim UML diijagramom:



Ovaj uzorak čini samo operacija koja dopušta pristup primjerku razreda s time da se propisuje stvaranje samo jednog primjerka. Potreba za ovim se javlja u kompleksnim sustavima gdje nekim dijelom sustava može (smije) upravljati samo jedan specijaliziran objekt. Singletonom bismo mogli predstaviti razred koji kontinuirano snima rad nekog web sustava. Svako pristupanje sustavu se bilježi, kao i pojedinačni zahtjevi korisnika. Sam sustav može imati nekoliko specijaliziranih modula, svaki od kojih također treba bilježiti svoju aktivnost. Ako ne postoji instanca loga u trenutku kada prvi modul treba zabilježiti neku aktivnost, on će ju stvoriti. Svaki sljedeći modul ne smije stvoriti novu instancu već treba preuzeti postojeću instancu kako bi svi podaci bili na jednom mjestu i ne bi došlo do nekonzistentnosti ili gubitka podataka.

Način na koji se Singleton ostvaruje jest jednostavan: definira se neki razred s metodom odgovornom za stvaranje primjerka tog razreda. Ta metoda će stvoriti novi objekt ukoliko on već ne postoji, u protivnom će vratiti postojeći objekt.

Zamislamo da postoji objekt koji logira spajanje na neki sustav – za svako pristupanje sustavu dodaje se novi zapis na kraj datoteke. Singleton uzorak ćemo ovdje iskoristiti kako bismo bili sigurni da se ne pokuša više zapisa istovremeno dodati na kraj datoteke.

```
using System;
namespace Primjeri.Uzorci.SingletonPrimjer {
    public class LogSingleton {
        private static LogSingleton Instanca = null;
        private static string Log = null;
        protected LogSingleton() {} // otvori datoteku

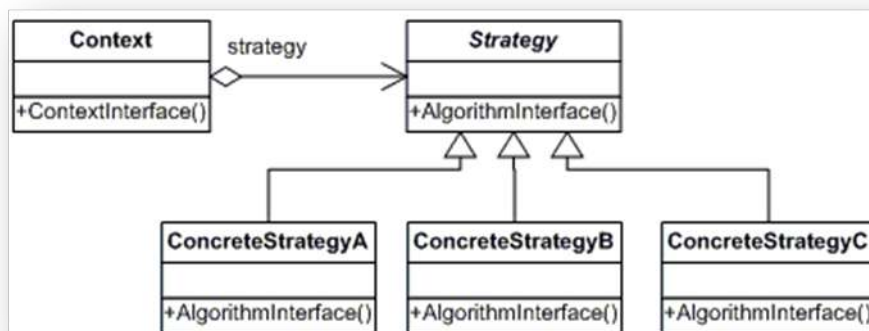
        public static LogSingleton getInstanca() {
            if(Instanca == null) {
                Instanca = new LogSingleton();
            }
            return Instanca;
        }

        public static void Log(string logAppend) {
            if(log==null) log = logAppend; //ili zapiši u
            datoteku
            else log = log+logAppend;
        }
    }
}
```

### 7.6.3. Strategy

Po namjeni i dometu, Strategy je ponašajni uzorak razreda.

Struktura uzorka prikazana je sljedećim UML dijagramom:



Ovaj uzorak se koristi za omogućavanje dinamičkog mijenjanja algoritama. To se postiže učahurivanjem algoritama u objekte. Uzorak, ovisno o uvjetima izvođenja a neovisno o korisnicima, omogućava izmjenjivanje algoritama.

Sučelje Strategy na gornjoj slici implementiraju konkretni razredi (ConcreteStrategyA, B i C) koji će imati svoje, prilagođene implementacije metode AlgorithmInterface(). Razred Context ćemo konfigurirati objektom neke konkretne implementacije.

Kao primjer ćemo uzeti razred koji ima kolekciju nekih elemenata, a implementacije sučelja Strategy će sadržavati različite metode sortiranja kolekcija.

```
using System;
using System.Collections;
namespace Primjeri.Uzorci.StrategyPrimjer{
    abstract class SortStrategy {
        public abstract void Sort(ArrayList lista);
    }

    //razred koji koristi QuickSort algoritam
    class QuickSort : SortStrategy {
        public override void Sort(ArrayList lista) {
            list.Sort(); //implementacija QuickSorta
        }
    }

    class ShellSort : SortStrategy {
        public override void Sort(ArrayList lista) {
            list.Sort(); // implementacija ShellSorta
        }
    }

    class MergeSort : SortStrategy {
        public override void Sort(ArrayList lista) {
            list.Sort(); // implementacija MergeSorta
        }
    }

    class SortiranaLista {
        private ArrayList ListaImena = new ArrayList();
        private SortStrategy Strategy;

        public void SetSortStrategy(SortStrategy st) {
            this.Strategy = st;
        }

        public void Add(string ime) {
            ListaImena.add(ime);
        }

        public void Sort() {
            strategy.Sort(ListaImena);
            foreach(string ime in ListaImena) {
```

```

        Console.WriteLine(""+ime);
    }
    Console.WriteLine();
}
}
}

```

## 7.7. Kako odabrati i koristiti uzorak dizajna

Postoji nekoliko pristupa pronalaženju idealnog uzorka za neki problem.

Najlogičnije je započeti proučavanjem namjene uzoraka. Klasifikacija uzoraka može suziti izbor a nakon toga detaljnim proučavanjem namjena razmotraju se svi uzorci koji «zvuče» primjenjivo za naš problem.

Također, razmatranje odnosa može nas uputiti na neki specifičan uzorak ili možda grupu uzoraka. Grupiranje uzoraka po sličnosti namjene (tvorbeni, strukturni, ponašajni), pregled njihovih općih značajki, te međusobnih razlika namjena može pomoći pri eliminaciji neupotrebljivih uzoraka. Predviđanjem mogućih uzroka redizajna i problema koji tim povodom mogu nastati moguće je odabrati uzorke koji omogućavaju zaobilaženje tih problema. Suprotno prethodnom, mogu se razmotriti elementi koje bismo htjeli mijenjati bez redizajna. Fokusiranjem na njih već u startu imamo veći izbor uzoraka koji podržavaju mijenjanje nekih svojih aspekata, bez redizajna.

Nakon što smo odabrali uzorak, kako ga iskoristiti?

Prvo se preporuča proučiti uzorak radi općenitog pregleda, a zatim se posvetiti dijelu koji opisuje primjenjivost i posljedice, kako bismo bili sigurni da smo odabrali dobar uzorak.

Slijedi proučavanje strukture, sudionika i suradnje kako bismo dobili jasnu sliku razreda i objekata uzorka, te odnosa u kojem se nalaze. Primjer koda nam može poslužiti kao dobra ilustracija implementacije uzorka.

Sljedeći korak jest dati sudionicima imena smisljena za specifičan kontekst, kako bismo smanjili razinu apstrakcije i približili se konkretnoj implementaciji. Nakon toga implementiramo konkretne razrede i nasljeđivanja, sučelja, te njihovu međusobnu zavisnost. Definiramo imena operacija u uzorku, specifična za aplikaciju, te koristimo odgovornosti i suradnje kao vodič kod svake operacije koju izdvojimo. Na kraju slijedi implementacija operacija kako bi se izvršavale propisane odgovornosti. Tu nam mogu biti korisni savjeti iz dijela implementacije i sami primjeri koda.

Posljedice koje neki uzorak nosi nam najbolje mogu pomoći pri procjeni koristi i ograničenja korištenja tog uzorka. Važno je još

napomenuti da uzorke treba koristiti samo onda kada smo sigurni da je fleksibilnost koju nude u našem slučaju doista i potrebna.

Kroz iskustvo se stvara individualan način rada s uzorcima dizajna, te gore navedeno treba shvatiti samo kao općenitu uputu za jednostavniji i manje problematičan početak.

### **7.8. Antiuzorci dizajna**

Antiuzorci dizajna (engl design antipatterns) daju opis problematičnog stanja aplikacije, nastalog kao rezultat nedovoljnog poznavanja korištenih tehnika. Problemi mogu biti u kodu, kao i samom razvojnom procesu. Dok uzorci služe ubrzanju i olakšavanju procesa stvaranja, svrha poznavanja i razumijevanja antiuzoraka jest da spriječi daljnje korištenje provjereno loših rješenja.

### **7.9. Zanimljivi linkovi**

1. Wikipedia, URL: [http://en.wikipedia.org/wiki/Design\\_patterns](http://en.wikipedia.org/wiki/Design_patterns) , (10.1.2007.)
2. Wikipedia, URL: [http://en.wikipedia.org/wiki/Design\\_pattern\\_%28computer\\_science%29](http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29), (10.1.2007.)
3. Wikipedia, URL: <http://en.wikipedia.org/wiki/Anti-pattern> , (10.1.2007.)
4. Wikipedia, URL: [http://en.wikipedia.org/wiki/Unified\\_Modeling\\_Language](http://en.wikipedia.org/wiki/Unified_Modeling_Language), (03.02.2007.)
5. UML tutorial, URL: [http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML\\_tutorial/index.htm](http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm) (03.02.2007.)

## 8. OOP VS OO dizajn

Naučiti osnovne koncepte objektno orijentiranog programiranja nije dovoljno da bi uspješno kreirali program na objektno orijentirani način. Kao što smo naučili u poglavlju o uzorcima, pažnja koja se posveti proučavanju zavisnosti i suradnje objekata neke aplikacije ili sustava može rezultirati manjim, jednostavnijim i puno razumljivijim rješenjem, kao i pomoći pri održavanju i dokumentaciji.

Korištenje objekata i razreda nije dovoljno da bi program bio dizajniran na objektno orijentirani način. Korištenje objekata i razreda bez planiranja svodi se na proceduralno programiranje korištenjem objektno orijentiranog jezika.

Medjutim, objektno orijentirani dizajn uvodi potrebu za definiranjem (određivanjem) koncepata koji u samoj analizi ne postoje, poput tipova atributa razreda, ili aplikacijske logike metoda.

Objektno orijentirani dizajn uključuje se u kreiranje objektno orijentiranog modela.

*„Design is something you do when your brain is too small to hold the entire project.*

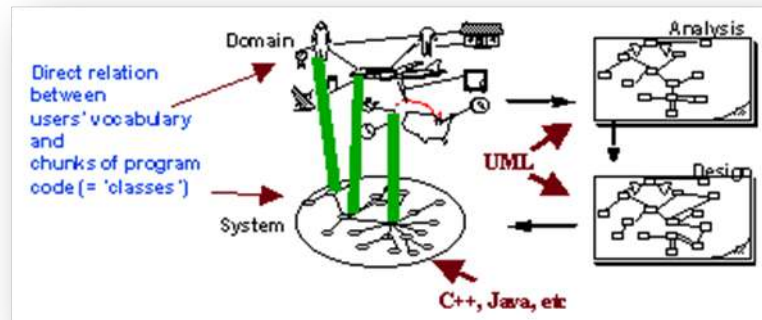
*Design is about managing complexity: a design method helps you to split big projects into manageable chunks the will fit in your brain.” - anonimus*

### 8.1 Proces dizajniranja objektno orijentiranog modela

Svaki proces dizajniranja sustava sastoji se od nekoliko koraka:

1. Lista zahtjeva koje mora ispunjavati program
2. Izbor jednog od zahtjeva
3. Razredi koji će biti potrebni za ostvarenje tog zahtjeva
4. Da li zahtjevi sugeriraju ikakvu vezu među razredima? (nacrtati dijagram razreda)
5. Kako objekti komuniciraju da bi zadovoljili zahtjeve (nacrtati dijagram objekata)
6. U dijagram razreda dodati sve aktivnosti i nove odnose među razredima
7. Probati razmišljati na različite načine (različiti razredi, različite interakcije među objektima) da bi zadovoljili zahtjeve. Prva ideja najčešće nije i najbolja ideja
8. Da li odluke koje ste donijeli imaju odjek na prijašnje dijelove dizajna?
9. Izbor sljedećeg zahtjeva ...





Uzmimo na primjer da radimo program za srednje poduzeće kojemu moramo isprogramirati aplikaciju za organiziranje vremena ,kontrolu zaposlenika, kontrolu trenutnog stanja poduzeća, planova za budućnost, ispisa novih projekata.... Poduzeće se bavi prodajom informatičke opreme. Sektori u poduzeću su:

1. Marketing
2. Management
3. IT – aplikativni, tehnički i sistemski dio
4. Planiranje i razvoj
5. Računovodstvo
6. Odjel za odnos s javnošću
7. Odjel za upravljanje ljudskim potencijalom (testovi osobnosti,

Po dolasku u tvrtku obavili ste razgovore s ljudima iz svakog sektora i saznali ste njihove zahtjeve koji su se prilično razlikovali, ali su imali i neke sličnosti. Nakon toga ste razgovarali s upraviteljima pojedinih sektora i dogovorili ste se o razinama ovlasti, te ste napravili analizu.

| Položaj            | Potrebe  |
|--------------------|--|
| <b>Manager</b>     | Kontrola zaposlenika svojeg sektora<br>skrivanje vlastitog adresara od ostalih zaposlenika                         |
|                    | dostupnost adresara i kalendara managerima ostalih sektora   |
|                    | dnevne zadaće zaposlenika dobiva na e-mail   |
|                    | dijagramsko praćenje svakog zaposlenika  |
|                    | dostupan kalendar zaposlenicima sektorima za prijavu sastanaka ali nemogućnost pregleda ostalih sastanaka i obveza |
| <b>Zaposlenici</b> | nemogućnost gledanja tuđih dnevnih obaveza   |
|                    | zajednički adresar poslovnih partnera  |
|                    | privatni adresari nedostupni ostalim zaposlenicima   |
|                    | prijavljuju se za termin sastanaka menageru i ostalim zaposlenicima  |
|                    | prijava dolaska na posao   |
|                    | dnevni izvještaj manageru šalju mejlom   |

Zajednička svojstva koja imaju manageri i zaposlenici svih sektora su:

| Sektor                             | Potrebe   |
|------------------------------------|---|
| <b>Marketing</b>                   | aplikaciju za konferencijske razgovore s poslovnim partnerima i zaposlenicima             |
|                                    | bazu ideja za reklame   |
| <b>Menadžment</b>                  | ispis najuspješnijih zaposlenika  |
|                                    | raspodjela vremena po projektima  |
|                                    | mogućnost koordiniranja rada ostalih djela  |
|                                    | mogućnost pregleda uspješnosti pojedinog zaposlenika                                      |
| <b>IT</b>                          | pregled troškova  |
|                                    | pristup podataka sektora računovodstva i planiranja i razvoja                             |
|                                    | pregled dobiti  |
|                                    | sigurnosni problem  |
|                                    | održavanje strjeva na kojima drugi rade   |
|                                    | prijava kvarova i bugova u program  |
|                                    | pregled timova i zadataka pri izradi sofvera  |
|                                    | deadline za izradu projekta   |
|                                    | projekti za daljnje širenje poduzeća  |
|                                    | studije prethodne uspješnosti   |
| <b>računovodstvo</b>               | popis radnika i dostupnost podataka svih radnika o radnim satima, bolovanjima, godišnjima |
|                                    | osobni podaci o radnicima zadi poreznih olakšica  |
|                                    | adresar svih radnika  |
|                                    | kalendar dolazaka revizora  |
|                                    | dijagrami plaćenog poreza   |
|                                    | popis potraživanja i dugovanja  |
| <b>Odjel za odnose s javnošću</b>  | informacije o poslovanju tvrtke   |
|                                    | dijagrami uspješnosti   |
|                                    | idje o novim projektima   |
|                                    | prezentacija novih proizvoda  |
| <b>Odjel za ljudski potencijal</b> | potrebe poduzeća za pojedinom strukom zaposlenika   |
|                                    | testovi inteligencije, sposobnosti, znanja  |
|                                    | statistika dosadašnjih zaposlenih ljudi i intervjua                                       |
|                                    | primanje i spremanje CV-a mogućih zaposlenika   |

Dakle ovdje već možete zaključiti da ćemo morati imati definiran korisnike s ovlastima (manageri i zaposlenici), osim toga u pojedinom sektoru svaki zaposlenici u različitim sektorima imat će različite potrebe za korištenjem vašeg sustava kontrole obaveza i rada poduzeća. Potrebno je razmisliti kako realizirati svaki od zahtjeva prema koracima koji su prethodno navedeni.

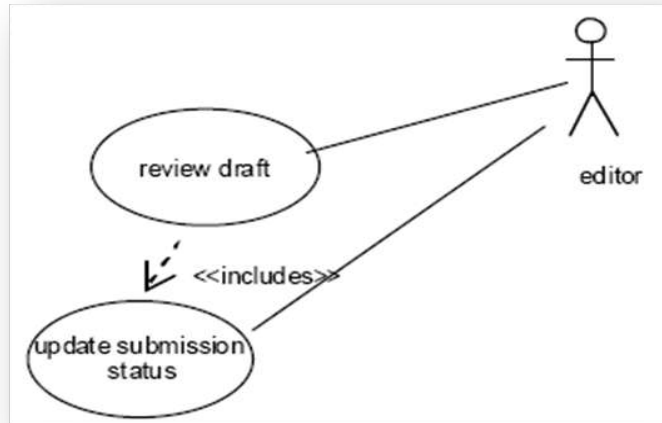
Svaki od sektora ima posebne potrebe:

Kako je prikazano u tablici, vidimo da svaki sektor ima neke svoje posebne zahtjeve ali da bi tvrtka napredovala potrebno je da sektori surađuju. Na temelju toga možemo zaključiti da sve zahtjeve koji su im zajednički objedinjenimo u pojedine cjeline koje će komunicirati. Tako će se osobni podaci o zaposlenicima nalaziti na jednom mjestu, a definirat ćemo tko će im imati pristup i na koji način (tko će moći čitati podatke, tko će ih moći mijenjati, te na koji način će se to obavljati).

Ovdje je prilično zahtjevno( no i ne nemoguće)korisiti proceduralnu programsku paradigmu, nego je potrebno koristi objektno orijentiranu paradigmu, ali niti tada ne smijemo odmah početi programirati. Prije samog programiranja potrebno je napraviti razgovore s korisnicima usluga o njihovim željama, zatim to podijeliti u smislene cjeline, odrediti odnose među cjelinama, zatim svaku cjelinu podijeliti na još manje cjeline.

Dakle, ovdje je sigurno da će jedna od aplikacija biti kalendar obveza, međutim svatko će taj kalendar koristiti na način koji je njemu potreban (knjigovođa će na kalendaru moći vidjeti datume za obračun poreza, datume kada treba proknjižiti plaću, IT službenik kada mu je rok za predaju projekta, itd.).

Sam postupak kreiranja aplikacije na objektno orijentiran način vrlo je složen proces



Kao što ste mogli vidjeti, svakoj aplikaciji koja će biti dizajnirana na objektno orijentiran način treba pristupiti prvenstveno na istraživački način, određujući zahtjeve koje ta aplikacija treba obrađivati. Isto tako potrebno je proučiti tržište kojem je namijenjena te aplikaciju izvesti što učinkovitije kako bi rukovanje i održavanje bilo jednostavno i prilagođeno krajnjem korisniku.

Ostatak razrade problema i programska implementacija ostavljeni su kreativnosti studenata ovog kolegija koje ovo područje dodatno zanima. Također oni koje to zanima mogu proučiti linkove o **UML**-u.

## 8.2 Zanimljivi linkovi

1. OOD,URL:  
<http://www.sei.cmu.edu/str/descriptions/oodesign.html> (5.2.2007.)
2. OOP,  
URL:[http://www.accu.org/acornsig/public/articles/ood\\_intro.html](http://www.accu.org/acornsig/public/articles/ood_intro.html)  
(7.2.2007)
3. Object, URL: <http://www.well.com/user/ritchie/oo.html>  
(7.2.2007)
4. UML : [http://en.wikipedia.org/wiki/Unified\\_Modeling\\_Language](http://en.wikipedia.org/wiki/Unified_Modeling_Language)  
(24.3.2007.)
5. Članci o OOD i OOP :  
[http://www.eventhelix.com/RealtimeMantra/Object\\_Oriented/](http://www.eventhelix.com/RealtimeMantra/Object_Oriented/)  
(24.3.2007)
6. Tipovi OOD: <http://ootips.org/> (24.3.2007)
7. Sve o OOP-u : <http://www.well.com/user/ritchie/oo.html>  
(24.3.2007)

## 9.BIBLIOGRAFIJA

1. Jesse Liberty, Programming C#, 2nd Edition, O'Reilly, 2002
2. Anders Hejlsberg and Scott Wiltamuth, C# Language Reference.doc, 2000.
3. Auditorne vježbe iz kolegija Programske paradigme i jezici, Zavod za primjenjeno računarstvo, Fakultet elektrotehnike i računarstva, 2006/2007.
4. Predavanja iz kolegija Objektno-orijentirano programiranje, Zavod za primjenjeno računarstvo, Fakultet elektrotehnike i računarstva, 2005/2006.
5. Microsoft, C# Language Specifications, 2001
6. J.P. Hamilton, Object-Oriented Programmingwith Visual Basic .NET, O'Reilly
7. Robin A. Reynolds-Haertle ,OOP with MS Visual Basic .NET & C# Step by Step, Microsoft Press, 2002.
8. Eric Gunnerson, A Programmer's Introduction to C#, A press, 2000.
9. Edward V. Berard, Abstraction, Encapsulation, and Information Hiding, URL:<http://www.toa.com/pub/abstraction.txt>
10. Edward V. Berard, Basic Object-Oriented Concepts, URL:<http://www.toa.com/pub/OOBasics.pdf>
11. J. Sharp, J.Jagger, Visual C#.NET, step by step,Microsoft Press 2002.
12. B. Motik, J. Šribar, Demistificirani C++, Element, Zagreb, 199
13. Erich Gamma... [et al.], Design Patterns, Elements of Reusable Object-Oriented Software, Oxford University Press, 1994.