

RAZVOJ PROGRAMSKIH RJEŠENJA

Sadržaj

Poglavlje 1: Razvoj programskih rješenja: pristupi, jezici, alati.....	7
1.1. Uvod u softver inženjeringu	7
1.2. Razvojne aktivnosti softverskog sistema	9
1.2.1. Metodologije razvoja softvera	11
1.3. Programske paradigme	13
1.3.1. Histrorijski pregled programskih jezika.....	16
1.3.2. C# i .NET platforma	18
Poglavlje 2: Osnovni programski koncepti u C# jeziku.....	20
2.1. Uvod u okruženje	20
2.2. Varijable i tipovi podataka	21
2.2.1 Identifikatori	21
2.2.2. Varijable	21
2.2.3. Vrijednosni tipovi.....	22
2.2.4. Referencni tipovi.....	27
2.2.5. Nizovi.....	30
2.2.6. Konstante.....	32
2.3. Iskazi selekcije	32
2.3.1. if iskaz.....	32
2.3.2. switch iskaz	33
2.4. Kontrolne petlje	34
2.4.1. While i do-while petlja.....	34
2.4.2. for petlja	35
2.4.3. foreach petlja	35
2.4.4. Iskazi promjene toka izvršavanja programskog koda	36
2.5. Metode.....	37
Poglavlje 3: Osnovni koncepti klase i dekompozicija programa	41
3.1. Klasa: osnovni koncepti i izražavanje u C#-u.....	41
3.1.1. Definicija klase	41
3.1.2. Osobine (<i>property</i>) klase	47

RAZVOJ PROGRAMSKIH RJEŠENJA

3.1.3. Indekser.....	49
3.1.4. Dodatne osobine članova klase za upravljanje privilegijama	50
3.2. Dekompozicija	53
3.2.1. Dekompozicija projekta u C#-u	53
3.2.2. Smještanje klase u biblioteku	55
3.2.3. Kompozicija klasa i dekompozicija programa	56
3.2.4. Modifikatori pristupa klasi	59
Poglavlje 4: Napredni objektno-orientirani koncepti	63
4.1. Nasljeđivanje.....	63
4.2. Polimorfizam	65
4.2.1. Apstraktne klase i metode	68
4.3. Kreiranje i korištenje interfejsa.....	69
4.4. Delegati	75
4.4.1. Anonimna metoda	77
4.4.2. Lambda izraz	78
4.5. Generički koncepti	80
Poglavlje 5: Grafički korisnički interfejs (GUI) i uvod u programiranje vođeno događajima	87
5.1. Osnove i značaj korisničkog interfejsa	87
5.1.1. Istorija grafičkog korisničkog interfejsa	88
5.1.2. Kratka istorija dizajniranja ekrana	90
5.2. GUI kontrola: Windows (prozor) – osnovne karakteristike	92
5.2.1. Karakteristike prozora.....	92
5.2.2. Stilovi prezentacije prozora	93
5.3. Windows Forme i dugme kontrola	96
5.3.1. Kreiranje forme iz konzolne aplikacije	96
5.3.2. Kreiranje formi sa dizajnerom	97
5.3.3. GUI kontrole	102
5.3.4. Dugme (<i>button</i>) kontrola	102
5.4. Model upravljanje događajima (Event Handling Model)	106
5.4.1. Programiranje događaja za dugme kontrolu	106
5.4.2. Kreiranje događaja sa dizajnerom	108

RAZVOJ PROGRAMSKIH RJEŠENJA

Poglavlje 6: Pregled GUI kontrola	111
6.1. GUI kontrole.....	111
Slijedi prikaz labele, tekst polja, kontrola za izbor jedne ili više opcija i kontrola za grupiranje.	111
6.1.1.Labela.....	111
6.1.2. TextBox kontrola	112
6.1.3. CheckBox i RadioButton kontrola.....	115
6.1.4.Formiranje grupe kontrola – GroupBox i Panel	119
6.2. Interakcija sa korisnikom	120
6.2.1. MessageBox.Show() metoda	120
6.2.2. Interakcija korisnika sa tastaturom i mišem	124
Poglavlje 7: Pregled dodatnih GUI kontrola.....	128
7.1. Kontrola za prikazivanje slike PictureBox	128
7.2. Izbor opcija: <i>List</i> i <i>Combo box</i> kontrole	130
7 . 3 .NumericUpDown kontrola	137
7.4.MonthCalendar kontrola	138
7.5. LinkLabel kontrola.....	139
7.6. TabControl kontrola.....	141
Predavanje 8: GUI kontrola – Meni i MDI aplikacije	146
8.1. Osnovni koncepti vezani za GUI kontrolu meni	146
8.1.1.Dizajner i meni	147
8.1.2. Dinamičko dodavanje menija na formu	150
8.1.3.Kontekstni meniji	152
8.1.4. Prozor meni.....	154
8.2.MDI (Multiple Document Interface) aplikacije	155
Poglavlje 9: Dizajniranje korisničkog interfejsa.....	161
9.1. Uloga i značaj dobro dizajniranog korisničkog interfejsa	161
9.2. Utjecaj ljudske psihologije na dizajn korisničkog interfejsa.....	161
9.2.1. Percepcija	162
9.2.2.Gestalt principi.....	162
9.2.3. Pamćenje.....	164
9.3. Smjernice za izbor i raspoređivanje GUI kontrola.....	165

RAZVOJ PROGRAMSKIH RJEŠENJA

9.3.1. Izbor kontrola.....	165
9.3.2. Pravilno raspoređivanje kontrola.....	168
9.4. Odabir boja i ikona	169
9.4.1. Biranje boja	169
9.4.2 Smjernice za korištenje ikona	171
9.5. Prikaz informacija za čitanje	172
9.5. Odziv sistema i pomoć korisniku.....	173
9.5.1. Odgovor sistema na korisničke akcije	173
9.5.2. Pomoć korisniku.....	174
9.6. Testiranje upotrebljivosti sistema.....	176
Poglavlje 10: Validacija podataka i upravljanje izuzecima	178
10.1. Validacija podataka.....	178
10.1.1. Validacija ulaznih podataka sa jedne kontrole	178
10.1.2. Događaji validiranja	180
10.1.3. ErrorProvider kontrola.....	182
10.1.4. StatusStrip kontrola	185
10.1.5. Unakrsna validacija	187
10.2. Upravljanje izuzecima (<i>exception handling</i>)	188
10.2.1. Upravljanje izuzecima	188
10.2.2. Klase za upravljanje izuzecima	188
10.2.3. Try/catch/finally struktura za upravljanje izuzecima	190
10.2.4. Eksplicitno pozivanje izuzetaka.....	193
Poglavlje 11: Rad sa datotekama i direktorijima	202
11.1. Uvod u datoteke i pregled klase za rad sa datotekama.....	202
11.2. I/O izuzeci.....	203
11.3. Klase za rad sa datotekama i direktorijima	205
11.3.1. File klasa	205
11.3.2. Directory klasa za rad sa direktorijima.....	206
11.3.3. Primjer upotrebe klasa za rad sa datotekama i direktorijima.....	207
11.4. Rad sa generičkim klasama za ulaz i izlaz.....	209
11.4.1. FileStream objekat.....	210

RAZVOJ PROGRAMSKIH RJEŠENJA

11.4.2. StreamReader i StreamWriter klase	214
11.4.3. Binarno pisanje i čitanje.....	218
11.5.Serijalizacija i deserijalizacija objekata	220
Poglavlje 12: Rad sa XML dokumentima.....	226
12.1.Uvod u XML.....	226
12.2. XML dokument.....	226
12.2.1.XML elementi.....	226
12.2.2.XML deklaracija.....	229
12.2.3.Validacija XML dokumenata	230
12.3.XML i .NET	231
12.3.1. XmlTextWriter klasa	232
12.3.2. XmlTextReader klasa	234
12.3.3. Prikaz XML dokumenta u DataGridView kontroli	235
12.3.4.XML DOM.....	237
12.4.XML Serijalizacija i deserijalizacija	241
12.4.1. XML serijalizacija	241
12.4.2. XML deserijalizacija.....	244
POGLAVLJE 13: Višenitnost i asinhrono programiranje	247
13.1.Uvod.....	247
13.2. Programsко управљање нитима.....	248
13.2.1. Kreiranje niti.....	248
13.2.2. Pauziranje niti	250
13.2.3. Prekidanje izvršavanja niti	252
13.2.4. Pozadinska nit	253
13.2.5. Prioritet niti	255
13.2.6. Predaja parametara niti	255
13.2.7. Sinhronizacija niti	256
13.3.Asinhroni mehanizmi	259
13.3.1. Skupina niti.....	259
13.3.2. Task	260
13.3.3. Asinhroni pristup sa async/await	262

RAZVOJ PROGRAMSKIH RJEŠENJA

13.4. GUI i asinhronne operacije.....	265
13.4.1. Korisnički interfejs i asinhroni poziv metoda	265
13.4.2. Iniciranje osnovne niti GUI kontrole	266
Reference	270
PRILOZI	272
Prilog 1: Primjer konzolne aplikacije	273
Prilog 2: Primjeri GUI kontrola i događaja	282
POPIS SLIKA	297
POPIS TABELA	298
Indeks pojmovaa	300

Poglavlje 1: Razvoj programskih rješenja: pristupi, jezici, alati

U okviru ovog poglavlja objašnjavaju se osnovni koncepti vezani za proces razvoja programskih rješenja, pojmom softver inženjeringu, daje se pregled programskih paradigmi i programskih jezika kroz tematske jedinice:

1. Uvod u softver inženjeringu
2. Razvojne aktivnosti i metodologije razvoja softvera
3. Programske paradigme i jezici
4. C# i .NET platforma

1.1. Uvod u softver inženjeringu

Programska (softverska) rješenja su kompleksna kreacija. Programska rješenja automatiziraju mnoge poslovne procese kao što su prodaja, nabava, proizvodnja, distribucija. Programska rješenja su našla veliku primjenu i u obrazovanju i nauci. Tipično, programska rješenja se razvijaju (pišu, grade) da bi izvršavali više funkcija. Naprimjer, programsko rješenje za rezervaciju karata za pozorišne predstave, između mnogih funkcionalnosti pruža funkcionalnost evidencije predstava, mogućnost kreiranja rasporeda održavanja predstava, mogućnost rezervacije karata, mogućnost prodaje karata, funkcionalnost plaćanja u gotovini i preko kreditne kartice. Evidentno je da softverska rješenja uključuju mnoge komponente pri čemu mnoge od njih su prilagođene korisničkim potrebama i veoma su kompleksne. Mnogi učesnici raznih sredina uzimaju učešća u razvoju ovih komponenti.

Nije jednostavno napisati program-softver, jer kvalitet i efikasnost softvera ovisi od više faktora (tima, tehnologija, programskih jezika), a posljedice čak i od malih programerovih propusta mogu biti velike. Razvojni inženjeri (*developeri*) postaju vremenom svjesniji te činjenice, razvijajući upotrebljiv, tržištu namijenjen softver.

Mnogi izjednačavaju softver sa programskim kodom. Međutim, programsko rješenje (softver) nije samo programski kod. Softver prema IEEE definiciji (IEEE 1991) i ISO definiciji (ISO, 1997 i ISO/IEC 9000-3) sastavljen je od četiri komponente: programskog koda, procedura, eventualne dokumentacije i podataka koji se odnose na operacije kompjuterskog sistema.

RAZVOJ PROGRAMSKIH RJEŠENJA

Sve četiri komponente su potrebne da bi se obezbijedio kvalitet razvojnog procesa softvera i kvalitet softvera kao produkta.

- Programski kod je glavni pokretač i vodič izvršavanja programa/softvera.
- Procedure su potrebne da bi se definirao redoslijed izvršavanja programskih cjelina, odredile osobe odgovorne za izvršavanje aktivnosti potrebnih za primjenu softvera, administraciju sistema i za uspostavljanje ostalih bitnih programsko-organizacijskih tokova.
- Različiti tipovi dokumentacije su potrebni za razvojne inženjere, korisnike i osoblje održavanja. Razvojna dokumentacija pospješuje efikasnu kooperaciju i koordinaciju između članova razvojnog tima i efikasan pregled i inspekciju dizajna i programskega proizvoda. Korisnička dokumentacija daje opis i način korištenja raspoloživih aplikacija. Dokumentacija za održavanje daje timu za održavanje sve potrebne informacije o kodu, strukturi i zadacima svakog softverskog modula i veoma je bitna za lociranje uzroka softverskih grešaka, izmjene i korekcije postojećeg softvera.
- Podaci koji su neophodni za djelovanje softvera uključujući između ostalog razne parametre, kodove, liste vrijednosti, podatke za testiranje, podatke za rad.

Generalno, postoje dva osnovna tipa softverskih produkata [1]:

- Generički produkti: To su samostalni (*stand-alone*) sistemi koji su proizvedeni od strane razvojnih organizacija i prodaju se slobodno na tržištu. Primjeri takvih tipova produkata uključuju softver za upravljanje bazama podataka, procesore teksta, pakete za crtanje, razne pakete za kolaboraciju.
- Prilagođeni (*customized*) produkti. To su sistemi koji su razvijeni za pojedinačnog klijenta. Primjeri ovih tipova softvera uključuju kontrolne sisteme za elektronske uređaje, sisteme pisane za podršku pojedinačnim poslovnim procesima, sisteme za kontrolu zračnog saobraćaja.

Dobar softver nije lako napisati. Sa stanovišta korisnika dobar softver je softver koji pruža sve potrebne funkcionalnosti tom korisniku i pri tome ih je jednostavno izvršavati. Sa stanovišta razvojnog tima dobar softver je dobro dizajniran, lagan za održavanje, ponovno korištenje i proširenje.

Razvoj programskih rješenja pripada oblasti softver inženjeringu. Softver inženjeringu je inženjerska disciplina, nastala početkom 70. godina prošlog vijeka, koja se tiče svih aspekata produkcije softvera od rane faze specifikacije korisničkih zahtjeva i softverskog sistema do faze njegovog održavanja tokom korištenja softverskog rješenja. U ovoj definiciji postoje dvije ključne fraze [1]:

1. Inženjeringu disciplina: Inženjeringu primjenjuje teorije, metode i alate koji najbolje odgovaraju za rješavanje problema. Inženjeri također moraju raditi sa organizacijskim i finansijskim ograničenjima i tražiti rješenje unutar ovih ograničenja.

RAZVOJ PROGRAMSKIH RJEŠENJA

2. Svi aspekti produkcije softvera: Softver inženjering se ne dotiče samo tehničkih procesa razvoja softvera već se susreće i sa aktivnostima kao što su razvojni alati, metode i teorija za podršku produkcije softvera.

Generalno, softver inženjering je usvojio sistematičan i organiziran pristup za svoj rad, što je najčešće efektivniji način da se proizvede kvalitetan softver (Sommerville).

1.2. Razvojne aktivnosti softverskog sistema

Razvojne aktivnosti softverskog sistema uključuju:

1. Određivanje zahtjeva (identificiranje potreba i zahtjeva korisnika)
2. Analiza (razvoj specifikacije sistema za hardver i softver)
3. Dizajn softvera (dizajn sistema, objekata, algoritama, korisničkog interfejsa)
4. Implementacija (pisanje koda i dokumentacije)
5. Testiranje
6. Instalacija
7. Održavanje softvera

Određivanje zahtjeva

Tokom određivanja zahtjeva klijent i predstavnici razvojnog tima definiraju svrhu sistema. Da bi se razvio funkcionalni softverski sistem, klijent mora definirati svoje zahtjeve. U mnogim slučajevima softverski sistem je dio nekog velikog sistema. Informacije o ostalim dijelovima tog sistema pomažu uspostavljanju kooperacije između timova i interfejsa razvojnih komponenti. Rezultat ove aktivnosti je opis funkcionalnosti sistema u uvjetima učesnika i opis slučajeva korištenja softverskog sistema.

Analiza

Tokom analize predstavnici razvojnog tima (analitičari) teže da proizvedu model sistema koji je tačan, kompletan, konzistentan i nedvosmislen. Razvojni tim transformira slučajeve proizvedene tokom određivanja zahtjeva u objekte – model koji potpuno opisuje sistem. Vrši se i analiza implikacija zahtjeva na formiranje modela softverskog sistema. Tokom ove aktivnosti mogu se otkriti nepravilnosti i nekonzistentnosti u slučaju modela, koje se usaglašavaju sa klijentom. Rezultat analize je model sistema obilježen atributima, operacijama i ostalim relevantnim elementima i vezama.

RAZVOJ PROGRAMSKIH RJEŠENJA

Dizajn softverskog sistema

Tokom dizajna sistema, predstavnici razvojnog tima (dizajneri) definiraju ciljeve dizajna projekta i razlažu sistem u manje podsisteme, koji mogu biti realizirani sa odvojenim timovima. U ovoj fazi najčešće se biraju strategije za izgradnju sistema, hardverske – softverske platforme na kojima će sistem raditi. Ova faza, između ostalog, obuhvata detaljnu definiciju ulaza, izlaza i procedura procesiranja, uključujući strukture podataka, baze podataka, arhitekturu sistema. Tokom ove faze radi se i kompletan model dizajna objekata sa preciznim opisima za svaki element, uzimajući u obzir i odabranu hardversku i softversku platformu. U okviru ove faze često se daje i prototip dizajna grafičkog korisničkog interfejsa. Dio knjige od poglavlja 5-9 posvećeno je dizajniranju korisničkog interfejsa.

Za specifikaciju korisničkih zahtjeva, analizu i dizajn sistema često se koristi UML (Unified Modeling Language).

Implementacija (kodiranje) softverskog sistema

Tokom implementacije programeri prevode rješenje domena modela u programski kod. Ovo uključuje implementaciju atributa i metoda svakog objekta i integraciju svih objekata tako da oni funkcioniraju kao jedan sistem. Kodiranje obuhvata i aktivnosti osiguranja kvaliteta kao što su inspekcija, testiranje jedinica (*unit tests*) i test integracije.

Testiranje

Testiranje sistema se vrši kada je faza kodiranja završena. Glavni cilj testiranja je da se otkrije što je više moguće grešaka da bi se dostigao prihvatljiv nivo kvaliteta softvera. Testiranje sistema se uobičajno vrši od strane predstavnika razvojnog tima i korisnika. Tokom testiranja provode se razne vrste testiranja. U okviru ove knjige provodi se djelimično funkcionalno testiranje, odnosno provjerava se da li sistem ispravno radi određeni zadatak (npr. da li je dobro obračunata cijena karata za grupu posjetioca).

Instalacija

Nakog što se softverski sistem odobri, sistem se instalira kao samostalni sistem ili češće kao dio (*firmware*) postojećeg informacijskog sistema. Ako novi informacijski sistem zamjenjuje postojeći sistem, mora se uraditi i proces konverzije podataka iz postojećeg softverskog sistema i pri tome osigurati da se organizacijske aktivnosti ne prekidaju tokom faze konverzije.

Održavanje softvera

Prilikom faze održavanja softvera, vrši se korekcija i nadogradnja softvera u fazi konkretne primjene softvera. Održavanje se sastoji od tri tipa servisa: korektivni - popravak softverskih pogrešaka identificiranih od strane korisnika tokom regularnog operativnog perioda; adaptivni –

RAZVOJ PROGRAMSKIH RJEŠENJA

korištenje postojećih softverskih funkcija da se ispune novi zahtjevi; perfektivni - dodavanje novih manjih mogućnosti da se unaprijede performanse softvera.

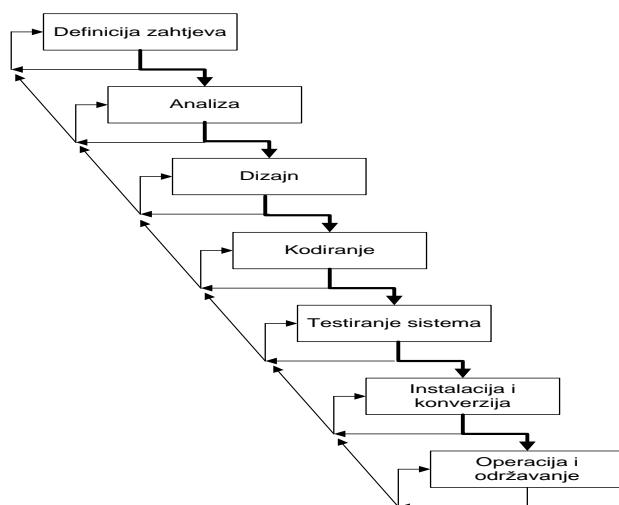
1.2.1. Metodologije razvoja softvera

Postoji više pristupa i metodologija razvoja softvera. Svi pristupi kombiniraju osnovne faze razvoja softvera i daju smjernice kako da se obavljaju. U okviru ove knjige se neće detaljno razmatrati metodologije/modeli razvoja softvera. Slijedi prikaz tri osnovna modela razvoja softvera.

- Vodopadni model
- Iterativni model
- Objektno-orientirani model

Vodopadni model

Model životnog ciklusa razvoja softvera (Software Development Life Cycle – SDLC model) je prvi, klasični model razvoja softvera koji se još uvijek primjenjuje. On obezbjeđuje najopsežniji opis raspoloživih procesa. Model prikazuje glavne gradivne blokove za čitav razvojni proces, opisan kao linearna sekvenca. Klasični model životnog ciklusa razvoja softvera (SDLC model) je linearни sekvenčnalni model koji započinje sa definiranjem zahtjeva i završava sa fazom održavanja softvera. Najstariji SDLC model je model vodopada (*waterfall model*) prikazan na slici 1.1. Model prikazan na slici 1.1 ima sedam faza:



Slika 1.1: Model vodopada

RAZVOJ PROGRAMSKIH RJEŠENJA

Broj faza može varirati s obzirom na karakteristike projekta. U složenom modelu, neke faze se dijele, dovodeći do povećanja broja faza. U manjim projektima neke faze se mogu objediniti čime se smanjuje broj faza.

Iterativni model

Iterativni model za razliku od klasičnog vodopadnog modela ne počinje sa potpunom specifikacijom odnosno definicijom zahtjeva. Umjesto toga, razvoj počinje sa specifikacijom i implementacijom samo dijela softvera. U sljedećoj iteracije se nadograđuje taj dio sistema ili se dodaje novi. Proces se ponavlja i proizvodi se nova verzije softvera na kraju svake iteracije modela. Osnovna ideja ove metode je iterativni razvoj sistema kroz ponavljajuće krugove (iteracije), pri čemu se inkrementalno dodaju manji dijelovi.

Spiralni model je napredniji iterativni pristup koji ugrađuje metode za osiguranje efektivnih performansi u svakoj od faza SDLC modela. Spiralni model u svakoj fazi integrira komentare klijenta i zahtjeve za izmjenama, analizira rizike predloženih rješenja, vrši planiranje razvoja softverskog sistema i ostale inženjerske aktivnosti.

Osnovna razlika između iterativnog i vodopadnog pristupa je način podjele projekta u manje dijelove. Vodopadni pristup dijeli projekt na osnovu aktivnosti. Jednogodišnji projekat npr. može imati dvomjesečnu fazu analize, četveromjesečnu fazu dizajna, pa tromjesečno pisanje koda, tromjesečno testiranje.

Iterativni proces dijeli projekat na dijelove po funkciji. U iterativnom procesu, prije stvarnog početka iteracija obično slijedi odgovarajuće istraživanje. Na taj način dobija se uvid u zahtjeve, dovoljan da se izvrši podjela po iteracijama.

Naprimjer, može se razvoj sistema podijeliti u 4 iteracije, pri čemu u svakoj iteraciji može se obraditi četvrtina zahtjeva i za njih obaviti sve aktivnosti životnog ciklusa razvoja softvera.

Ovaj opis je svakako pojednostavljen, ali obuhvata suštinske razlike. U praksi razvojni proces nije tako precizno podijeljen. U vodopadnom razvojnog procesu obično postoji neki oblik formalnog prijenosa između različitih faza, ali su česti i povratni tokovi što je vidljivo i na slici 1.1. U praksi se često koristi i odgovarajući hibridni procesi kao i agilne metode razvoja softvera.

Objektno-orientirani model

Objektno-orientirani model se razlikuje od ostalih modela intenzivnim korištenjem već postojećih softverskih komponenti. Prema objektno-orientiranom modelu razvojni proces počinje sa sekvencom objektno orientirane analize i dizajna. Poslije dizajn faze slijedi nabavka podesnih komponenti iz postojećih softverskih biblioteka ako su raspoložive. U suprotnom se provodi regularni razvoj softvera primjenom vodopadne, iterativne ili neke druge metode. Kopije novog razvijenog modula se zatim smještaju u softverske biblioteke za buduću upotrebu.

RAZVOJ PROGRAMSKIH RJEŠENJA

Povećanje zaliha softverskih komponenti u softverskim bibliotekama uglavnom dovodi do povećanog ponovnog korištenja (*reuse*) softvera. Ponovno iskorištenje softverskih komponenti ima više prednosti:

- Troškovi integracije već korištenih softverskih komponenti su mnogo manji nego troškovi razvoja novih komponenti.
- Korištene softverske komponente sadrže značajno manje defekata nego nove razvijene softverske komponente zbog detekcije grešaka od strane prijašnjih korisnika.
- Integracija već korištenih softverskih komponenti smanjuje vrijeme razvoja.

*U okviru ovog udžbenika koji prati gradivo predmeta **Razvoj programskih rješenja** fokus je još uvjek na jednostavnijim primjerima (manjim programskim rješenjima) koji će biti zasnovani na zadacima koji zahtijevaju analizu teksta zadatka i rješavanje problema uključujući logički dizajn na osnovu zadatka i programske kod (objektno orijentiran, sa GUI (grafički korisnički interfejs) interfejsom, datotekama, XML-om, grafikom). To će biti dobra osnova za efikasnije i kvalitetnije pisanje softvera, učenje metoda za razvoj softvera i unapređenje programskih vještina.*

1.3. Programske paradigme

Termin programska paradigma se koristi da objasni fundamentalne razlike u načinima programiranja. Osnovne programske paradigme su:

1. imperativna paradigma
2. objektno-orijentirana paradigma
3. funkcionalna paradigma
4. logička paradigma.

Imperativna paradigma

Imperativna paradigma je karakterizirana programiranjem sa stanjem i komandama koje mijenjaju to stanje. Fraza „prvo uradi ovo pa zatim ono“ odražava sekvensijalnu prirodu izvršavanja programa, odnosno prirodu izvršavanja programskih koraka po redoslijedu određenom od strane kontrolnih struktura. Osnovna ideja su naredbe koje imaju mjerljivi efekat na stanje programa. Tipične naredbe u imperativnom programiranju su dodjeljivanje, ulazno-izlazne operacije, pozivi procedura. Kada se imperativno programiranje kombinira sa manjim programskim jedinicima (procedurama), onda je riječ o proceduralnom programiranju. U oba slučaja, princip je isti: programi su upute ili naredbe za izvođenje određenih akcija. Najpoznatiji imperativni programski jezici su Fortran, Algol, Pascal, C.

Objektno-orientirana paradigma

Osnovne karakteristike objektno orijentiranog pristupa su apstrakcija, enkapsulacija, modularnost i hijerarhija.

Apstrakcija

Apstrakcija je jedan od fundamentalnih principa pomoću kojeg se ljudi bore sa kompleksnošću. Suština apstrakcije je u uočavanju osnovnih karakteristika objekta i njegovom razlikovanju od ostalih objekata na temelju uočenih osobina. Ona se fokusira na vanjski izgled objekta, zanemarujući potpuno njegovu unutrašnjost odnosno implementaciju.

U objektno-orientiranim programskim jezicima, svakoj apstrakciji iz domena problema odgovara po jedna klasa. Klase su prototip iz koga se kreiraju primjeri (instance) objekata. Objekti u programskom jeziku su primjeri klase, i do njih se dolazi instanciranjem na osnovu definirane klase.

Enkapsulacija

Enkapsulacija je koncept potpuno komplementaran apstrakciji. Apstrakcija se fokusira na zapažanje osobina i ponašanja objekata, a enkapsulacija se fokusira na implementaciju koja će dovesti do želenog ponašanja. Programske jezice omogućavaju jednostavne mehanizme za podršku enkapsulacije. U programskim jezicima, implementacija obuhvata kreiranje identificiranih klasa u postupku apstrakcije. Svaka klasa ima svoje podatke-članove i svoje metode-članice, koji se jednim imenom nazivaju članovi klase. Sama enkapsulacija se realizira razdvajanjem javnih i privatnih dijelova klase. Javni dijelovi klase, obično su metode, koje čine interfejs te klase ka okolini. Privatni dijelovi, uglavnom podaci koji izražavaju stanje objekta, obuhvaćaju detalje implementacije koji za korisnika klase nisu važni i stoga on ne mora da zna.

Modularnost

Modularnost je koncept povezan isključivo sa samim programom, a sastoji se u formiranju modula koji mogu da se prevode odvojeno, ali koji ipak imaju dobro definirane veze prema drugim modulima. U tradicionalnom strukturiranom dizajnu, modularnost programa se uglavnom sastojala u podjeli programa u procedure i funkcije i njihovim povezivanjem u odgovarajući modul (fajl). U objektnom orijentiranom dizajnu prvo je potrebno uočiti logičke cjeline koje čine izvjestan broj klasa i onda formirati module. Razni jezici se vrlo različito odnose prema modulima. U C#-u to su programski moduli koji se grupiraju u imenovane prostore (*namespace*), a u Javi to su paketi.

Hijerarhija

I na kraju, postavlja se pitanje što raditi kada je broj apstrakcija identificiranih u domenu problema toliki da je njima teško upravljati. Za to je potrebno novo logičko grupiranje apstrakcija - hijerarhija. Hijerarhija klasa se u objektnim modelima i objektno-orientiranim programskim jezicima ostvaruje preko mehanizama nasljeđivanja uz koji se veže i polimorfizam, a sastoji se u tome da jedna apstrakcija može biti vrsta neke druge apstrakcije sa dodatnim osobinama. Naprimjer, ako su u domenu problema identificirane apstrakcije "prijevozno sredstvo", "voz" i "avion", onda o druge dvije apstrakcije se može razmišljati kao o specifičnim vrstama prve. Opravданje za to može se naći u činjenici da avion i voz jesu prijevozna sredstva. Klase koje odgovaraju ovim apstrakcijama povezane su na taj način što su klase koje odgovaraju avionu i vozu izvedene iz klase koja odgovara prijevoznom sredstvu, i na taj način nasljeđuju kompletну funkcionalnost te klase i dodaju, eventualno, neke svoje karakteristike koje ih čine autonomnim entitetima.

Neki od jezika koji pripadaju objektno orijentiranoj paradigmi su: Objective-C, Smalltalk, Delphi, Java, Javascript, C#, Perl, Python, Ruby. Mnogi jezici nisu čisto objektno-orientirani ali imaju ugrađene neke karakteristike objektno-orientirane paradigmе. U okviru ove knjige izučava se objektno-orientirani programski jezik C# i .NET okruženje.

Funkcijska paradigma

Osnovni nivo apstrakcije kod funkcionalne paradigmе je funkcija. Ova paradigmа zasnovana je na matematici, teoriji funkcija i lambda kalkulaciji. Primjeri funkcionalnih jezika su Common, Lisp, Haskell, F#. Također, jezici, koji nisu čisto funkcionalni jezici (Java, PHP, C#) ugrađuju sposobnost funkcijskog izražavanja.

Logička paradigma

Logička paradigmа bazirana je na aksiomima, pravilima i zaključcima. Ova paradigmа se naročito koristi u oblasti vještacke inteligencije. Izvršavanje programa se svodi na pretragu seta činjenica i primjenu pravila zaključivanja. Programski jezik Prolog je predstavnik ove paradigmе.

RAZVOJ PROGRAMSKIH RJEŠENJA

1.3.1. Historijski pregled programskih jezika

Historijski gledano, postojali su mnogi programski jezici. Neki su se zadržali do danas, neki su veoma kratko trajali, a neki su doživjeli transformaciju u neki od današnjih jezika. Slijedi prikaz samo nekih od tih jezika.

Prvi programski jezici su nastali u periodu od 1940 do 1960 godine.

Asemblerski jezik je prvi programski jezik niskog nivoa, nastao u ovom periodu, koji je predstavljao veći nivo od mašinskog jezika. Za pisanje koda koriste se simboli (ADD, SUM,...), ali i dalje postoje stroge ovisnosti između koda i računarske arhitekture.

Fortran (Formula Translating System) je imperativni programski jezik, razvijen u IBM-ovim laboratorijama 1954 godine, prvo bitno namijenjen za numeričke kalkulacije i naučne proračune. U svojoj inicijalnoj verziji sadržavao je 32 naredbe a njihov broj se povećavao sa svakom novom verzijom. FORTRAN je i danas u upotrebi.

Lisp je još jedan programski jezik nastao u ovom vremenskom periodu. Liste su osnovne strukture podataka u programskom jeziku Lisp i njihovo procesiranje je osnova ovog programskog jezika po čemu je i dobio ime. Osnovna podloga Lisp jezika je funkcionalno programiranje i lambda račun. Novija izvedenica ovog jezika je Common Lisp koji kombinira proceduralno, funkcionalno i objektno-orientirano programiranje.

COBOL (skraćeno od **CO**mmon **B**usiness **O**riented **L**anguage) nastao je 1959. godine.

COBOL program se dijeli na sekcije (značajne sekcije podataka i sekcije procedura), podržava rad sa datotekama, kreiranje raznih tipova izvještaja. Dugo je COBOL bio jedan od najkorištenijih jezika za razvoj poslovnih rješenja.

Drugi značajni period nastanka programskih jezika je period od 1960 do 1980 godina.

U ovom periodu nastali su mnogi programski jezici, od kojih su najviše uticaja na daljnji razvoj programiranja imali C i BASIC. Mnogi moderni programski jezici imaju korijene u nekom od programskih jezika ovog perioda.

BASIC (**B**eginners **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode) programski jezik nastao je 1964. godine u Dartmouth kolodžu u SAD-a. Jezik je podržavao manipulaciju sa podacima, varijablama, tipovima podataka, kontrolne strukture (if, petlje, goto,...), rad sa ulazno-izlaznim operacijama, funkcije. Ovaj programski jezik je okarakteriziran kao jezik opšte namjene, lagan za korištenje, interaktivan, neovisan o hardveru i operativnom sistemu. Na osnovu ovog jezika je

RAZVOJ PROGRAMSKIH RJEŠENJA

nastao i Visual Basic koji je omogućio i kreiranje grafičkog korisničkog interfejsa. Danas je ugrađen u mnoga okruženja.

C programske jezik spada u proceduralne programske jezike. Razvijen je i standardiziran u ranim sedamdesetim godinama 20. stoljeća. Autor programskog jezika C je Ken Thompson i Dennis Ritchie. C je prvenstveno bio namijenjen za kodiranje sistemskih programa. Projektiran je tako da omogući računarskim procesorima direktni pristup i rad sa registrima procesora, bitovima, bajtovima, memorijskim adresama i slično. Programi napisani u C-u su se veoma brzo izvršavali. Koristio se i za programiranje poslovnih aplikacija. C program sadrži predprocesorski dio, sekciju definiranje varijabli različitih ugrađenih i korisnički definiranih tipova podataka, glavnu funkciju `main` i korisnički definirane funkcije. Bogat je i kontrolnim iskazima.

C je prethodnik i osnova za jezike poput Concurrent C (Gehani 1989), Objective C(Cox, 1986), C++ (Stroustrup 1986).

Osamdesetih godina nastao je veliki broj novih programskih jezika od kojih su najznačajniji C++, Ada, MATLAB i Objective-C.

Bjarne Stroustrup, danski naučnik, 1979. godine je započeo rad na novom jeziku kojeg je prvo nazvao „C sa klasama“, a kasnije C++. Kao osnovu za C++ koristio je programski jezik C, ali i programski jezik Simula koji je posjedovao pojam klase. C++ posjeduje jezičke konstrukcije za izražavanje svih objektno-orientiranih koncepata koji su pomenuti u sekciji objektno-orientirana paradigma.

Objective-C je programski jezik opšte namjene nastao tokom ranih osamdesetih godina dvadesetog stoljeća. U suštini Objective-C zadržava sve aspekte C-a i dodaje sintaksu i semantiku koja omogućava objektno-orientirano programiranje. Objective-C je glavni programski jezik kompanije Apple.

Rapidni rast Interneta sredinom devedesetih godina označio je prekretnicu u programiranju. Najpoznatiji programski jezici razvijeni u periodu od 1990 do 2000 godine su Haskell, Python, Visual Basic, Java, Javascript i PHP. U ovo vrijeme pojavili su se i *markup* jezici - HTML.

Od 2000 godine do danas nastali su još mnogi programski jezici. Neki od njih su: ActionScript, C#, Visual Basic .NET, F#.

RAZVOJ PROGRAMSKIH RJEŠENJA

1.3.2. C# i .NET platforma

Programski jezik C# se distribuira zajedno sa posebnim okruženjem na kojem se izvršava – Common Language Runtime (CLR). CLR je skupina standardnih biblioteka koje pružaju osnovne funkcionalnosti, kompjulere, debagere (*debug*) i ostale razvojne alate. Zahvaljujući CLR-u programi su portabilni i mogu raditi bez izmjena ili sa minornim izmjenama na različitim hardverskim platformama i operativnim sistemima. C# podržava imenovane prostore (*namespace*), ima potpune jezičke konstrukcije za izražavanje objektno-orientiranih koncepata. Daje mogućnost implementacije interfejsa klase. Uvodi delegat tip podataka, omogućava programiranje vođeno događajima, lambda kalkulaciju, asinhrono programiranje. Posjeduje ekstenzije za rad sa grafičkim korisničkim interfejsom, datotekama, XML-om, bazama podataka. Omogućava paralelnu implementaciju pojedinih taskova, mehanizme za upravljanje greškama i izuzecima, kao i mehanizme za validaciju podataka. Navedene osobine C#-a su ključne za razvoj složenih programske rješenja i ti se koncepti obrađuju u okviru narednih poglavlja ovog udžbenika.

Karakteristično za C# jeste to što se sa svakom novom verzijom implementira sve više ključnih koncepata funkcionalnih programske jezike. Najbolji primjer za to je LINQ (Language Integrated Query) koji je implementiran u C# verziji 3.

Programsko okruženje (*framework*) je kolekcija alata koji se koriste za razvoj softvera. .NET je okruženje koje u sebi ima jako puno integriranih alata. Neki od njih su tekst editor, kompjuler, linker, debager, preglednik solucija (rješenja), interfejs za kreiranje grafičkog interfejsa i svakom se pristupa preko uniformnog korisničkog interfejsa.

Rezime:

U okviru ovog poglavlja dat je uvod u oblast softver inženjeringu. Razmatrano je šta je softver i kako se može napisati kvalitetan softver odnosno kvalitetno softversko rješenje. Navedeni su osnovni koraci razvoja programske rješenja i osnovne metodologije koje se primjenjuju prilikom razvoja programske rješenja. Pri razvoju programske rješenja bitan je i izbor programske jezike. Zbog toga je dat kratak pregled programskih paradigmi i hronološki historijski pregled jezika. Naveden je i razlog odabira C# jezika i .NET platforme.

RAZVOJ PROGRAMSKIH RJEŠENJA

Pitanja za ponavljanje i samostalni rad:

1. Navedite IEEE definiciju za softver.
2. Objasnite šta se podrazumjeva pod pojmom softver inženjeringu.
3. Navedite neka programska rješenja koja pripadaju skupini generičkih softverskih produkata.
4. Ukoliko bi trebali razviti prilagođeno programsko rješenje za evidenciju o održanoj nastavi i prisustvu studenata, koje funkcionalnosti bi ponudili u okviru tog rješenja.
5. Navedite osnovne korake razvoja softvera. Objasnite svaki pojedinačni korak razvoja.
6. Navedite i objasnite principe osnovnih metodologija/modela za razvoj softvera.
7. Navedite i objasnite osnovne programske paradigme.
8. Navedite predstavnike programskih jezika kroz karakteristična historijska razdoblja.
9. Navedite neke od karakteristika C# jezika i .NET platforme.

Poglavlje 2: Osnovni programski koncepti u C# jeziku

U okviru ovog poglavlja opisuju se elementarni tipovi podataka i osnovne programske strukture u C# jeziku kroz tematske jedinice:

1. Uvod u okruženje
2. Varijable i tipovi podataka
3. Iskazi selekcije
4. Kontrolne petlje
5. Metode

2.1. Uvod u okruženje

Postoji veliki broj okruženja koja omogućavaju razvoj programskih rješenja. S obzirom da je cilj ovog udžbenika da se korištenjem C# jezika prikaže osnovni razvoj programskih rješenja korišten je: Microsoft Visual C# Express Edition 2013 i Microsoft Visual Studio Ultimate Version. Ovo okruženje omogućava razvoj raznih tipova aplikacija. U okviru ovog udžbenika razvijaju se konzolne aplikacije (Console Application), biblioteke klase (Class Library) i Windows forms aplikacije (Windows Forms Application). Struktura konzolne aplikacije je prikazana ispod (kod 2.1). Prilikom kreiranja konzolne aplikacije kreira se imenovani prostor (*namespace*) sa nazivom koji je dat projektu. U okviru njega je klasa `Program` i metoda `Main` od koje počinje izvršavanje programa. *Namespace* je metoda organizacije programskog koda koja pomaže grupirajući tipove podataka i određene programske logike.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplicationP
{
    class Program      // 'osnovna' klasa sadrži Main metodu
    {
        static void Main(string[] args) // Main metoda
        {
            // programski iskazi
        }
    }
}
```

Kod 2.1: Osnovna struktura C# konzolne aplikacije

RAZVOJ PROGRAMSKIH RJEŠENJA

Sintaksa C# jezika u većini slučajeva je ista kao sintaksa C++ jezika. S obzirom da je auditorij ove knjige upoznat sa C++ jezikom samo će se kratko izložiti pojedini koncepti i specifičnosti C# jezika.

C# posjeduje većinu ključnih riječi kao i C++. Neke specifične ključne riječi koje postoje u C# jeziku i koje se objašnjavaju u ovom udžbeniku su:

```
abstract, bool, checked, delegate, event, explicit, fixed, implicit, readonly,  
volatile, unckecked, from, join, select, yield, get, let, set, group, orderby,  
value, into, partial, where.
```

Dio specifičnih C# ključnih riječi

2.2. Varijable i tipovi podataka

Varijable, klase, metode, ključne riječi imenuju se pomoću identifikatora.

2.2.1 Identifikatori

Sintaksna pravila za identifikatore:

- Koriste se mala i velika slova, cifre, i _.
- Identifikator mora početi sa slovom ili _.
- C# je osjetljiv na veličinu slova.

Primjeri validnih identifikatora: `result, Result` (različiti identifikatori), `_bodovi, fudbalskiTim, plan9`. `9plan` je nevalidan identifikator. C# također koristi identifikatore koji imaju posebno značenje ali nisu C# ključne riječi.

2.2.2. Varijable

Varijabla je sinonim za memorijsku lokaciju na kojoj se čuva vrijednost. Svakoj varijabli u programu se dodjeljuje jedinstven identifikator.

Opšte preporuke za imenovanje varijabli:

- Naziv varijable početi sa malim slovom.
- Naziv varijable treba da tačno i potpuno opisuje šta varijabla predstavlja.
- Umjesto _ u identifikatorima, bolje je koristiti sastavljeni dvije riječi.
- U identifikatorima sa više riječi, drugu i svaku novu početi sa velikim slovom (**projektVrijednost**).
- Ne davati imena varijablama da se razlikuju samo u jednom ili dva slova ili samo u veličini slova (npr. **projektVrijednost** i **ProjektVrijednost**)
- Izbjegavati brojeve pri imenovanju varijabli (npr. broj1, broj2,...-nije dobra praksa).

RAZVOJ PROGRAMSKIH RJEŠENJA

C# je jezik koji vrši jaku kontrolu tipova podataka (*strongly-typed*). Za svaku varijablu se veže tip varijable kojim se navodi mogući rang vrijednosti i operatori koji se mogu nad njima primijeniti.

C# posjeduje dvije osnovne kategorije tipova podataka: vrijednosni tipovi (*value-type*) i referencni tipovi (*reference-type*).

Varijable referencnih tipova podataka smještaju reference na podatke (objekte), dok varijable vrijednosnih tipova direktno sadrže podatke. Više varijabli referencnih tipova mogu referencirati isti objekat. Za razliku od varijable referencnog tipa, svaka varijabla vrijednosnog tipa ima svoju kopiju podatka.

2.2.3. Vrijednosni tipovi

U okviru vrijednosnih tipova podataka su osnovni (elementarni, primitivni) tipovi podataka za cjelobrojne, realne, karakterne vrijednosti, složeni tip podataka – struktura i prebrojivi tip podataka.

Osnovni elementarni tipovi podataka su:

-**byte** (označava cjelobrojni tip u rangu od 0-255),

-**sbyte** (označava cjelobrojni tip u rangu od -128 do 127),

-**short, int, long** (16, 32, 64 -bitni tip podataka, za smještanje cjelobrojnih vrijednosti sa predznakom),

-**ushort, uint, ulong** (označavaju pozitivne cjelobrojne vrijednosti koje se mogu smjestiti na 16, 32, 64-bitu),

-**float** (realne vrijednosti sa predznakom koje se smještaju na 32 bita, preciznost 7 cifara),

-**double** (realne vrijednosti sa predznakom koje se smještaju na 64 bita, preciznost 15-16 cifara),

-**decimal** (128-bitni tip podataka, koristi se za financijska i monetarna računanja, u odnosu na **float** ima manji rang, ali veću preciznost),

-**bool** (logički tip podataka-vrijednosti **true, false**),

-**char** (označava jedan karakter, koji se naznačava između jednostrukih navodnika).

Za sve varijable je potrebno izvršiti eksplicitnu deklaraciju navođenjem tipa i naziva varijable. Prilikom deklaracije varijable, poželjno je varijabli dodijeliti inicijalnu vrijednost. Operator pridruživanja = se koristi da se izvrši dodjela vrijednosti. Iskazi se završavaju sa tačka-zarez ;. Deklaraciju varijablu je najbolje uraditi blizu njenog prvog korištenja. U ovisnosti od mesta deklaracije varijabla ima određen vidokrug (*scope*) vidljivosti i upotrebe.

RAZVOJ PROGRAMSKIH RJEŠENJA

Primjeri deklaracija i inicijalizacija varijabli:

- long iznosIsplate; // deklaracija long tipa varijable sa nazivom iznosIsplate
- int osnova=0,dodatak=0; // deklaracija i inicijalizacija int tipa varijabli osnova i dodatak
- float uplata; // deklaracija float tipa varijable uplata
- decimal kusur = 23.5m; //deklaracija i inicijalizacija decimal varijable kusur

Moguće je izvršiti istovremenu deklaraciju i inicijalizaciju varijable korištenjem **new** operatora.

Iskazom,

```
bodoviPismeni = new int();
```

izvršena je deklaracija varijable `bodoviPismeni`, dodijeljen joj je `int` tip podataka i inicijalna vrijednost 0.

Prilikom dodjeljivanje vrijednosti varijabli koriste se i dodatni specifikatori F,f-za `float` varijable, D,d-za `double` varijable; L,l-za `long` varijable; M,m-za decimalne varijable; U,u-za cjelobrojne (`int`) vrijednosti bez predznaka; ul,UL-za `long` cjelobrojne tipove bez predznaka. U slučaju izostavljanja specifikatora ako je vrijednost varijable u rangu drugog srodnog tipa podataka, varijabla se tretira kao da je tog tipa.

Npr: `long suma=50;` se tretira (ukoliko se ne javi kompjuterska greška) kao varijabla `int` tipa, dok `long suma=50L` se tretira kao varijabla `long` tipa.

Osnovni vrijednosni tipovi podataka su već ugrađeni i unaprijed definirani u okviru implementacije C# jezika i .NET okruženja. .NET *framework* za sve osnovne tipove podataka ima alijsase koji su definirani u okviru imenovanog prostora `System`. Naprimjer, za `int` tip podataka alias je `System.Int32`, za `float` `System.Single`, za `double` `System.Double`. Ujedno okruženja nude i mnoge metode za rad sa tipovima podataka, npr. metoda `GetType()` određuje tip varijable. Kod 2.2. prikazuje način deklaracije, inicijalizacije varijabli raznih tipova podataka.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;

namespace tipPodataka1
{
    class Program
    {
        static void Main(string[] args)
        {
            // deklaracija i inicijalizacija varijabli
            float uplata=3866.92F;
            uint ocjena = 8;
            decimal kusur = 70.6m;
            bool flag = true;
            int brojStudenata=new Int32();

            // prikaz tipa i vrijednosti varijabli
            Console.WriteLine("Tip za 1E06 je {0}", 1E06.GetType());
            Console.WriteLine("uplata Tip/Vrijednost: {0} {1}",uplata.GetType(),uplata);
            Console.WriteLine("ocjena Tip/Vrijednost: {0} {1}", ocjena.GetType(),ocjena);
            Console.WriteLine("kusur Tip/Vrijednost: {0} {1}", kusur.GetType(),kusur);
            Console.WriteLine("flag Tip/Vrijednost: {0} {1}", flag.GetType(),flag);
            Console.WriteLine("brojStudenata Tip: {0}",brojStudenata.GetType());
        }
    }
}
```

Kod 2.2: Deklaracija i inicijalizacija varijabli, prikaz tipa i vrijednosti varijabli

Izvršavanjem koda 2.2 se dobija:

```
C:\Windows\system32\cmd.exe
Tip za 1E06 je System.Double
uplata Tip/Vrijednost: System.Single 3866.92
ocjena Tip/Vrijednost: System.UInt32 8
kusur Tip/Vrijednost: System.Decimal 70.6
flag Tip/Vrijednost: System.Boolean True
brojStudenata Tip: System.Int32
```

Za detaljniji opis pojedinih klasa (u ovom slučaju `Console` klase) preporučuje se izučavanje osobina i metoda klase sa oficijelne stranice .Net frameworka: <http://msdn.microsoft.com/en-us/library/system.console.aspx>.

Konverzije tipova

C# pretvara implicitno ili eksplisitno instancu jednog tipa podataka u drugi tip podataka.

Implicitna konverzija je tip sigurna konverzija koja ne uzrokuje gubljenje ili promijenu vrijednosti podatka. Kompajler automatski obavlja ovaj tip konverzije za ugrađene numeričke tipove podataka.

Eksplisitna numerička konverzija se koristi za konverziju varijable jednog tipa podatka u drugi. Obavlja se sa `cast` operatorom. Eksplisitna konverzija može uzrokovati gubljenje preciznosti. Prilikom ovog tipa konverzije mogu se pojaviti i izuzeci.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
int x = 12345;
long y = x;      // Implicitna konverzija
short z = (short)x; // Eksplisitna konverzija
```

enum tip podataka

Za definiranje tipa podataka koji se sastoji od liste imenovanih konstanti koristiti se ključna riječ `enum` kojom se uvodi nabrojivi (prebrojivi) vrijednosni tip podataka. Podrazumijevani tip podataka članova `enum` (nabrojivog) tipa podataka je cjelobrojni (`int`) tip podataka. Koristi se kada su poznate sve moguće vrijednosti varijabli i kada je bolje da se njihove vrijednosti izraze riječima. Ukoliko se naprimjer želi koristiti skup boja, određeni dani, mjeseci, definirati minimalna i maksimalna vrijednost korisno je kreirati `enum` tip podataka. Upotreba ovog tipa podataka povećava čitljivost, kod je jednostavniji za promjenu, lakša je provjera opcija izbora u višestrukim iskazima izbora. Najbolje je da se definira direktno unutar imenovanog prostora da bi ga sve članice mogle koristiti.

```
enum Boje {Crvena, Plava, Bijela, Zelena, Siva}; // definicija tipa Boje
Boje bojice; // deklaracija varijable bojice
bojice=Boje.Plava; // dodjela vrijednosti varijabli bojice
```

struct tip podataka

Struktura (`struct`) je vrijednosni tip podataka koji se koristi za formiranje logičke cjeline -grupe više povezanih varijabli koje mogu biti različitih tipova. Naprimjer, knjiga se opisuje sa nazivom, identifikacijskim brojem, autorom, pravougaonik sa vrijednostima koordinati, datum sa danom, mjesecom, godinom. Prilikom definicije strukture, strukturi se dodjeljuje naziv koji opisuje namjenu strukture, u okviru {} se navode pojedini članovi.

Slijedi primjer predstavljen sa kodom 2.3. jednostavne definicije strukture:

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;

class Program
{
    // definicija strukture Datum
    struct Datum
    {
        public int dan;
        public int mjesec;
        public int godina;
    };

    static void Main()
    {

        Datum tekuciDatum; //deklaracija varijable tekuciDatum tipa strukture Datum

        // postavljanje vrijednosti za članove varijable tekuciDatum tipa strukture Datum
        tekuciDatum.dan = 1;
        tekuciDatum.mjesec = 1;
        tekuciDatum.godina = 2015;

        // prikaz vrijednosti članova varijable tekuciDatum tipa strukture Datum
        Console.WriteLine(" Danas je {0}/{1}/{2}",tekuciDatum.dan,
        tekuciDatum.mjesec,tekuciDatum.godina);
    }
}
```

Kod 2.3: Upotreba strukture u C# programu

Struktura je korisna zbog direktne manipulacije blokovima podataka. Struktura pojašnjava veze između podataka i čini čitljivijim program jer je često teško ako postoji veliki broj varijabli uočiti veze između njih. Strukture pojednostavljaju operacije nad blokovima podataka, jer je jednostavnije izvršavati operacije nad strukturom nego izvršavati iste operacije nad svakim pojedinačnim elementom. Strukture pojednostavljaju održavanje programskog koda. Veće mogućnosti od strukture imaju klase koje se objašnjavaju u narednom poglavlju.

Nad varijablama elementarnog tipa podataka moguće je primjeniti i operatore u cilju izvršavanja raznih matematičkih operacija, operacija pridruživanja, poređenja i ostalih operacija.

RAZVOJ PROGRAMSKIH RJEŠENJA

Operatori

- Operator pridruživanja je `=` (znak jednako).
- C# podržava regularne aritmetičke operatore:
`(+)` sabiranje, `(-)` oduzimanje, `*` množenje, i `/` dijeljenje.
- Aritmetički operatori se mogu koristiti na tipovima: `char`, `int`, `long`, `float`, `double`, `decimal`.
- Ne mogu se aritmetički operatori (izuzetak `+`) koristiti za `string` ili `bool`.
- Tip rezultata aritmetičkih operacija ovisi o tipu korištenih operanada.
- C# podržava i operator za ostatak cjelobrojnog dijeljenja `(%)`.
- Operator inkrement `++` i dekrement u pre-fiksnoj i post-fiksnoj notaciji.
- C# ima više korisnih logičkih operatora: `!` je logički NOT operator, `&&` je logički AND operator, `||` je logički OR operator.
- Operatori jednakosti `==`, `!=`.
- Relacijski operatori `<`, `<=`, `>`, `>=`.

Redoslijed izvršavanja

- C# redoslijed izvršavanja operatora je kao u algebri `(* / %)` prednost u odnosu na operatore `(+ i -)`.
- Zgrade imaju najveći prioritet.
- Asocijativnost s lijeva na desno.

Primjeri iskaza: `suma = 10; suma=suma+10; suma++; (suma-10)%2;`

2.2.4. Referenci tipovi

C# obezbjeđuje ugrađene referencne tipove koji se označavaju ključnim riječima: `dynamic`, `object`, `string` i koji se objašnjavaju u nastavku ove sekcije. U referencne tipove podataka ubrajaju se i korisnički tipovi podataka - klase, interfejsi, delegati čije uloge se objašnjava u poglavljima 3 i 4. Kada se objekat referencnog tipa kreira, alocira se memorijski prostor na hipu (*heap*) i vraća se referenca na objekat. Za vrijeme izvršavanja (*runtime*) programa, skupljač smeća (*garbage collector*) periodično dealocira objekte sa hipom.

RAZVOJ PROGRAMSKIH RJEŠENJA

object tip podataka

Varijabli tipa `object` može se dodijeliti vrijednost bilo kojeg tipa. Svi tipovi podataka referenci i vrijednosni, ugrađeni ili korisnički definirani su direktno ili indirektno naslijedjeni od `object` tipa (u .NET *frameworku* alias `Object`). Proces konverzije vrijednosnog tipa u `object` tip je implicitni proces i naziva se *boxing*, dok proces ekstrakcije vrijednosnog tipa iz objekta je eksplicitni proces i naziva se *unboxing*.

```
int i = 123;  
object o = i;  
  
o = 123; // boxing  
i = (int)o; // unboxing
```

dynamic tip podataka

C# vrši statičku provjeru ispravnosti tipa neke varijable. Sa `dynamic` tipom se zaobilazi statička provjera tipa podataka za vrijeme kompilacije i zbog toga može se desiti da objekat promijeni svoj tip podatka.

string tip podataka

Tip podataka koji predstavlja sekvencu (literal) nula ili više karaktera uključujući i *escape* karaktere (\) naziva se `string`. String literali mogu se pisati u dvije forme: označeni sa navodnicima (*quoted*) i @-*quoted*. Karakteristika @-*quoted* literala je da *escape* sekvence se ne procesiraju. Druga namjena @ simbola je da referencira identifikatore koji su C# ključne riječi.

Primjer string literalisa označenog navodnicima: "Kurs Razvoj programskih rješenja".

Primjer @-*quoted* literala za pisanje puta do fajla: @"c:\Docs\test\a.txt".

Iako je `string` referenci tip podataka, operatori jednakosti (== i !=) se upotrebljavaju za poređenje objekata `string`ova. Nad `string`ovima je dozvoljen i operator sabiranja (+) koji spaja `string`ove, operator [] koji se koristi za pristup individualnim karakterima stringa.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;

namespace StringPrimjer
{
    class Program
    {
        static void Main(string[] args)
        {
            // deklaracija string varijabli
            string prvaRijec = "Razvoj ";
            string drugaRijec = "programskih rješenja";

            // primjena operatora plus + i operatora jednakosti ==
            Console.WriteLine( prvaRijec + drugaRijec == "Razvoj programskih rješenja" );
            Console.WriteLine(prvaRijec + drugaRijec == "Razvoj Programske Rješenja");

        }
    }
}
```

Kod 2.4: Upotreba stringa u C# programu

Prilikom izvršavanja koda 2.4 koji demonstrira upotrebu stringa i operatora u C# programu prvim pozivom metode `Console.WriteLine` prikazao bi se rezultat `True`, a drugim `False` jer je operacija poređenja stringova osjetljiva na veličinu slova.

Za rad sa stringovima razvojna okruženja pružaju razne biblioteke funkcija. Net *framework* pruža klase `String`, `StringBuilder` sa velikim brojem metoda za rad sa stringovima.

Često potrebna konverzija `string` tipa podataka u `int` se radi sa `System.Int32.Parse` metodom.

```
System.Int32.Parse("42");
```

var oznaka

Ako se varijabla pri deklaraciji označi sa `var` tada C# određuje koji je tip varijabla na osnovu izraza koji se koristi za inicijalizaciju.

```
var broj = 99; // varijabla broj je tipa int;
var predmet = "Razvoj"; // varijabla predmet je tipa string;
```

Provjera konteksta

C# omogućava da se odredi da li će se neki kontekst provjeravati (`checked`) ili neće (`unchecked`) u cilju otkrivanja prekoračenja. Prekoračenju su podložni izrazi koji koriste predefinirane operatore (`++, --, -, *, /, ...`) i eksplisitne numeričke konverzije između tipova podataka.

RAZVOJ PROGRAMSKIH RJEŠENJA

Ako se eksplisitno ne navede provjera za kontekst, podrazumijevana opcija ovisi od postavljenih opcija kompjerala. Naprimjer, ako je `checked` kontekst, prekoračenje kod aritmetičkog računanja podiže izuzetak (*exception*), dok ako je `unchecked` kontekst dobija se skraćen rezultat. Može se koristiti za izraz ili blok iskaza.

```
int a = 1000000;
int b = 1000000;
int c = checked (a * b); // provjerava prekoračenje, javlja se izuzetak
int c = unchecked (a * b); // ne provjerava prekoračenje, ne javlja se izuzetak
```

POINTER tip podataka

Pointer tip podataka sadrži adresu neke varijable. Za svaki vrijednosni tip podataka, pointer tip podataka i strukturu koja ne sadrži reference može se definirati pointer tip podataka.

Deklaracijom se navodi na koji tip podatka pokazuje pointer i naziv pointera, tako što se navede tip podatka, `*`, i jedan ili više identifikatora koji predstavljaju naziv pointera (`tip_podataka* nazivPointera;`)

Naprimjer:

```
int* p1; // p1 je pointer na varijablu int tipa, adresa te varijable je sadržana u p1, *p1-se koristi za pristup sadržaju te varijable
int* p1, p2, p3; // p1, p2, p3 su pointeri na varijablu int tipa
```

Blokovi koda u kojima je smještena manipulacija sa pointer podacima označavaju se kao nesigurni (*unsafe*).

Pored operatorka `*`, nad varijablama pointer tipa se koriste i sljedeći operatori: `&` kojim se očitava adresa varijable, operator `→` za pristup članovima strukture preko pointera, operatori inkrementa, dekrementa pointer vrijednosti, aritmetički operatori sabiranja i oduzimanja, operatori poređenja, indeksni operator `[]`, `stackalloc` koji se koristi za alociranje memorije, `fixed` koji se koristi za privremeno fiksiranje adrese varijable.

2.2.5.Nizovi

Nizovi se koriste za smještanje i grupiranje više varijabli istog tipa. Moguće je kreirati jednodimenzionalni niz, višedimenzionalni niz i *jagged* niz. Elementi niza mogu biti bilo kojeg tipa. Broj dimenzija i veličina svake dimenzije se uspostavlja kada se instanca niza kreira i ne može se mijenjati za vrijeme života te instance. Indeks prvog elementa niza je 0, n -tog elementa $n-1$. Operator `new` se koristi za kreiranje niza i inicijalizaciju elemenata niza sa inicijalno

RAZVOJ PROGRAMSKIH RJEŠENJA

podrazumijevanim vrijednostima. Postoje različiti načini deklaracije i inicijalizacije niza. Slijedi prikaz nekih od njih:

```
// deklaracija jednodimenzionalnog niza od 5 elemenata korištenjem new operatorom
int[] niz1 = new int[5];

// deklaracija i inicijalizacija jednodimenzionalnog niza od 5 elemenata korištenjem new operatora
int[] niz2 = new int[] { 1, 3, 5, 7, 9 };

// deklaracija i inicijalizacija elemenata jednodimenzionalnog niza bez korištenja new operatora
int[] niz3 = { 1, 2, 3, 4, 5, 6 };

// deklaracija dvodimenzionalnog (2 reda, 3 kolone) niza korištenjem new operatora
int[,] dvodimenzionalniNiz1 = new int[2, 3];

// deklaracija dvodimenzionalnog niza (2 reda, 3 kolone)
int[,] dvodimenzionalniNiz2 = { { 1, 2, 3 }, { 4, 5, 6 } };

// deklaracija niza string tipa
string[] radniDani = { "Ponedeljak", "Utorak", "Srijeda", "Četvrtak", "Petak" };
```

U C#-u se može kreirati niz kojem se implicitno na osnovu vrijednosti članova inicijalne liste dodjeljuje tip (*implicitly typed* niz). Ovaj tip nizova se obično koristi u izrazima upita zajedno sa anonimnim tipovima.

Primjeri deklaracija i upotrebe *implicitly-typed* niza:

```
var nizInt = new[] { 10, 100, 1000, 10000 };
var nizString = new[] { "Razvoj", "programskih", "rješenja" };

Console.WriteLine("Drugi element niza cjelobrojnih vrijednosti: {0}", nizInt[1]);
Console.WriteLine("Zadnji element niza string vrijednosti: {0}", nizString[2]);
```

Jagged (zupčast) niz je niz čiji elementi su nizovi. Naziva se i "niz nizova". Elementi *jagged* niza mogu biti različitih dimenzija i veličina.

Sljedeći primjer pokazuje kako se deklarira, inicijalizira i koristi *jagged* niz.

```
int[][] jaggedNiz = new int[3][];
jaggedNiz[0] = new int[5];
jaggedNiz[1] = new[] { 10, 100, 1000, 1000 };
jaggedNiz[2] = new int[2];
```

Svaki od elemenata *jagged* niza je jednodimenzionalni niz cjelobrojnih vrijednosti (prvi niz ima 5, drugi 4 i treći 2 elementa). Prilikom deklaracije *jagged* niza moguće je koristiti inicijalizatore za elemente niza. U tom slučaju nije potrebna veličina niza.

.NET *framework* tretira niz kao referencni tip podataka, koji je izведен iz apstraktne klase *Array*.

2.2.6. Konstante

Konstantne su nepromjenjive vrijednosti, određene i poznate za vrijeme kompilacije. Deklaracija konstanti se vrši sa `const` modifikatorom. Prilikom deklaracije konstanti vrši se i njihova inicijalizacija. Samo C# ugrađeni tipovi (isključujući `System.Object`) mogu biti deklarirani kao `const`. Korisnički definirani tipovi uključujući klase, strukture, ili nizove ne mogu biti konstante. Za deklaraciju klasa, struktura ili nizova za koje se želi da članovi budu nepromjenjivi koristi se `readonly` specifikator pristupa.

```
public const int mjeseci = 12;
const int mjeseci = 12, dani = 365; //više konstanti istog tipa
const double brojDanaMjesec = (double) dani / (double) mjeseci;
```

Konstantama se mogu dodijeliti `public`, `private`, `protected`, `internal`, ili `protected internal` specifikator pristupa koji će biti objašnjeni u poglavlju posvećenom klasama.

Posebna vrsta podataka su generički tipovi podataka koji se objašnjavaju u 4.5.

2.3. Iskazi selekcije

U okviru ove sekcije se prikazuje sintaksa i semantika uvjetnih iskaza selekcije u C#-u.

2.3.1. if iskaz

C# podržava više varijanti `if` iskaza. Najjednostavniji je `if` iskaz kojim se navodi logički izraz, testira njegova vrijednost i ovisno od rezultata izraza se izvršava ili ne izvršava blok naredbi. Sa `if-else` iskazom se ovisno od rezultata logičkog izraza koji se navodi iza ključne riječi `if` izvršava jedan od dva bloka naredbi (blok iza `if` ili blok iza `else`). Sintaksa `if-else` iskaza je:

```
if ( logickiIzraz )
    // jedan ili više iskaza ako je više koristi se {}

else
    // jedan ili više iskaza ako je više koristi se {}
```

Ukoliko nije navedena `else` grana dobija se `if` iskaz. Prilikom pisanja `if` iskaza preporučuje se da se prvi piše slučaj za koji se očekuje da će se češće procesirati prilikom izvršavanja koda.

RAZVOJ PROGRAMSKIH RJEŠENJA

Moguće je ugnježdavanje `if` iskaza unutar drugih `if` iskaza. Pri tome se formiraju kaskadni (ugnježdeni) `if` iskazi forme `if-else-if`. Primjer kaskadnog ugnježdenog iskaza:

```
if (dan == 1)
    danNaziv = "Ponedeljak";
else if (dan == 2)
    danNaziv = "Utorak";
else if (dan == 3)
    danNaziv = "Srijeda";
else if (dan == 4)
    danNaziv = "Cetvrtak";
else if (dan == 5)
    danNaziv = "Petak";
else
    danNaziv = "Vikend dan";
```

Prilikom pisanja `if-else-if` iskaza preporučuje se da se opcija za koju je najveća vjerovatnoća da će se desiti prva testira sa prvim `if`-om; ukoliko ima složenih testova izraza da se isti pojednostavije sa pozivima logičkih funkcija. Provjeriti i da su svi slučajevi pokriveni kodom. Također, preporučuje se da ukoliko je lanac `if-else-if` dug, da se zamijeni sa nekom drugom jezičkom konstrukcijom za višestruki izbor.

2.3.2. `switch` iskaz

Za višestruki izbor sa kojim je potrebno testirati varijablu u odnosu na različit set konstantnih vrijednosti koristi se `switch` iskaz. Koristi se samo za testiranje vrijednosti osnovnih tipova podataka kao što su `int` ili `string`. Varijabla koja se testira se piše iza `switch` klauzule, a sa `case` klauzulama se navode vrijednosti sa kojima se vrijednost te varijable upoređuje i pripadajući kod koji se izvršava ako je poređenje tačno. Pri tome, sve `case` labele moraju imati konstantni izraz, jedan izraz se veže uz jedan `case`. Može biti više `case` labele vezanih za jednu granu. Sa `break` iskazom se postiže da se prekida daljne procesiranje `switch` iskaza. Ako nijedna grana se ne izvrši, tada se ako je navedena izvršava `default` grana koja se koristi za detekciju legitimne podrazumijevane vrijednosti ili za detekciju grešaka. Primjer `switch` iskaza na kojem se može uočiti i sintaksa ovog iskaza dat je ispod.

```
string mjesec = "Oktobar";
switch (mjesec)
{
    case "Oktobar":
    case "Decembar": Console.WriteLine("Nastava");
        break;
    case "Novembar": Console.WriteLine("Prvi Parcijalni Ispit");
        break;
    case "Januar": Console.WriteLine("Drugi Parcijalni Ispit");
        break;
}
```

RAZVOJ PROGRAMSKIH RJEŠENJA

Za povećanje čitljivosti koda bitna je organizacija `case` iskaza. Ako postoji veći broj `case` iskaza njihov redoslijed je bitan. Pri tome, preporučuje se da se slučaj za koji se očekuje da će se najviše izvršavati postavi prvi, ako su `case` slučajevi iste važnosti i učestalosti preporučuje se da se poredaju alfabetски ili numerički.

2.4. Kontrolne petlje

Petlja (*loop*) je neformalni izraz koji označava bilo koju vrstu iterativne kontrolne strukture – strukture koja uzrokuje da program ponavlja izvršavanje bloka koda. C# podržava uobičajne vrste petlji u programskim jezicima koje se izvršavaju određen broj puta (`for` petlja), evaluirajuće petlje koje testiraju uvjet prekida poslije svake iteracije (`while`, `do-while`), iteratorske petlje izvršavaju navedene akcije za svaki element u kontejnerskoj klasi (`foreach`).

2.4.1. `while` i `do-while` petlja

`While` i `do-while` petlje se koriste ukoliko nije unaprijed poznat broj iteracija. `While` petlja na početku svake iteracije provjera uvjet izvršavanja, i ukoliko je tačan izvršava se ta iteracija i pripadajući blok naredbi, dok `do-while` petlja nakon svake iteracije provjerava uvjet izvršavanja.

Opšta forma `while` petlje je:

```
while ( logičkiIzraz )
    blokNaredbi
```

a `do-while` je:

```
do
{
    blokNaredbi
} while (logickiIzraz);
```

Primjer `while` i `do-while` petlji u C#-u:

```
int i = 0;
while (i < 10)
{
    Console.WriteLine(i);
    i++;
}

int j=0;
do
{
    Console.WriteLine(j) ;
    j++;
} while (j<10) ;
```

2.4.2. `for` petlja

`for` petlja se koristi kada je potrebno izvršavati neki blok iskaza specificirani broj puta. Često se koristi i pri radu sa nizovima ili kontejnerskim klasama. Kontrola broja izvršavanja petlje je na kontrolnoj liniji petlje, dok unutar same petlje se navode iskazi koje je potrebno izvršiti.

Opšta forma `for` petlje je:

```
for (inicijalizacije; logickiIzraz; kontrolnaVarijabla)
    iskaz(i)
```

Primjeri `for` petlji u C#-u:

```
for (int i = 0; i < 10; i++)
    Console.WriteLine(i);
```

```
for (int i=0; i<10;i+=2)
    Console.WriteLine(i);
```

2.4.3. `foreach` petlja

C# pruža i `foreach` petlju koja omogućava jednostavan način iterativnog prolaska kroz elemente niza, prebrojive (*enumerable*) kolekcije, kao i kolekcije klase. Prednost ove petlje u odnosu na `for` petlju je eliminacija dodatnih iskaza za pristup elementima, čime je eliminirana šansa od grešaka prekoračenja ili pristupa pogrešnim elementima niza. `foreach` iskaz se koristi za iteraciju kroz kolekciju da bi se dobole željene informacije, ali se ne koristi da bi se dodale ili izbrisale stavke iz izvorne kolekcije. Ako je potrebno da se dodaju ili izbrišu stavke iz izvorne kolekcije koristi se `for` petlja.

Sljedeći kod kreira niz pod nazivom `brojevi` i vrši iteraciju kroz niz sa `foreach` iskazom:

```
int[] brojevi = { 40, 50, 60, 70, -40, -50, -60, -70 };
foreach (int i in brojevi)
{
    System.Console.Write("{0} ", i);
}
```

Izvršavanjem programa koji sadrži ovaj kod na standardnom izlazu se pišu vrijednosti niza: 40, 50, 60, 70, -40, -50, -60, -70

U bilo kojoj tačci `foreach` bloka, može se prekinuti petlja korištenjem `break` ključne riječi ili preći na novu iteraciju korištenjem `continue` ključne riječi. `foreach` loop se može prekinuti i sa `goto`, `return` ili `throw` iskazima.

RAZVOJ PROGRAMSKIH RJEŠENJA

2.4.4. Iskazi promjene toka izvršavanja programskog koda

Iskazi skakanja ili iskazi promjene toka izvršavanja vrše prekid izvršavanja određene sekvence izvršavanja koda i uzrokuju promjenu toka programa u odnosu na navedenu. C# iskazi skakanja su: `break`, `continue`, `goto`, `return` i `throw`.

Iskaz `break` se koristi za prekid (terminiranje) izvršavanja petlje ili `switch` iskaza i skok na prvi iskaz nakon iskaza koji se terminira.

Iskaz `continue` se koristi za preskakanje dijela iskaza u tijelu petlje i nastavljanje sa narednom iteracijom petlje. Koristi se u okviru `while`, `do`, `for`, ili `foreach` iskaza.

Iskaz `goto` prebacuje programsku kontrolu direktno na labelirani iskaz. Preporučuje se izbjegavanje upotrebe `goto` iskaza jer čine program nečitljivim. Ovaj iskaz ipak može biti i koristan, naprimjer u slučaju kada treba iz duboko ugnježdene petlje prekinuti izvršavanje svih petlji.

Iskaz `return` se koristi za terminiranje izvršavanja metode i vraćanje programske kontrole na mjesto poziva te metode. Sa `return` iskazom može se vratiti i neka vrijednost. Iskaz `throw` se koristi za izazivanje izuzetka za vrijeme izvršavanja programa (detaljnije u poglavljju 10).

```
using System;

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

            for (int i = 1; i <= 20; i++)
            {
                if (i % 5 == 0)
                {
                    continue;
                }

                if (i == 16)
                {
                    break;
                }

                Console.WriteLine(i);
            }

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

Kod 2.5: Upotreba `break` i `continue` iskaza

RAZVOJ PROGRAMSKIH RJEŠENJA

Na standardnom izlazu prilikom izvršavanja koda 2.5 koji sadrži iskaze `continue` i `break` ispisuju se brojevi: 1,2,3,4,6,7,8,9,11,12,13,14. Sa `continue` su preskočene iteracije za `i=5`, `i=10` i `i=15`. Sa `break` pri `i=16` prekinuto je izvršavanje for petlje.

2.5. Metode

Metoda je imenovana sekvenca iskaza. Koristi se da bi se program podijelio u manje logičke cjeline, čime se povećava čitljivost i održavanje programa. Bitne tačke za metodu su definicija metode i tačka poziva metode. Definicija metode u C#-u je uvijek unutar neke klase. Jedna metoda se može pozivati više puta.

Definicija metode je:

```
vidljivost povratniTip nazivMetode( listaParametara )
{
    // iskazi
    return povratnaVrijednost;
}
```

Naziv metode je identifikator koji opisuje namjenu metode (npr. *izracunajTaksu*). Metoda može imati nijedan, jedan ili više parametara. Eksplicitno se moraju specificirati tipovi svakog parametra, ne može se koristiti `var` ključna riječ. Parametri se odvajaju sa zarezom. Metoda može vratiti vrijednost. U tom slučaju obavezno je navesti povratni tip i `return` iskaz u tijelu metode. Ako metoda ne vraća vrijednost tada je njen povratni tip `void`. Ukoliko se prilikom definicije metode ne navede vidljivost metode tada je metoda private vidljivosti odnosno vidljiva je u klasi u kojoj je definirana. Više o specifikatorima vidljivosti u sljedećem poglavljju.

Prilikom poziva metode, navodi se naziv metode i lista argumenata, pri čemu argumenti redoslijedom i tipom moraju odgovarati listi parametara u definiciji metode. Sintaksa za poziv C# metode je:

```
[rezultat = ] nazivMetode ( [listaArgumenata] )
```

Ukoliko metoda vraća vrijednost može se navesti u koju varijablu se prihvata ta vrijednost `[rezultat =]`. Parametri se mogu metodi predati po vrijednosti i po referenci. Način smještanja tih parametra u memoriju odgovara načinu smještanja vrijednosnih i referencnih tipova.

Varijable definirane u metodi imaju vidokrug (*scope*) metode i nazivaju se lokalne. Vidokrug variabile je region u kojem se varijabla može koristiti. U svakom bloku {} mogu se definirati lokalne variabile. Pored lokalnih postoje i globalne variabile koje se definiraju kao zajedničke za

RAZVOJ PROGRAMSKIH RJEŠENJA

sve metode u klasi ili za više klasa. Vidokrug varijable se može odrediti i specifikatorima pristupa koji se navode u sljedećem poglavlju. Kod 2.6. predstavlja C# program sa standardnom metodom `Main` i jednom korisnički definiranom metodom `celzijusKelvin`. Metoda `Main` je ulazna tačka za svaku C# aplikaciju, i poziva se kada se program startuje. Unutar nje se poziva korisnički definirana metoda `celzijusKelvin` kojoj se predaje parametar koji predstavlja temperaturu u celzijusima. Metoda pretvara tu temperaturu u kelvine i vraća vrijednost u metodu `Main`.

```
using System;

namespace PrimjerMetoda
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine (celzijusKelvin (26.7)); // poziv metode
        }

        // definicija metode
        static double celzijusKelvin (double celzijus)
        {
            // tijelo metode u { }, sadrži deklaracije varijabli, programske iskaze, return iskaz

            double kelvin = celzijus+273.15;
            return kelvin;
        }
    }
}
```

Kod 2.6: Primjer metode u okviru C# programa

Rezime:

U okviru ovog poglavlja prikazana je struktura C# programa. Navedeni su osnovni vrijednosni i referencni tipovi podataka. Prikazane su osnovne strukture selekcije, kontrolne petlje. Data je i struktura metode. Navedene su i specijalne ključne riječi za C#. S obzirom da je udžbenik namijenjen auditoriju koje ima osnovno znanje o programskim konceptima u ovom poglavlju nisu detaljno objašnjavani osnovni koncepti, ali dat je presjek najvažnijih koncepata i koncepata koji su osobeni za C#.

RAZVOJ PROGRAMSKIH RJEŠENJA

Pitanja za ponavljanje i samostalni rad:

1. Upoznajte se sa kreiranjem konzolne aplikacije u Visual Studio okruženju. Potrebno je napisati jednostavnu aplikaciju koja će na standardni izlaz napisati poruku „Početak RPR kursa“.
2. Navedite ulogu i način imenovanja identifikatora u C# jeziku.
3. Objasnite šta predstavlja varijabla i način deklaracije varijabli u C#-u.
4. Nabrojite i navedite osnovne karakteristike vrijednosnih tipova podataka u C#-u.
5. Koji operatori se primjenjuju na vrijednosne tipove podataka?
6. Koji rezultat daju iskazi navedeni ispod:

```
Console.WriteLine(5.0/2.0);
Console.WriteLine(5/2);
Console.WriteLine(5/2.0);
Console.WriteLine (9 % 2) ;
Console.WriteLine (7.0 % 2.4) ;
```

7. Iskaz `brojac++` napišite u prefiksnoj notaciji.
8. Na varijablu `brojac` primijeniti dekrement operator u postfiksnoj i prefiksnoj formi.
9. Napišite iskaz pridruživanja, izraz poređenja, izraz koji sadrži logičku negaciju, izraz koji sadrži logičko i, izraz koji sadrži logičko ili.
10. Objasnite iskaze:

```
bool validProcenat, invalidProcenat;
validProcenat = (procenat >= 0) && (procenat <= 100);
invalidProcenat = (procenat < 0) || (procenat > 100);
```
11. Koji tipovi konverzija postoje u C#-u?
12. Koja pravila se koriste prilikom implicitne konverzije podataka?
13. Kako se vrši eksplicitna konverzija podataka?
14. Objasnite i navedite primjer `enum` tipa podataka.
15. Objasnite strukturu kao tip podataka i navedite primjer strukture.
16. Objasnite način smještanja podataka u memoriju za referencne tipove podataka.
17. Navedite C# referencne tipove podataka.
18. Objasnite `object` tip podataka, *boxing* i *unboxing*.
19. Objasnite značenje `dynamic` ključne riječi.

RAZVOJ PROGRAMSKIH RJEŠENJA

20. Navedite osnovne karakteristike string tipa podataka i operatore koji se primjenjuju nad ovim tipom podataka.
21. Objasnite pointer tip podataka.
22. Navedite vrste nizova u C# jeziku. Navedite i primjere deklaracije.
23. Objasnite značenje ključne riječi `var`. Da li je ispravna deklaracija `var kurs;` ?
24. Koja je uloga `checked` i `unchecked` ključnih riječi u C#-u.
25. Objasnite *boxing* i *unboxing* u C#-u.
26. Navedite osnovne iskaze selekcije u C#. Za svaki iskaz selekcije radi ilustracije sintakse napišite po jedan primjer.
27. Objasnite sljedeći dio koda:

```
int sekunda= 59;
        int minuta = 0;
        if (sekunda == 59)
        {
            sekunda = 0;
            minuta++;
        }
        else
            sekunda++;

Console.WriteLine("Minute {0} Sekunde {1}", minuta, sekunda);
```

28. Navedite osnovne strukture iteracije (petlje) u C#-u. Za svaki iskaz iteracije radi ilustracije sintakse napišite po jedan primjer.
29. Sve primjere koji su služili kao ilustracija uradite sami kroz okruženje.
30. Prilikom rada za pitanje 29 napravite namjerno sintaksne greške i čitajte poruke kompjerala.

Poglavlje 3: Osnovni koncepti klase i dekompozicija programa

U okviru ovog poglavlja prikazuje se način izražavanja klase u C#-u i način dekompozicije programa u odvojene cjeline kroz tematske jedinice:

1. Klasa: osnovni koncepti i izražavanje u C#-u
2. Dekompozicija programa

3.1. Klasa: osnovni koncepti i izražavanje u C#-u

Klasa je korisnički definirani tip podataka koji se koristi da se opišu različiti tipovi objekata. Klasa opisuje/odražava stanje (pomoću atributa) i ponašanje (pomoću metoda) objekata koji su istog tipa.

3.1.1. Definicija klase

U C# programskom jeziku prilikom definicije klase navode se sljedeći elementi:

```
modifikator_klase class NazivKlase ( generički parametri)  
{  
    Članovi klase  
}
```

Postoji više modifikatora pristupa klasi. Neki od njih su: `public`, `internal`, `abstract`, `sealed`, `static`, `unsafe`, `partial`. Značenje pojedinih specifičnih modifikatora pristupa za klase se navodi u nastavku ovog poglavlja i u sljedećim poglavljkima. Članovi klase su: metode, osobine (*properties*), polja, konstruktori, događaji, destruktori, indekseri. I na članove klase se primjenjuju različiti specifikatori pristupa. U ovoj sekciji se pominju samo `private` i `public` specifikatori pristupa za članove klase. Ukoliko nije potrebna restrikcija za pristup koristi se `public` specifikator pristupa. Ukoliko je potrebno da se vrši restrikcija pristupa tada se koristi `private` specifikator koji se navodi uz člana klase. Privatni članovi su vidljivi samo unutar tijela klase (ili strukture) u kojoj su deklarirani. Ako nije eksplicitno naveden pristup podrazumijevani pristup članu klase je privatni. Pretežno su svi članovi klase koji se odnose na polja privatni, a metode članice klase su uglavnom javne (`public`). Javne metode klase čine interfejs klase. Metode koje su privatne su uglavnom uslužne metode i koriste ih javne metode za svoju implementaciju.

RAZVOJ PROGRAMSKIH RJEŠENJA

U okviru ove sekcije navode se različite korisnički definirane klase da bi se što više razmišljalo objektno-orientirano i da bi čitaoci udžbenika razmišljali o objektima, klasama, sami dodavali nove atribute i metode. Kod 3.1. predstavlja definiciju klase `Vrijeme` koja sadrži tri atributa za sat, minute, sekundu koji su označeni kao privatni i dvije javne (`public`) metode. Metoda `setVrijeme` postavlja sat, minutu, sekundu na dozvoljenu vrijednost ili 0, a metoda `konvertujStringUFormatHHMMSS()` prikazuje vrijeme na standardni izlaz u formatu HH:MM:SS.

```
public class Vrijeme // class ključna riječ, Vrijeme naziv klase, u {} članice klase
{
    private int sat;
    private int minuta;
    private int sekunda;

    // postavlja vrijeme (sat, minuta, godina) na dozvoljene vrijednosti opsega ili 0
    public void SetVrijeme(int h, int m, int s)
    {
        sat = ((h >= 0 && h < 24) ? h : 0);
        minuta = ((m >= 0 && m < 60) ? m : 0);
        sekunda = ((s >= 0 && s < 60) ? s : 0);
    }

    // pretvara string u format vremena (HH:MM:SS)
    public string konvertujStringUFormatHHMMSS()
    {
        return string.Format("{0:D2}:{1:D2}:{2:D2}",
            sat, minuta, sekunda);
    }
} // kraj definicije klase Vrijeme
```

Modifikatori (specifikatori) pristupa članovima klase:
private, public, internal, protected, override,
partial

metoda članica klase

metoda članica klase

Kod 3.1: Klasa sa atributima i metodama

Prilikom definiranja klasa bitna je konstruktor metoda za koju važe obilježja:

- Konstruktor klase se koristi za inicijalizaciju atributa (polja) klase.
- Konstruktor klase pokreće inicijalni kod prilikom instanciranja objekta klase.
- Naziv konstruktora je isti kao naziv klase.
- Konstruktor je metoda koja ne vraća vrijednost.
- Konstruktor klase je obično `public`, ali ne mora uvijek biti (ne-`public` konstruktor može se koristiti za kontrolu kreiranja instanci u okviru poziva `static` metoda).
- Konstruktori se mogu preklapati.

- Konstruktor klase** pokreće inicijalni kod prilikom instanciranja klase.
- Naziv konstruktora je isti kao naziv klase.
- Konstruktor ne vraća povratnu vrijednost.

```
public class KonvertorJedinica
{
    int procenat ;
    public KonvertorJedinica(int jedinicaProcenat) { procenat = jedinicaProcenat; }
    public int Konvertor(int jedinica) { return jedinica * procenat; } // Metoda
}
```

Kod 3.2: Klasa KonvertorJedinica sa konstruktorem

Objekat tipa klase KonvertorJedinica se može instancirati korištenjem operatora new na sljedeći način:

```
KonvertorJedinica konvertor = new KonvertorJedinica(10);
```

Gornjim iskazominstanciran je objekat pod nazivom konvertor tipa klase KonvertorJedinica, za koji je atribut procenat postavljen na 10.

Slijedi primjer C# programa (kod 3.3) unutar kojeg se definira klasa Udzbenik na osnovu koje se instancira objekat preporuceniUdzbenik u okviru klase Program. Nakon toga se preko instanciranog objekata poziva metoda PrikaziPoruku.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;    1
namespace Udzbenik 2
{
    class Udzbenik
    {
        // metoda za prikazivanje poruke
        public void PrikaziPoruku()
        {
            Console.WriteLine(" Izbor udžbenika!");
        }
    } // kraj klase Udzbenik
    class Program 4
    {
        static void Main(string[] args)
        {
            // kreira instancu (objekat) Udzbenik klase
            Udzbenik preporuceniUdzbenik = new Udzbenik(); 5

            // poziva metodu PrikaziPoruku objekta preporuceniUdzbenik
            preporuceniUdzbenik.PrikaziPoruku();
        }
    }
}
```

Kod 3.3: Definicija i upotreba klase u okviru C# projekta

Objašnjenje tačaka označenih u prikazanom programu 3.3:

1. *Namespace* (imenovani prostor) u koji se smještaju definicije za logički povezane tipove podataka i funkcionalnosti, mogu se ugnježdavati; sa direktivom `using` se mogu koristiti u više programske cijelina.
2. Definicija klase `Udzbenik`.
3. Klasa `Program` – automatski se kreira prilikom kreiranja projekta/aplikacije.
4. Metoda `Main` unutar klase `Program` – početna tačka izvršavanja programa.
5. Instanciranje objekta `preporuceniUdzbenik` klase `Udzbenik`.
6. Poziv metode `prikaziPoruku` klase `Udzbenik` za objekat/instancu `preporuceniUdzbenik`.

Dio koda naveden pod koracima 4-5-6 predstavlja test klase. Testiranje klase se ostvaruje pozivom svih metoda klase.

Metode klase mogu imati i parametre. Klasa `Udzbenik` u primjeru ispod (kod 3.4) sadrži metodu `PrikaziPoruku` sa parametrom.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System ;  
  
namespace udzbenik  
{  
    class Udzbenik  
    {  
        public void PrikaziPoruku(string naziv)  
        {  
            Console.WriteLine("Udzbenik za \n{0}!", naziv);  
        }  
    } // kraj definicije klase Udzbenik  
  
    class UdzbenikTest  
    {  
        static void Main(string[] args)  
        {  
  
            // kreira objekt preporuceniUdzbenik tipa klase Udzbenik  
            Udzbenik preporuceniUdzbenik = new Udzbenik();  
  
            Console.WriteLine("Unesi naziv udzbenika: ");  
            string nazivUdzbenika = Console.ReadLine(); // čita liniju teksta  
  
            preporuceniUdzbenik.PrikaziPoruku(nazivUdzbenika) ;  
        } // kraj metode Main  
    } // kraj definicije klase UdzbenikTest  
}
```

Metoda klase sa parametrom

Poziv metode sa parametrom

Kod 3.4: Primjer metode sa parametrom

Predaja parametara C# metodi može se izvršiti kao: predaja po vrijednosti i predaja po referenci.

Predaja parametara po vrijednosti znači da se metodi predaje vrijednost varijable. Ta vrijednost se smješta na drugu memorijsku lokaciju. Metoda u ovom slučaju ne može da utiče na memorijsku lokaciju parametra. Ovaj način predaje parametara se podrazumijeva i prilikom predaje ne navodi se nikakav specifikator. Ilustriran je kodom 3.4. Predaja parametara po referenci se obavlja tako da se preda memorijska adresa parametra. To omogućava da se vrijednost parametra unutar metode promijeni i da se nakon povratka iz metode nastavi da radi sa promijenjenom vrijednošću varijable koja je predana kao parametar. Za predaju parametara po referenci koristiti se `ref` ili `out` modifikator pristupa.

Kod 3.5 sadrži dvije `static` metode unutar klase, pa se mogu pozvati bez instanciranja objekta. Prvoj metodi `MetodaPrimaVrijednost` predaje se parametar po vrijednosti a drugoj `MetodaPrimaReferencu` predaje se parametar po referenci.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
namespace PredajaParametara
{
    class Program
    {
        static void MetodaPrimaVrijednost(int p)
        {
            p = p + 1;
            Console.WriteLine(p); // vrijednost varijable p je 9
        }

        static void MetodaPrimaReferencu(ref int p)
        {
            p = p + 1; // vrijednost varijable na koju pokazuje p (na x iz maina) je 9
            Console.WriteLine(p);
        }

        static void Main(string[] args)
        {

            int x = 8;
            MetodaPrimaVrijednost(x); // predaje se vrijednost 8,
                                       // na drugu memorijsku lokaciju
            Console.WriteLine(x); // x je i dalje 8

            MetodaPrimaReferencu(ref x); // predaje se adresa varijable x,
                                         // na kojoj je upisana vrijednost 8

            Console.WriteLine(x); // x je sada 9
        }
    }
}
```

Kod 3.5: Predaja parametara po vrijednosti i po referenci

U primjeru iznad korišten je `ref` modifikator. Pored toga koriste se i dodatni modifikatori koji daju specifično značenje. Neki od njih su `out`, `params`. Modifikator `out` sličan je `ref` modifikatoru, ali ne zahtjeva da se varijabli parametra dodijeli vrijednost prije predaje varijable-parametra metodi. Varijabli se dodjeljuje vrijednost nakon povratka iz metode.

`Params` modifikator – specificira se uz nizove i naznačava da metoda može prihvati bilo koji broj parametara koji su tipa argumenta uz koji je naznačen ovaj modifikator.

```
static int SumaNiza(params int[] cjelobrojniNiz)
{
    int suma = 0;
    for (int i = 0; i < cjelobrojniNiz.Length; i++)
        suma += cjelobrojniNiz[i];
    return suma;
}
```

Kod 3.6: Metoda sa `param` modifikatorom

RAZVOJ PROGRAMSKIH RJEŠENJA

3.1.2. Osobine (*property*) klase

Pristup privatnim podacima klase potrebno je pažljivo kontrolirati sa metodama članicama klase. Za čitanje privatnih podataka klase koriste se *get* metode, dok za modifikaciju privatnih podataka klase koristi se *set* metode. Blok koda koji obezbeđuje *get* i/ili *set* metode za privatno polje klase naziva se *osobina (property)*. Preporučuje se da se za svako privatno polje klase definira *property* osobina. Kod 3.7. sadrži definiranu *property* osobinu za privatno polje nazivUdzbenika.

```
class Udzbenik
{
    private string nazivUdzbenika; Polje/atribut za koje je napisana osobina-property

    // konstruktor inicijalizira nazivUdzbenika
    public Udzbenik(string naziv)
    {
        NazivUdzbenika = naziv; // inicijalizira nazivUdzbenika korištenjem property NazivUdzbenika
    }

    // property get i set za naziv udžbenika
    public string NazivUdzbenika
    {
        get
        {
            return nazivUdzbenika;
        }
        set
        {
            nazivUdzbenika = value;
        }
    } // kraj property NazivUdzbenika

    public void PrikaziPoruku()
    {
        Console.WriteLine("Udzbenik za \n{0}! ", NazivUdzbenika);
    }
} PROPERTY // kraj klase Udzbenik
```

Kod 3.7: Klasa sa *property* osobinom

RAZVOJ PROGRAMSKIH RJEŠENJA

Kod 3.8 prikazuje način kako se može instancirati objekat klase koja sadrži implementirane osobine.

```
class UdzbenikTest
{
    static void Main(string[] args)
    {

        // kreiranje i postavljanje inicijalnih vrijednosti za Udzbenik objekte
        Udzbenik udzbenikProgramiranjeOsnovni = new Udzbenik("C++ - Tehnike
Programiranja ");
        Udzbenik udzbenikProgramiranjeDodatni = new Udzbenik("C# - Razvoj Programske
Rjesenja ");

        // prikaz inicijalnih vrijednosti za naziv udzbenika
        Console.WriteLine("{0} ", udzbenikProgramiranjeOsnovni.NazivUdzbenika);
        Console.WriteLine("{0} ", udzbenikProgramiranjeDodatni.NazivUdzbenika);
    }
} // kraj klase UdzbenikTest
```

Instanciranje objekta, poziv
konstruktora koji inicijalizira polje
nazivUdzbenika preko *PROPERTY*
NazivUdzbenika

Kod 3.8: Instanciranje objekta klase koja sadrži *property*

Prilikom definicije klase mogu se navesti osobine (*properties*) odnosno *get* i/ili *set* metode koje čitaju i pišu u privatno polje objekta i na način prikazan u sklopu koda 3.9.

```
// Klasa sa automatski navedenom property Ime
public class Osoba
{

    public string Ime { get; set; }
}
```

Kod 3.9: Klasa sa automatskim navedenim *property* – *get* i *set* metodom za atribut *Ime*

Klasa *Osoba* prikazana sa kodom 3.9 sadrži osobinu (*property*) *Ime* za koju kompjajler automatski generira implementacijski kod.

3.1.3. Indekser

Za nizove, liste podataka koristi se *indekseri* koji omogućavaju indeksirani pristup listi elemenata. *Indekseri* su slični sa osobinama (*properties*), ali im se pristupa sa indeksom umjesto sa imenom osobine.

Opšta forma *indeksera* je:

```
modifikator_pristupa povratniTip this[ indeks1Tip naziv1, indeks2Tip naziv2, ... ]  
{  
    get { }  
    set { }  
}
```

Slijedi kod 3.10 sa primjerom definicije i upotrebe klase sa *indekserom*.

```
using System;  
namespace IndexerPrimjer  
{  
    class TemperaturaDan  
    {  
        float[] dnevnaTemperatura = new float[7] { 56.2F, 56.7F, 56.5F, 56.9F, 58.8F,  
                                                    61.3F, 65.9F };  
  
        public int Duzina  
        {  
            get { return dnevnaTemperatura.Length; }  
        }  
        // Indekser deklaracija  
        // Ako je indeks izvan ranga, izaziva se izuzetak  
        public float this[int indeks]  
        {  
            get  
            {  
                return dnevnaTemperatura[indeks];  
            }  
            set  
            {  
                dnevnaTemperatura[indeks] = value;  
            }  
        }  
    }  
  
    // nastavak na sljedećoj stranici -
```



RAZVOJ PROGRAMSKIH RJEŠENJA

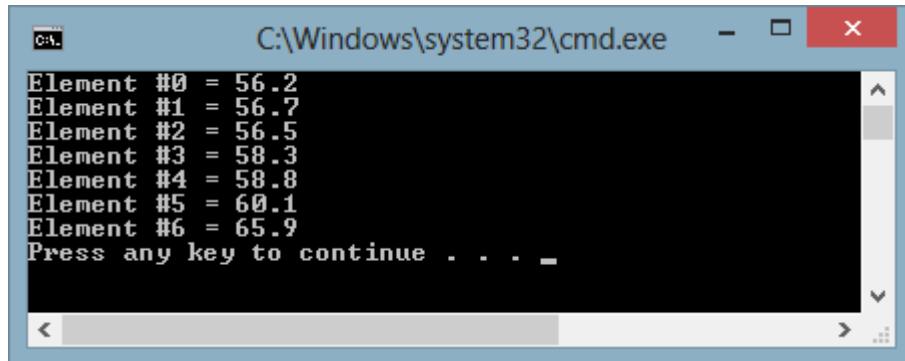
```
class GlavnaKlasa
{
    static void Main()
    {
        TemperaturaDan temperatura = new TemperaturaDan();

        // koristi indekser set pristup
        temperatura[3] = 58.3F;
        temperatura[5] = 60.1F;

        // Koristi indekser get pristup
        for (int i = 0; i < 7; i++)
        {
            System.Console.WriteLine("Element #{0} = {1}", i, temperatura[i]);
        }
    }
}
```

Kod 3.10: Primjer definicije i upotrebe klase sa indekserom

Izvršavanjem koda 3.10. dobija se rezultat:



```
Element #0 = 56.2
Element #1 = 56.7
Element #2 = 56.5
Element #3 = 58.3
Element #4 = 58.8
Element #5 = 60.1
Element #6 = 65.9
Press any key to continue . . .
```

3.1.4. Dodatne osobine članova klase za upravljanje privilegijama

Važnije dodatne osobine klase pomoću kojih se dodjeljuju prava pristupa članovima klase su:

static i readonly.

static polje klase

Polja (atributi) klase koja su do sada uvedena u primjerima klasa su polja koja pripadaju instanci klase odnosno objektu klase. Svaka instanca klase ima svoje kopije tih polja. Često se ova polja nazivaju i *instance* (polja pojedinačne instance). Ukoliko je potrebno da se jedno polje dijeli između više objekata neke klase tada se prilikom deklaracije tog polja navodi specifikator

RAZVOJ PROGRAMSKIH RJEŠENJA

pristupa static. Static polja pripadaju klasi, instance te klase nemaju kopije tih polja, svi objekti te klase pristupaju istoj memorijskoj lokaciji.

Slijedi primjer klase, kod 3.11, sa *instance* i *static* poljem. Klasa broji koliko se instanci klase PredmetIISemestar instancira. U okviru donjeg programa polje BrojPredmeta je static, njegova vrijednost je inicijalno postavljena na nulu. Nakon instanciranja prvog objekta (predmet1) povećava se njegova vrijednost na 1, a nakon instanciranja drugog objekta (predmet2) na 2.

```
using System;

namespace StaticPoljeKlase
{
    public class PredmetIISemestar
    {
        public string naziv;           // Instance polje
        public static int BrojPredmeta; // Static polje
        public PredmetIISemestar (string n)          // Konstruktor
        {
            naziv = n;                // dodjela vrijednosti instance polju
            BrojPredmeta = BrojPredmeta + 1; // Inkrement static polja BrojPredmeta
        }
    }

    class Test
    {
        static void Main()
        {
            PredmetIISemestar predmet1 = new PredmetIISemestar ("RPR");
            PredmetIISemestar predmet2 = new PredmetIISemestar ("OOAD");
            Console.WriteLine(PredmetIISemestar.BrojPredmeta);

        }
    }
}
```

Kod 3.11: Klasa sa static poljem

Pored polja i metoda unutar klase može biti static. Static metoda klase ne može koristiti polja klase koja nisu static. Static metoda se ne može pozvati sa instancom klase, već se poziva navođenjem imena klase. Pristup static članovima klase je veoma restriktivan. Neke od restrikcija su prikazane sa kodom 3.12.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
namespace StaticRestrikcije
{
    class TestStatic
    {
        int x;

        static int y; //static polje

        void metoda()
        {
            x = 1;          // Ured u je, isto kao this.x
            y = 1;          // Ured u je, može se pristupiti static polju preko imena
        }
        static void metodaStatic()
        {
            x = 1;          // Greška, ne može se pristupiti polju x - jer je metoda static
            y = 1;          // Ured u, y je static polje
        }

        static void Main()
        {
            TestStatic t = new TestStatic();
            t.x = 1;          // Ured u
            t.y = 1;          // Greška, ne može se pristupiti static članu preko instance
            TestStatic.x = 1; // Greška, ne može se pristupiti instance članu preko tipa
            TestStatic.y = 1; // Ured u
            t.metodaStatic(); // Greška, ne može se pristupiti static metodi preko instance
            metodaStatic();  // Ured u, Poziv static metode
        }
    }
}
```

Kod 3.12: Restrikcije pristupa static članovima klase

readonly specifikator

Specifikator `readonly` koristi se za kontrolu privilegija pristupa varijablama. Varijabla (instanca objekta) ne može se mijenjati ako je označena kao `readonly` (samo za čitanje). Preporučuje se da se takve varijable imenuju velikim slovima. Mogu se inicijalizirati pri deklaraciji ili preko konstruktora, mogu se koristiti u iskazima na lijevoj strani.

```
// deklaracija readonly varijable (neinicijalizirana)
private readonly int BROJAC;

// konstruktor inicijalizira readonly instancu varijable BROJAC
public PovecajBrojac(int vrijednost)
{
    BROJAC = vrijednost; // inicijalizira readonly varijablu
}

// readonly varijabla BROJAC se može koristiti u iskazima
ukupno += BROJAC;
```

3.2. Dekompozicija

Prilikom pisanja programskih rješenja bitna je podjela programskog koda u zasebne, smislene cjeline. Ta podjela naziva se dekompozicija programa. Jedna od metoda dekompozicije je odvajanje interfejsa klase od implementacije. Npr., C++ odvaja definiciju klase u .h (header) fajl, implementaciju u .cpp fajl, a test klase u poseban .cpp program.

3.2.1. Dekompozicija projekta u C#-u

Dekompozicija projekta vezanog za programsko rješenje u C#-u se odvija na sljedeći način:

- Definicija klase se smješta u jedan .cs fajl.
- Test klase se smješta u drugi .cs fajl.

Pri tome, svi .cs fajlovi mogu biti u jednom projektu. Fajlovi koji se odnose na definiciju klase često se smještaju u zasebne biblioteke.

Kod 3.13. se odnosi na definiciju klase Racun. Klasa Racun nudi polje stanje, osobinu za čitanje i pisanje vrijednosti stanja i metodu za dodavanje novog iznosa na već postojeće stanje. Nakon kreiranja projekta RacunDekompozicija dodaje se novi fajl Racun.cs (u .NET okruženju dodaje se Visual C# Item Class) u kojem se definira klasa Racun.

```
using System;
namespace RacunDekompozicija
{
    public class Racun
    {
        private decimal stanje;

        public Racun( decimal inicialnoStanje )           // konstruktor
        {
            Stanje = inicialnoStanje; // postavlja stanje korištenjem property
        }
        public void dodajIznos( decimal iznos )          // metoda dodavanje iznosa na račun
        {
            Stanje = Stanje + iznos; // dodaje iznos na stanje
        }
        public decimal Stanje    // property za get i set stanja na račun
        {
            get { return stanje; }
            set {
                if ( value >= 0 ) stanje = value;
            }
        }
    } // kraj klase Racun
}
```

Kod 3.13: Klasa Racun u posebnom fajlu Racun.cs

RAZVOJ PROGRAMSKIH RJEŠENJA

Klasa `Racun` se može koristiti od strane drugih klasa koje su fizički smještene u drugom fajlu. Naprimjer, u fajlu `KlijentKlase` je smještena klasa `KlijentKlase`, unutar koje se instancira objekat klase `Racun` i za taj objekat se vrši promjena stanja. Kod 3.14. se odnosi na fajl `KlijentKlase`. Ovaj kod se nalazi u okviru istog projekta `RacunDekompozicija`.

```
using System;

namespace RacunDekompozicija
{
    class KlijentKlase
    {
        static void Main(string[] args)
        {

            Racun racunMoj = new Racun(500.00M);

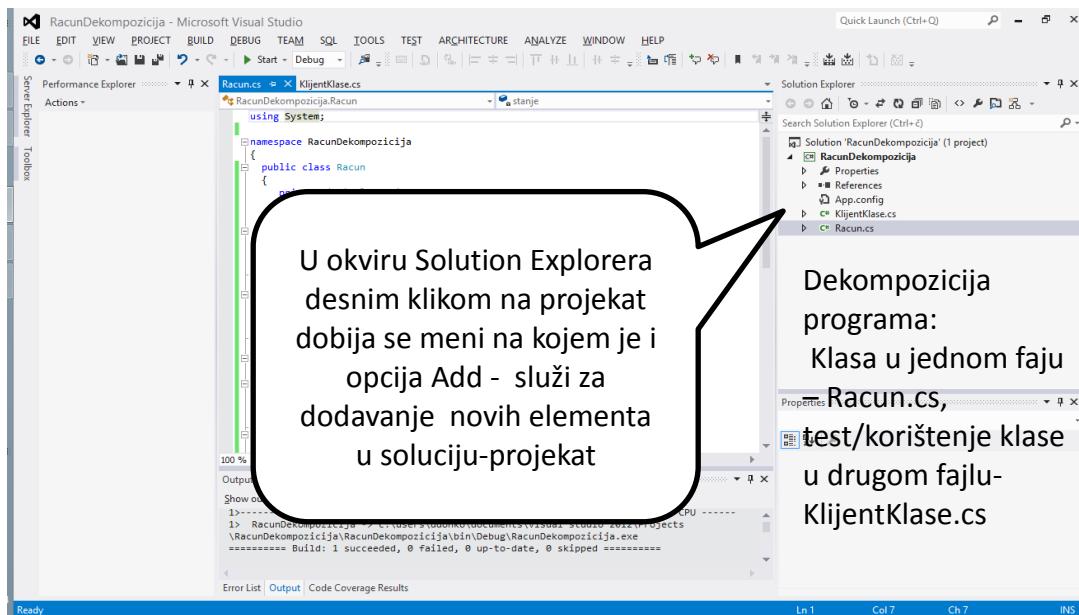
            // prikazuje inicijalno stanje objekta korištenjem osobine Stanje
            Console.WriteLine("Stanje računa {0}KM", racunMoj.Stanje);

            racunMoj.dodajIznos(600.00M);

            Console.WriteLine("Stanje na racunu {0}KM ", racunMoj.Stanje);
        }
    }
}
```

Kod 3.14: Klasa `KlijentKlase` u fajlu `KlijentKlase.cs`

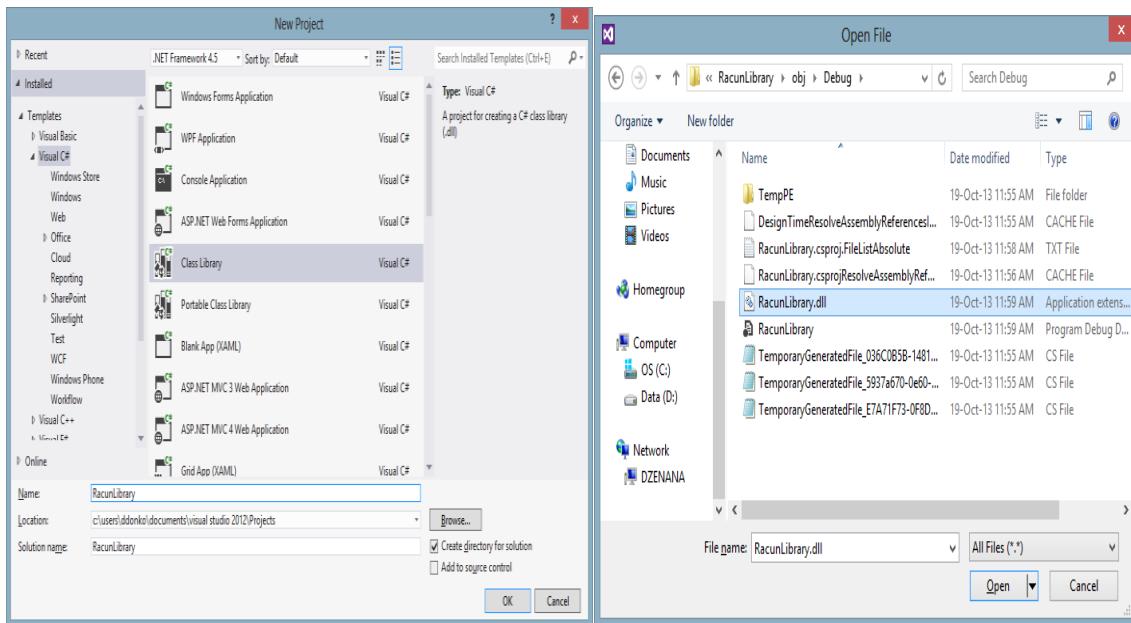
Slika 3.1 prikazuje strukturu projekta `RacunDekompozicija` kroz Visual Studio razvojno okruženje.



Slika 3.1: Kreiranje projekta `RacunDekompozicija`

3.2.2. Smještanje klase u biblioteku

Često se definicija i implementacija klase smješta u biblioteku. Okruženja uglavnom nude mogućnost kreiranja biblioteka klasa (ClassLibrary). Nakon smještanja definicije klase u fajl biblioteke, formira se `dll` (dynamic link library). Formirani `dll` se može koristiti u drugim projektima sa `using` direktivom. U okviru Visual Studio formiranje biblioteke je prikazano na slici 3.2a. U formirani fajl smješta se definicija klase `Racun`. Nakon toga se izvršava kompilacija (*build*) čiji proizvod je `dll` fajl smješten u folder projekta (slika 3.2b).



Slika 3.2: a)Kreiranje biblioteke -`dll` b)Kreirani `dll`

Da bi se biblioteka (`dll`) mogla koristiti potrebno je izvršiti njeno povezivanje sa projektom (koristi se Reference Manager-Add Reference opcija). Nakon toga može se koristiti unutar projekta, sa `using` direktivom (kod 3.15).

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using RacunLibrary; ←  
  
using System;  
namespace usingLibraryRacun  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
  
            Racun racunMoj = new Racun(500.00M);  
  
            // prikazuje inicijalno stanje objekta korištenjem property Stanje  
            Console.WriteLine("Stanje računa {0:C} ",racunMoj.Stanje );  
  
            racunMoj.dodajIznos(600.00M);  
  
            Console.WriteLine("Stanje na računu {0:C} ",racunMoj.Stanje );  
        }  
    }  
}
```

Sa using direktivom koristi se biblioteka RacunLibrary

Kod 3.15: Korištenje klase iz biblioteke sa using direktivom

3.2.3. Kompozicija klase i dekompozicija programa

Član-polje neke klase može biti tipa neke druge klase. U tom slučaju radi se o kompoziciji klase. Slijedi ilustracija kompozicije klase. Naprimjer, klasa `Uposlenik` sadrži pored ostalih članova i polja koja se odnose na datum rođenja i datum uposlenja. Ta polja mogu biti članovi korisnički definirane klase `Datum` koja sadrži polja za dan, mjesec, godinu i metode `set` i `get` za ta polja.

Organizacija koda u skladu sa pravilima dekompozicije bi bila:

1. `Datum.cs` (definicija klase `Datum`)
2. `Uposlenik.cs` (definicija klase `Uposlenik`)
3. `ProgramTest.cs` (test/upotreba klase `Uposlenik`)

Kodovi 3.16-3.18 ilustriraju ovaj koncept. Kod 3.16 sadrži definiciju i implementaciju klase `Datum`, kod 3.17 klase `Uposlenik` koja koristi klasu `Datum`, a kod 3.18 implementaciju glavnog programa koji koristi klasu `Uposlenik`.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
namespace KompozicijaKlasa
{
    // Definicija klase Datum
    public class Datum
    {
        private int mjesec;
        private int dan;
        private int godina;

        // konstruktor: koristi property Mjesec, Dan, Godina za validaciju vrijednosti za
        // mjesec, dan, godinu
        public Datum( int pMjesec, int pDan, int pGodina )
        {
            Mjesec = pMjesec;
            Godina = pGodina;
            Dan = pDan;
            Console.WriteLine("Datum - konstruktor {0} ", this );
        }
        public int Godina      // property - get i set godine
        {
            get
            {
                return godina;
            }
            private set //   set - ne može se pristupiti izvan klase
            {
                godina = value;
            }
        } // kraj property Godina
        public int Mjesec      // property get i set mjeseca
        {
            get
            {
                return mjesec;
            }
            private set
            {
                if ( value > 0 && value <= 12 )
                    mjesec = value;
                else
                {
                    Console.WriteLine("Ne validan mjesec ({0}) postavi na 1. ", value );
                    mjesec = 1;
                }
            }
        } // kraj property Mjesec

        // property get i set dana - izostavljena implementacija
        public override string ToString()
        {
            return string.Format("{0}/{1}/{2}", Mjesec, Dan, Godina );
        }
    } // kraj klase Datum
}
```

Kod 3.16: Definicija klase Datum u fajlu Datum.cs

RAZVOJ PROGRAMSKIH RJEŠENJA

Klasu `Datum` koriste druge klase za postavljanje tipa pojedinim članovima. Unutar koda 3.17 klasa `Uposlenik` koristi klasu `Datum`. Ovaj kod je dio posebnog fajla `Uposlenik.cs`.

```
using System;
namespace KompozicijaKlase
{
    public class Uposlenik
    {
        private string ime;
        private string prezime;
        private Datum datumRodenja; // polje tipa klase Datum
        private Datum datumUposlenja; // polje tipa klase Datum

        // konstruktor za inicijalizaciju polja: ime, prezime, datumRodenja, datumUposlenja
        public Uposlenik( string pime, string pprezime,
                           Datum pdatumRodenja, Datum pdatumUposlenja )
        {
            ime = pime;
            prezime = pprezime;
            datumRodenja = pdatumRodenja;
            datumUposlenja = pdatumUposlenja;
        }

        public override string ToString()
        {
            return string.Format("{0}, {1} Roden: {2} Uposlen: {3} ",
                prezime, ime, datumRodenja, datumUposlenja);
        }
    } // kraj klase Uposlenik
}
```

Kod 3.17: Tipovi pojedinih članova klase `Uposlenik` su tipa klase `Datum`, klasa `Uposlenik` je u fajlu `Uposlenik.cs`

Upotreba klase `Uposlenik` se pokazuje sa kodom 3.18. Prilikom poziva konstruktora navode se parametri za sve članove vodeći računa o članovima klase koji su tipa neke druge klase (u ovom slučaju `Datum`).

```
using System;
namespace KompozicijaKlase
{
    class ProgramTest
    {
        static void Main(string[] args)
        {
            Datum rodenje = new Datum( 8, 6, 1984 );
            Datum uposlenje = new Datum( 3, 12, 2001 );
            Uposlenik uposlenik = new Uposlenik("Vedad", "Vedadovic ", rodenje, uposlenje );
            Console.WriteLine(uposlenik);
        }
    } // kraj klase ProgramTest
}
```

Kod 3.18: Test i upotreba klase `Uposlenik`

3.2.4. Modifikatori pristupa klasi

Na početku poglavlja je navedeno da se modifikatori pristupa koriste za specifikaciju načina pristupa članu klase ili klasi. Klase navedene u primjerima kroz poglavlje uglavnom su bile sa specifikatorom pristupa `public`. Ako je klasa označena sa `public` specifikatorom pristupa tada je pristup klasi omogućen iz asemblija (*assembly*) u kojem je klasa definirana ali i iz drugih asemblija. U .NET asembli se odnosi na fizičko grupiranje koda. Asemblji je kolekcija tipova i resursa kojim se omogućavaju logičke funkcionalnosti. Svaki asemblji se smješta kao `.exe` ili `.dll` fajl. Moguće je kreirati asembli koji je smješten u više fizičkih fajlova, međutim često nema potrebe za tim. .NET *framework* koristi asemblji i za sljedeće namjene: sigurnost, identifikaciju tipova, vidokrug referenci, određivanje verzija, raspoređivanje (*deployment*).

Ukoliko pristup klasi treba ograničiti na tekući asemblji koristi se specifikator `internal`. Ukoliko nije naveden specifikator pristupa za klasu tada se podrazumijeva `internal` pristup, odnosno klasa je vidljiva samo unutar asemblija unutar kojeg je i definirana. Sa `protected internal` pristup klasi je ograničen na tekući asemblji ili na tipove izvedene iz klase tog asemblija.

Pristup označen ključnom riječi `internal` je modifikator pristupa i za članove klase.

```
public class OsnovnaKlasa
{
    internal static int x = 0; // x vidljivo unutar tekućeg asemblija
}
```

partial definicija klase

Definicija klase se može implementirati u više različitih fajlova koji su u istom asembliju. U tom slučaju koristi se `partial` specifikator pristupa. Navodi se uz svaki dio definicije klase. Svi dijelovi parcijalnih klasa moraju biti u istom imenovanom prostoru. Svi dijelovi klase moraju imati isti naziv. Kod 3.19 prikazuje dio implementacije parcijalne klase `Racun`. Drugi dio implementacije koji se odnosi na implementaciju osobine može biti fizički u drugom fajlu.

```
public partial class Racun
{
    // property za get i set polja stanja
    public decimal Stanje
    {
        get
        {
            return stanje;
        }
        set
        {
            // provjerava da li je vrijednost veća ili jednaka 0 i ako jeste mijenja
            if (value >= 0)
                stanje = value;
        }
    } // kraj property Stanje
} // kraj prvog dijela implementacije klase Racun
```

Kod 3.19: Dio implementacije klase Racun u fajlu DioRacun1.cs

Parcijalni način implementacije klase je koristan prilikom definiranja složenijih klasa i dodjeljivanja zadataka razvojnog timu. Parcijalne klase su korisne i za odvajanje ulazno-izlaznih operacija, korisničkog interfejsa od poslovne logike.

Rezime:

U okviru poglavlja navedeni su osnovni koncepti klase i način izražavanja tih koncepata u C# jeziku. Posebno su naglašeni koncepti koje nisu podržani u drugim srodnim jezicima. Objasnjen je i način dekompozicije projekta. Dat je i primjer kompozicije klasa i za taj slučaj način organizacije projekta.

Pitanja za ponavljanje i samostalni rad:

1. Objasnite šta se opisuje sa klasom kao tipom podataka.
2. Navedite i objasnite modifikatore pristupa članovima klase.
3. Objasnite ulogu konstruktora klase.
4. Objasnite ulogu osobina (*property*) klase.
5. Navedite primjere dobro definiranih klasa.
6. Navedite i objasnite način predaje parametara i modifikatore koji se koriste prilikom predaje parametara.
7. Objasnite namjenu `static` članova klase.

RAZVOJ PROGRAMSKIH RJEŠENJA

8. Objasnite namjenu `readonly` članova klase.
9. Objasnite ulogu i način definiranja indeksera.
10. Objasnite način dekompozicije programa u C#-u.
11. Šta je `.dll`, objasnite njegov način kreiranja i korištenja?
12. Šta predstavlja asembli (*assembly*) u .NET okruženju?
13. Objasnite kompoziciju klasa.
14. Navesti primjere klasa koji ilustriraju kompoziciju i objasnite kako se vrši u tom slučaju dekompozicija koda.
15. Navedite i objasnite značenje `public` i `internal` modifikatora pristupa za klase.
16. Objasnite kako se može izvršiti parcijalna definicija klasa.
17. Analizirajte kod, pročitajte grešku, objasnite zašto je nastala i ispravite grešku.

```
using System;
namespace udzbenik
{
    class Udzbenik
    {
        public void PrikaziPoruku(string naziv)
        {
            Console.WriteLine("Udzbenik za \n{0}!", naziv);
        }
    } // kraj klase Udzbenik

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

            // kreira objekat preporuceniUdzbenik tipa klase Udzbenik
            Udzbenik preporuceniUdzbenik = new Udzbenik();

            Console.WriteLine("Unesi naziv udzbenika:");
            string nazivUdzbenika = Console.ReadLine(); // čita liniju teksta

            Udzbenik.PrikaziPoruku(nazivUdzbenika);

        }
    } // kraj klase UdzbenikTest
}
```

Error 1 - An object reference is required for the non-static field, method, or property
'udzbenik.Udzbenik.PrikaziPoruku(string)'

RAZVOJ PROGRAMSKIH RJEŠENJA

18. Analizirajte kod, pročitajte grešku, objasnite zašto je nastala i ispravite grešku:

```
using System ;  
  
namespace udzbenik  
{  
    class Udzbenik  
    {  
        void PrikaziPoruku(string naziv)  
        {  
            Console.WriteLine( "Udzbenik za \n{0}!", naziv);  
        }  
    } // kraj klase Udzbenik  
  
    class UdzbenikTest  
    {  
        static void Main(string[] args)  
        {  
            // kreira objekt preporuceniUdzbenik tipa klase Udzbenik  
            Udzbenik preporuceniUdzbenik = new Udzbenik();  
  
            Console.WriteLine( "Unesi naziv udzbenika:" );  
            string nazivUdzbenika = Console.ReadLine(); // čita liniju teksta  
  
            preporuceniUdzbenik.PrikaziPoruku(nazivUdzbenika) ;  
        }  
    } // kraj klase UdzbenikTest
```

Error 1 'udzbenik.Udzbenik.PrikaziPoruku(string)' is inaccessible due to its protection level

19. Analizirajte kod, pročitajte grešku, objasnite zašto je nastala i ispravite grešku:

```
using System ;  
namespace udzbenik  
{  
    class Udzbenik  
    {  
        private string autor ;  
        public void PrikaziPoruku(string naziv)  
        {  
            Console.WriteLine( "Udzbenik za \n{0}!", naziv);  
        }  
    } // kraj klase Udzbenik  
  
    class UdzbenikTest  
    {  
        static void Main(string[] args)  
        {  
  
            // kreira objekat preporuceniUdzbenik tipa klase Udzbenik  
            Udzbenik preporuceniUdzbenik = new Udzbenik();  
  
            Console.WriteLine( "Unesi naziv udzbenika:" );  
            string nazivUdzbenika = Console.ReadLine(); // čita liniju teksta  
  
            preporuceniUdzbenik.PrikaziPoruku(nazivUdzbenika) ;  
            Udzbenik.autor = "RI";  
        } // Main  
    } // kraj klase UdzbenikTest
```

Error 1 An object reference is required for the non-static field, method, or property
'udzbenik.Udzbenik.autor'

Error 2 'udzbenik.Udzbenik.autor' is inaccessible due to its protection level

Poglavlje 4: Napredni objektno-orientirani koncepti

U okviru ovog poglavlja opisani su napredni objektno-orientirani koncepti i način njihovog izražavanja u C#-u kroz tematske jedinice:

1. Nasljeđivanje
2. Polimorfizam
3. Interfejs
4. Delegati
5. Generički koncepti

4.1. Nasljeđivanje

Nasljeđivanje je važan koncept prilikom dizajniranja klasa i odnosi se na dizajniranje nove (izvedene, podklase) klase koja nasljeđuje članove postojeće (bazne, nadklase) klase. Izvedena klasa dodaje nova polja i metode. Izvedena klasa predstavlja specijaliziranu grupu objekata. Nasljeđivanjem može se formirati hijerarhija klasa tako da se iz već izvedene klase izvede odnosno definira nova klasa. C# ne podržava višestruko nasljeđivanje, odnosno klasa ne može biti nasljeđena od više klasa. U tu svrhu koristi se koncept interfejsa.

Kod nasljeđivanja važnu ulogu igra `protected član klase`.

`protected član klase`

Zaštićenom (`protected`) članu klase mogu pristupiti članovi bazne klase i članovi izvedenih klasa. Kod 4.1 prikazuje baznu klasu A koja sadrži `protected član x`. Na osnovu te klase se izvodi klasa B, što je naznačeno sa `class B : A { }`. Klasa B može pristupiti članu x. Ukoliko bi se unutar koda nalazile neke druge klase, koje nisu izvedene na osnovu klase A, one ne bi mogle pristupiti `protected članu x` klase A. U C#-u pored `protected`, postoji i `protected internal` zaštita za polje i odnosi se na omogućavanje pristupa izvedenim klasama koje su u istom asembliju.

```
using System;
class A
{
    protected int x = 123;
}
class B : A
{
    static void Main()
    {
        A a = new A();
        B b = new B();
        b.x = 10;
    }
}
```

Bazna klasa
Protected član klase
Izvedena klasa
Objekat izvedene klase pristupa protected članu x

Kod 4.1. Nasljeđivanje klasa i pristup protected članu klase

Korištenje `protected` polja prouzrokuje dva glavna problema:

- Objekti izvedene klase mogu direktno dodijeliti nedozvoljene vrijednosti za `protected` članove.
- Metode izvedene klase ovise o implementaciji osnovne klase.

Umjesto `protected`, bolje je da su članovi osnovne klase `private` i da se omoguće metode pomoću kojih će izvedene klase pristupati članovima bazne klase.

Kod 4.2 sadrži baznu klasu `Imovina`. Na osnovu nje se izvode klasa `Gotovina` i klasa `Kuca`. Bitni aspekti tog nasljeđivanja označeni su u okviru samog koda.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;

namespace NasljedivanjeKlasa
{
    public class Imovina ← Bazna klasa
    {
        public string vlasnik;
    }

    public class Gotovina : Imovina
    { public long iznos; }

    public class Kuca : Imovina
    { public string lokacija; }

    class Program
    {
        static void Main(string[] args)
        {
            Gotovina bInzenjer = new Gotovina { vlasnik = "InzenjerX", iznos = 10000 };
            Console.WriteLine(bInzenjer.vlasnik); // InzenjerX
            Console.WriteLine(bInzenjer.iznos); // 10000

            Kuca zgrada = new Kuca { vlasnik = "Kampus", lokacija = "Sarajevo" };
            Console.WriteLine(zgrada.vlasnik); // Kampus
            Console.WriteLine(zgrada.lokacija); // Sarajevo
        }
    }
}
```

Izvedena klasa Gotovina iz klase Imovina. Naslijedila polje vlasnik i dodala svoje polje iznos.

Izvedena klasa Kuca iz klase Imovina. Naslijedila polje vlasnik i dodala svoje polje lokacija.

Instance klasa Gotovina i Kuca koriste polje vlasnik klase Imovina

Kod 4.2: Nasljeđivanje klasa: Bazna klasa `Imovina` i nasljedene klase `Gotovina` i `Kuca`

4.2. Polimorfizam

Postoje dva tipa polimorfizma: statički polimorfizam za vrijeme kompilacije i dinamički polimorfizam koji se primjenjuje za vrijeme izvršavanja programa. Primjer statičkog polimorfizma je preklapanje funkcija (*function overloading*) i preklapanje operatora (*operator overloading*).

Dinamički polimorfizam (često se naziva samo polimorfizam) se dešava prilikom izvršavanja programa. Polimorfizam je svojstvo da svaki objekt izvedene klase izvršava metodu članicu onako kako je to definirano u njegovoj izvedenoj klasi, iako mu se pristupa kao objektu osnovne klase. Postiže se pomoću virtualnih metoda. Bazna klasa definira i implementira virtualne

RAZVOJ PROGRAMSKIH RJEŠENJA

(*virtual*) metode. Izvedena klasa ‘gazi’ (*override*) virtualnu metodu. Izvedena klasa može *override* člana bazne klase samo ako je deklariran sa *virtual* ili *abstract*. Izvedeni član mora koristiti *override* ključnu riječ. Polja klase ne mogu biti virtualna, samo metode, osobine, događaji, indekseri mogu biti virtuelni. Prilikom izvršavanja (run-time) na osnovu tipa objekta poziva se odgovarajuća *override* virtualna metoda. Virtualne metode omogućavaju da se sa grupom povezanih objekata radi na uniforman način. Polimorfizam se može iskoristiti u mnogim situacijama. Naprimjer, prilikom dizajniranja aplikacije za crtanje koja omogućava kreiranje različitih vrsta oblika. Za vrijeme kompilacije nije poznato koji će oblik korisnik kreirati. Prilikom dizajniranja klase, preporučuje se da se kreira hijerarhija klasa, u kojoj svaki specifični oblik se izvodi iz osnovne klase. Virtuelna metoda se koristi da omogući mehanizam pozivanja odgovarajuće metode na izvedenoj klasi i to na osnovu jedinstvenog poziva metode osnovne klase. Kreira se lista oblika u koju se dodjeljuju specifični objekti i koristi se *foreach* petlja za iteraciju kroz listu i za poziv metode za crtanje za svaki objekt.

U cilju ilustracije polimorfizma slijedi dopuna koda 4.2. Baznoj klasi je dodana metoda *PrikaziPodatke* koja je virtualna i njena osnovna namjena je prikaz imena vlasnika imovine. Metoda sa istim imenom postoji i u klasama *Kuca* i *Gotovina*, sa namjenom da prikaže lokaciju kuće i iznos gotovine vlasnika imovine.

```
using System;
namespace NasljedivanjeKlasa
{
    public abstract class Imovina
    {
        public string vlasnik;

        public virtual void PrikaziPodatke()
        {
            Console.WriteLine("Vlasnik je {0}", vlasnik);
            Console.WriteLine("_____ \n");
        }
    }

    public class Gotovina : Imovina <-- Izvedena klasa Gotovina
    {
        public long iznos;

        public override void PrikaziPodatke() <-- Override metoda PrikaziPodatke
        {
            Console.WriteLine("Iznos gotovine je {0}", iznos);
            base.PrikaziPodatke(); <-- Poziv metode PrikaziPodatke bazne klase
        }
    }
}

// nastavak na sljedećoj stranici
```

RAZVOJ PROGRAMSKIH RJEŠENJA

```
public class Kuca : Imovina
{
    public string lokacija;
    public override void PrikaziPodatke()    Override metoda PrikaziPodatke
    {
        Console.WriteLine("Lokacija kuce je {0}",lokacija);
        base.PrikaziPodatke();
    }
}

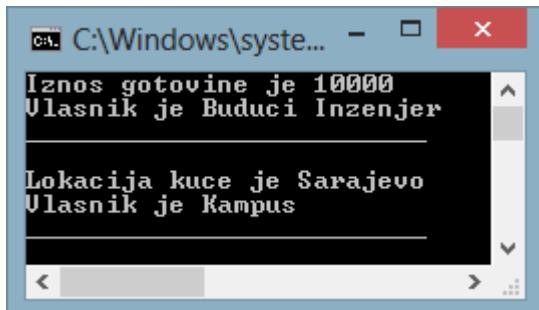
// TESTIRANJE POLIMORFIZMA
class Program
{
    static void Main(string[] args)
    {

        // kreiranje kolekcije objekata
        System.Collections.Generic.List<Imovina> imovina = new
        System.Collections.Generic.List<Imovina>();
        imovina.Add(new Gotovina { vlasnik = "Buduci Inzenjer", iznos = 10000 });
        imovina.Add(new Kuca { vlasnik = "Kampus", lokacija = "Sarajevo" });

        // poziva se virtualna metoda PrikaziPodatke svake izvedene klase
        foreach (Imovina s in imovina)
        {
            s.PrikaziPodatke();
        }
    }
}
```

Kod 4.3: Polimorfizam – nasljedivanje klasa i upotreba `override` metoda

Izvršavanjem koda 4.3. dobija se rezultat:



U baznoj i izvedenoj klasi može postojati metoda sa istim imenom, pri čemu implementacija metode u izvedenoj klasi može biti potpuno neovisna (ne vrši se `override`, ne poziva se sa

RAZVOJ PROGRAMSKIH RJEŠENJA

base) od implementacije metode u baznoj klasi. U tom slučaju se koristi operator new da se naglasi ta činjenica.

```
public new void PrikaziPodatke()
{
    Console.WriteLine("Iznos gotovine je {0}", iznos);
}
```

is i as operatori

Prilikom formiranja relacija između operatorka koristi se i operatori is, as.

- **is operator**

```
if (bInzenjer is Gotovina)
    Console.WriteLine("Jeste, objekat bInjezer je tipa Gotovina");
```

is provjerava da li je objekat određenog tipa ili tipa izvedenog na osnovu tog tipa podataka. U ovom slučaju provjera se da li je bInzenjer tipa Gotovina.

as operator

as operator izvršava konverziju između kompatibilnih referencnih tipova, vraća null ako je konverzija neuspješna.

```
Imovina bklasa = bInzenjer as Imovina ;
if ( bklasa !=null )
    Console.WriteLine("Moguća konverzija");
```

Prilikom izvršavanja ovog dijela koda s obzirom da je bInzenjer tipa klase Gotovina koja je izvedena iz klase Imovina, konverzija bi bila moguća.

4.2.1.Apstaktne klase i metode

Apstaktna klasa je klasa na osnovu koje se ne može instancirati objekat. Tokom poglavljia na osnovu svih prikazanih klasa mogli su se instancirati objekti. Klase na osnovu kojih se mogu instancirati objekti nazivaju se konkretne klase. Apstaktne klase sadrže neke apstaktne metode koje nisu implementirane. Implementacija apstraktnih metoda je u izvedenim klasama. Klasa se definira kao apstaktna sa abstract ključnom riječju ispred imena klase. Na vrhu hijerarhije klasa preporučuje se i dobro je imati apstraktnu klasu.

Kod 4.4. sadrži apstraktnu klasu Uposlenik sa apstraktnom metodom zarada.

```
// Uposlenik apstraktna bazna klasa.
public abstract class Uposlenik
{
    private string ime;
    private string prezime;
    private string idBroj;
    // konstruktor
    public Uposlenik(string pime, string pprezime, string pidBroj)
    {
        ime = pime;
        prezime = pprezime;
        idBroj = pidBroj;
    }

    public override string ToString()
    {
        return string.Format(" Ime {0} Prezime {1} Identifikacioni broj {2}",
            ime, prezime, idBroj);
    }

    // apstraktna metoda redefinira se - overridden sa izvedenim klasama
    public abstract decimal Zarada(); // nema ovdje implementacije, implementacija je u
    izvedenim klasama
```

Kod 4.4: Apstraktna klasa i apstraktna metoda

sealed metode i klase

Metoda označena sa `sealed` u baznoj klasi ne može se redefinirati (*overridden*) u izvedenoj klasi. Metode koje su `private` i `static` su implicitno `sealed` tj. ne mogu se redefinirati. Klasa, također, može biti `sealed`, takva klasa ne može biti bazna klasa.

4.3.Kreiranje i korištenje interfejsa

Interfejs sadrži definiciju za grupu povezanih funkcionalnosti koje klasa implementira. Interfejs služi kao specifikacija (definicija prototipa) javno raspoloživih (`public`) metoda (kao i propertija, događaja i indeksera). Klase ne nasljeđuju već implementiraju interfejse, odnosno implementiraju sve metode iz interfejsa kao javne.

Interfejsi nisu ništa isključivo svojstveno programskom jeziku C# - kao koncept postoje ili ih je moguće realizirati u svim objektno-orientiranim jezicima ili jezicima koji podržavaju objektnu orientaciju. U C++ interfejs se realizira kao apstraktna klasa koja nema nikakve atribute već se isključivo sastoji iz deklaracija apstraktnih virtualnih metoda, dok C# interfejse čini sastavnim dijelom svoje sintakse. C# sadrži ključnu riječ `interface` i na taj način pojednostavljuje

RAZVOJ PROGRAMSKIH RJEŠENJA

deklaraciju interfejsa. Još jedan od razloga zašto interfejsi imaju važnu ulogu u C#-u leži u činjenici da je višestruko nasljeđivanje u C#-u zabranjeno.

Primjer interfejs deklaracije:

```
interface IPlacanje
{
    decimal iznosPlacanja();
}
```

Interfejsi su tipično `public` vidljivosti. Njihova deklaracija se vrši u odvojenim fajlovima koji imaju isto ime kao interfejs i `.cs` ekstenziju. Sve metode interfejsa su implicitno deklarirane kao `public` i `abstract`.

C# dozvoljava da se izvedena klasa naslijedi iz bazne klase i da implementira više interfejsa.

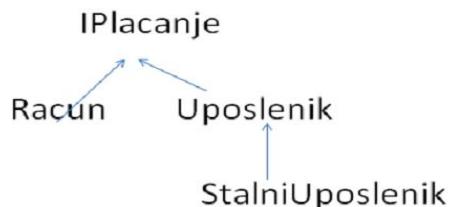
```
public class NazivKlase : OsnovnaKlase, PrviInterfejs, DrugiInterfejs, ...
```

Klasa koja djelimično implementira interfejs mora biti apstraktna i mora sadržavati apstraktну deklaraciju za svaki ne implementirani član interfejsa.

Konkretna klasa koja u potpunosti implementira interfejs mora deklarirati svaki član interfejsa kako je specificirano u deklaraciji interfejsa.

Interfejs se tipično koristi kada nepovezane klase dijele zajedničke metode. To omogućava objektima nepovezanih klasa da se procesiraju polimorfno.

Sljedeći primjer uvodi interfejs nazvan `IPlacanje` koji nudi i daje prototip za funkcionalnost plaćanja koju realizuju klase `Racun`, `Uposlenik`, `StalniUposlenik`.



Skica interfejsa `IPlacanje`

Kod za skicirani interfejs je:

```
interface IPlacanje
{
    decimal iznosPlacanja();
}
```

RAZVOJ PROGRAMSKIH RJEŠENJA

Ovaj interfejs je potrebno implementirati. Više klasa može implementirati jedan interfejs. Ispod je prikazan moguć scenarij implementacije interfejsa `IPlacanje`. Implementacija interfejsa sa klasom `Racun` data je sa kodom 4.5.

```
using System;

namespace InterfejsPlacanje
{
    // Racun klasa implementira interfejs IPlacanje
    public class Racun : IPlacanje
    {
        private string broj;
        private int kolicina;
        private decimal cijenaArtikla;

        public Racun(string pbroj, int pkolicina, decimal pcijenaArtikla)
        {
            broj = pbroj;
            kolicina = pkolicina;
            cijenaArtikla = pcijenaArtikla;
        }

        // get/set properties, ToString .....

        // implementacija metode iz interfejsa IPlacanje
        public decimal iznosPlacanja()
        {
            return kolicina * cijenaArtikla;
        }
    }
}
```

Kod 4.5. Klasa `Racun` implementira interfejs `IPlacanje`

Klase `Uposlenik`-apstraktna bazna klasa implementira interfejs `IPlacanje` sa izvedenom klasom `StalniUposlenik`. Kod 4.6 prikazuje definiciju apstraktne klase `Uposlenik`.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;

namespace InterfejsPlacanje
{// Uposlenik - apstraktna klasa
    public abstract class Uposlenik : IPlacanje
    {
        private string ime;
        private string prezime;
        private string idBroj;

        public Uposlenik(string pime, string pprezime, string pidBroj)
        {
            ime = pime;
            prezime = pprezime;
            idBroj = pidBroj;
        }

        // properties (set, get) za sve clanove klase, ...

        // Ovdje se ne implementira metoda iznosPlacanja, implementira se u izvedenoj
        klasiji klase Uposlenik
        public abstract decimal iznosPlacanja();
    }
}
```

Kod 4.6: Apstraktna klasa Uposlenik

Klase StalniUposlenik (izvedena iz klase Uposlenik) implementira Iplacanje interfejs (kod 4.7).

```
using System;
namespace InterfejsPlacanje
{
    // StalniUposlenik klasa nasljedena iz klase Uposlenik
    public class StalniUposlenik : Uposlenik
    {
        private decimal mjesecnaZarada;

        public StalniUposlenik(string pime, string pprezime, string pidBroj,
                               decimal pmjesecnazarada)
            : base(pime, pprezime, pidBroj)
        {
            mjesecnaZarada = pmjesecnazarada;
        }
        //racuna zaradu, implementira interfejs Iplacanje -metodu iznosPlacanja

        // iznosPlacanja je apstraktna u baznoj klasiji Uposlenik
        public override decimal iznosPlacanja()
        {
            return mjesecnaZarada;
        }
    }
}
```

Kod 4.7: Klasa StalniUposlenik implementira Iplacanje interfejs

RAZVOJ PROGRAMSKIH RJEŠENJA

Implementirani interfejs se može koristiti polimorfno. Ispod sa kodom 4.8 je prikazano polimorfno korištenje `IPlacanje` interfejsa preko klasa `Racun` i `StalniUposlenik`.

```
using System;

// Test interfejsa InterfejsPlacanje sa nevezanim klasama Racun, Uposlenik
namespace InterfejsPlacanje
{
    public class PlacanjeInterfaceTest
    {
        public static void Main(string[] args)
        {
            // kreira niz IPlacanje elemenata
            IPlacanje[] placanjeObjekti = new IPlacanje[2];

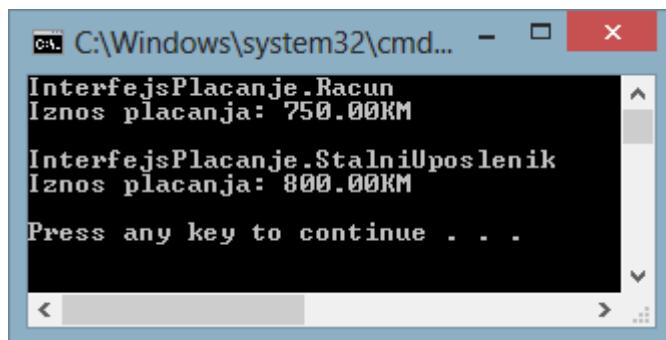
            // popunjavanje niza sa objektima koji implementiraju IPlacanje
            placanjeObjekti[0] = new Racun("01234", 2, 375.00M);

            placanjeObjekti[1] = new StalniUposlenik("Adnan", "Prezi", "12451", 800.00M);

            // procesira polimorfno svaki element u nizu placanjeObjekti
            foreach (IPlacanje tekucePlacanje in placanjeObjekti)
            {
                Console.WriteLine("{0} \n{1}: {2}KM\n", tekucePlacanje,
                    "Iznos placanja", tekucePlacanje.iznosPlacanja());
            }
        }
    }
}
```

Kod 4.8. Polimorfno korištenje interfejsa

Prilikom implementacije projekta primijenila se i programska dekompozicija. Svi iznad prikazani kodovi su fizički u drugim fajlovima. Izvršavanjem gore navedenog projekta dobija se:



RAZVOJ PROGRAMSKIH RJEŠENJA

Interfejsi imaju višestruku namjenu. Slijedi opis scenarija kada se mogu iskoristiti. Razvoj jednog programskog rješenja podijeljen je na dvije velike komponente koje će razvijati odvojeni razvojni timovi. Postoji komunikacija između komponenti (elementi jedne komponente pozivaju elemente druge komponente). Pitanje je kako osigurati timovima neovisnost u razvoju bez da moraju čekati jedni na druge da razviju elemente koji su potrebni u njihovoj vlastitoj komponenti.

Dva razvojna tima na početku mogu specificirati niz interfejsa sa njihovim specifikacijama, te nakon toga razvijati softver neovisno jedni o drugima (ovisno o arhitekturi i dizajnu sistema). Kada god prvom timu zatreba element kojeg razvija drugi tim, na raspolažanju imaju interfejs koji je zajednički usaglašen (kod 4.9). Prvi tim može razviti *dummy* element (kod 4.10) koji će služiti samo za potrebe testiranja ispravnosti rada a koji će implementirati odgovarajući interfejs (ili više njih). Kada oba tima završe razvoj svojih komponenti, samo se iz koda izbaci instanciranje *dummy* elemenata i zamjeni se instanciranjem pravih elemenata (kod 4.11). Više nikakve izmjene nije potrebno raditi što se tiče poziva metoda.

```
using System;

namespace DogovorenIInterf
{

    public interface DogovorenIInterfejs
    {
        Boolean dogovorenaMetoda(int dogovoreniParametar);
    }

    public class DummyKomponenta : DogovorenIInterfejs
    {
        public Boolean dogovorenaMetoda(int dogovoreniParametar)
        {
            return true;
        }
    }

    public class PravaKomponenta : DogovorenIInterfejs
    {
        public Boolean dogovorenaMetoda(int dogovoreniParametar)
        {
            return true;
        }
    }
}
```

Kod 4.9: Usaglašeni interfejs

RAZVOJ PROGRAMSKIH RJEŠENJA

Kod tokom razvoja pojedinačnih komponenti:

```
static void Main(string[] args)
{
    DogovoreniInterfejs pluggableKomponenta = new DummyKomponenta();

    // kod koji se oslanja na metodu pluggableKomponenta
    pluggableKomponenta.dogovorenMetoda(10);
}
```

Kod 4.10: Dio koda sa *dummy* komponentom

Kod nakon integracije komponenti. Uočava se da poziv dogovorene metode nije bilo potrebno mijenjati.

```
static void Main(string[] args)
{
    DogovoreniInterfejs pluggableKomponenta = new PravaKomponenta();

    // kod koji se oslanja na metodu pluggableKomponenta
    pluggableKomponenta.dogovorenMetoda(10);
}
```

Kod 4.11: Kod nakon integracije implementiranih dijelova interfejsa

4.4.Delegati

Tip podataka koji omogućava da se smjeste reference na metode (funkcije) naziva se delegat (eng. *delegate*). Delegat se deklarira kao metoda bez tijela označena sa ključnom riječu `delegate`. Delegate deklaracija specificira povratni tip i listu parametara.

- Deklaracija delegata:

```
delegate double ProcessDelegate(double param1, double param2);
```

Nakon deklaracije delegata može se instancirati varijabla tipa delegata kao:

```
ProcessDelegate process;
```

Varijabla delegat tipa može se inicijalizirati kao referenca na bilo koju metodu koja ima isti povratni tip i parametar listu kao delegat. Naprimjer, može se referencirati na metodu `Mnozenje`:

```
process = new ProcessDelegate(Mnozenje);
```

metoda
↑

RAZVOJ PROGRAMSKIH RJEŠENJA

Nakon inicijalizacije može se pozvati metoda (`Mnozenje`) korištenjem varijable (`process`) koja je delegat tipa:

```
rez = process(param1, param2)
```

Kod 4.12 prikazuje primjer upotrebe delegata. Deklariran je delegat `ProcessDelegate` preko kojeg se pozivaju dvije različite metode `Dijeljenje` i `Mnozenje`, koje imaju isti povratni tip (`double`) i parametar listu (dva `double` parametra) kao i sam delegat. Kod 4.12 kao i ostali kodovi u knjizi imaju ilustrativnu namjenu određenog koncepta, tako da nije implementirana validacija unosa, inicijalizacija parametara i slično.

```
using System;

namespace DelegatePrimjer
{
    class Program
    {
        delegate double ProcessDelegate(double param1, double param2);

        static double Mnozenje(double param1, double param2)
        {
            return param1 * param2;
        }

        static double Dijeljenje(double param1, double param2)
        {
            return param1 / param2;
        }

        static void Main(string[] args)
        {
            ProcessDelegate process; ← Instanciranje delegata

            double param1 = 10.2;
            double param2 = 5.1;
            Console.WriteLine("Unesite M za množenje ili D za djeljenje:");

            string input = Console.ReadLine();

            if (input == "M")
                process = new ProcessDelegate(Mnozenje);
            else
                process = new ProcessDelegate(Dijeljenje);

            Console.WriteLine("Rezultat: {0}", process(param1, param2));
        }
    }
}
```

Deklaracija delegata

Metode: `Mnozenje` i `Dijeljenje` imaju isti broj, tip parametara i povratnu vrijednost kao i delegat `ProcessDelegate`

Instanciranje delegata

Inicijalizacija delegata `process` (referenca na metodu)

Poziv metode korištenjem delegata

Kod 4.12: Deklaracija i upotreba delegata

RAZVOJ PROGRAMSKIH RJEŠENJA

Korištenjem delegata može se kreirati lista metoda koje će se pozvati kada se delegat uključi. Na listu delegata mogu se primjenjivati operatori + i -, kojim se dodaju odnosno brišu metode iz liste. Delegat se može predati metodi kao parametar. Delegati su veoma važni za događaje (*event*) što se detaljno obrađuje u poglavlju 5.

4.4.1. Anonimna metoda

Anonimne (bez imena) metode obezbjeđuju tehniku da se preda blok koda kao delegat parametar. Anonimne metode su ustvari metode bez imena, sastoje se samo od tijela metode. Nije potrebno specificirati povratni tip u anonimnim metodama. Povratni tip se određuje na osnovnu `return` iskaza unutar tijela metode. Kod 4.13 sadrži anonimnu metodu sa tijelom `Console.WriteLine("Anonimna metoda: {0}", x);`. Delegat može biti pozvan i sa anonimnim metodama kao i sa imenovanim metodama (u kodu 4.13 metoda `DodajBroj`).

```
using System;

delegate void PromjenaBroja(int n);
namespace AnonimnaMetoda
{
    class TestDelegate
    {
        static int broj = 10;

        public static void DodajBroj(int pbroj)
        {
            broj += pbroj;
            Console.WriteLine("Metoda DodajBroj: {0}", broj);
        }

        static void Main(string[] args)
        {
            //Kreiranje delegata instance korištenjem anonimne metode
            PromjenaBroja delegatPromjenaBroja = delegate(int x)
            {
                Console.WriteLine("Anonimna metoda: {0}", x);
            };

            //pozivanje delegata korištenjem anonimne metode
            delegatPromjenaBroja(106);

            //instanciranje delegata korištenjem metode DodajBroj
            delegatPromjenaBroja = new PromjenaBroja(DodajBroj);

            //pozivanje delegata korištenjem metode DodajBroj
            delegatPromjenaBroja(321);
        }
    }
}
```

Kod 4.13: Poziv anonimne i imenovane metode preko delegata

RAZVOJ PROGRAMSKIH RJEŠENJA

Izvršavanjem gore navedenog koda na standardnom izlazu se nakon proslijedivanja anonimne metode delegatu i izvršavanja koda anonimne metode ispisuje 106. Nakon referenciranja delegata na metodu DodajBroj ispisuje se 331.

4.4.2. Lambda izraz

Lambda izraz (*lambda expression*) je anonimna funkcija koja se koristi za kreiranje delegata i složenijih tipova izraza u obliku drveta (*expression tree*). Korištenjem lambda izraza, mogu se pisati lokalne funkcije koje se mogu predati kao argumenti. Lambda izrazi su korisni za pisanje LINQ izraza.

Da bi se kreirao lambda izraz, ulazni parametri se specificiraju na lijevu stranu lambda operatora =>, a na drugoj strani lambda operatora se piše izraz. Na lijevoj strani lambda izraza nisu dozvoljeni operatori is, as.

Dakle, osnovna lambda definicija: (Ulagni parametri) => Izraz.

Naprimjer, u lambda izazu broj => broj % 2 == 0, broj je ulazni parametar, broj % 2 == 0 je izraz. Potpuni lambda izraz broj => broj % 2 == 0 bi se mogao riječima opisati, za ulazni parametar broj poziva se anonimna funkcija koja provjerava da li je broj paran. Ako je paran vraća se rezultat 0.

Zagrade za ulazne parametre nisu obavezne samo ako je jedan ulazni parametar. Ako ima više ulaznih parametara odvajaju se sa ,. Primjer lambda izraza sa dva ulazna parametra (x, y) => x == y

Ponekad je teško ili nemoguće da kompjajler zaključi kojeg je tipa ulazni parametar. U tim slučajevima tipovi se mogu specificirati eksplisitno.

```
(int x, string s) => s.Length > x
```

Lambda izrazi se mogu koristiti i prilikom kreiranja delegata. Kod 4.14 ilustrira primjenu lambda izraza prilikom definicije delegata.

```
delegate int del(int i);
static void Main(string[] args)
{
    del racunDelegate = x => x * x;
    int mnozenje = racunDelegate(5);
}
```

Kod 4.14: Upotreba lambda izraza prilikom kreiranja delegata

Lambda iskazi

Lamda iskazi na desnoj strani imaju iskaz ili iskaze (tipično ne više od 2-3) koji se nalaze u vitičastim zagradama.

```
(ulazni parameteri) => {iskaz(i);}
```

```
using System;

namespace lambdaIskaz
{
    class Program
    {
        delegate void TestDelegate(string s);

        static void Main(string[] args)
        {

            TestDelegate predmetDelegat = n =>
                { string s = "C# RAZVOJ" + " " + n; Console.WriteLine(s); };

            predmetDelegat("PROGRAMSKIH RJEŠENJA");

        }
    }
}
```

Kod 4.15: Primjer lambda iskaza

Izvršavanjem koda 4.15. na standardnom izlazu ispisuje se poruka C# RAZVOJ PROGRAMSKIH RJEŠENJA.

Lambda izraz se može iskoristiti i za specificiranje nula ulaznih parametara. To se postiže sa praznim zagradama. U tom slučaju koristi se sljedeća sintaksa.

```
() => Metoda()
```

U ovom slučaju tijelo lambda izraza sadrži poziv metode. Primjena ovog koncepta je ilustrirana u poglavlju koje obrađuje višenitnost.

4.5. Generički koncepti

Programski jezik C# nudi mogućnost kreiranja generičkih metoda, klasa, interfejsa i delegata koji su sposobni da rade sa više tipova podataka i da su pri tome tip-sigurni. Nabrojani koncepti koriste parametre koji predstavljaju tip podatka. Taj tip podataka je generički tip. Specifični tip podataka se obezbjeđuje za vrijeme pozivanja metode ili drugog navedenog koncepta.

Primjer jednostavne generičke klase ilustriran je sa kodom 4.16. Definirana je jednostavna generička klasa `TestGenerickaKlasa` sa dvije metode. Klasa `TestGenerickaKlasa` prima parametar koji predstavlja tip podatka (`TipPodatka`). Umjesto definiranja dvije odvojene klase koje rade sa različitim tipovima podataka definirana je jedna generička klasa.

```
using System;

namespace genericKlasa
{
    class TestGenerickaKlasa<TipPodatka>
    {
        TipPodatka podatak ;

        public TestGenerickaKlasa(TipPodatka parametarPodatak)
        {
            this.podatak = parametarPodatak;
        }

        public void PrikaziPodatak()
        {
            Console.WriteLine(this.podatak);
        }
    }

    class Program
    {
        static void Main()
        {
            // korištenje klase TestGenerickaKlasa sa parametrom tipa double
            TestGenerickaKlasa<double> test1 = new TestGenerickaKlasa<double>(5.6);
            test1.PrikaziPodatak();

            // korištenje klase TestGenerickaKlasa sa parametrom tipa string
            TestGenerickaKlasa<string> test2 = new TestGenerickaKlasa<string>("RPR");
            test2.PrikaziPodatak();
        }
    }
}
```

Kod 4.16: Primjer generičke klase

RAZVOJ PROGRAMSKIH RJEŠENJA

Konstruktor klase u kodu 4.16. je ujedno i primjer generičke metode. Generička metoda može imati više parametara, pri čemu svi mogu biti generički, ali može neki od njih da bude i parametar sa poznatim tipom podataka. Generički interfejs se implementira slijedeći pravila definiranja interfejsa uz predaju generičkog parametra koji predstavlja tip podatka. U .NET postoji mnogo generičkih interfejsa preko kojih se radi sa generičkim kolekcijama podataka (liste, stek, redovi, riječnik i sl.) koji se nalaze u `System.Collections.Generics`.

Generički koncepti pomažu da se odvoje podaci od logičkog koda, smanji količina koda, doprinose boljoj iskorištenosti koda i performansama izvršavanja programa.

Rezime:

U okviru ovog poglavlja obrađeni su koncepti objektno-orientiranog programiranja-nasljeđivanje, polimorfizam sa posebnim osvrtom na njihovo izražavanje u C#-u. Obrađen je veoma važan koncept interfejsa. Poglavlje je dalo osvrt i na lambda izraze, iskaze, delegate kao i generičke koncepte. Izuzetno je važno primjeniti ove koncepte pri razvoju programskih rješenja, pa se čitaocima skreće pažnja da urade zadatke za samostalni rad.

Pitanja za ponavljanje i samostalni rad:

1. Objasnite pojam nasljeđivanja.
2. Koja je razlika između baznih i izvedenih klasa?
3. Objasnite modifikatore pristupa `protected` i `protected internal`.
4. Koji se problemi mogu javiti upotrebom `protected` pristupa?
5. Navedite primjere nasljeđenih klasa.
6. Objasnite pojam polimorfizma.
7. Šta je virtualna metoda, koja je njena uloga pri polimorfnom procesiranju klasa?
8. Kako se obilježava C# apstraktna klasa.
9. Navedite primjer međusobno povezanih klasa koje se mogu polimorfno procesirati.
10. Objasnite `is`, `as` operatore.
11. Kada se koristi `base` ključna riječ?
12. Šta se postiže sa `sealed` modifikatorom?
13. Objasnite programsku konstrukciju interfejsa.
14. Navedite primjer koji ilustrira upotrebu interfejsa.
15. Objasnite ulogu i način kreiranja delegata.
16. Navedite primjer koji ilustrira upotrebu delegata.
17. Analizirajte kod i ispravite greške ukoliko postoje:

```
using System;
class A
{
    protected int x = 123;
}
class B : A
{
    static void Main()
    {
        A a = new A();
        B b = new B();

        a.x = 10;
        b.x = 10;
    }
}
```

RAZVOJ PROGRAMSKIH RJEŠENJA

18. Analizirajte kod i ispravite greške ukoliko postoje:

```
using System;

namespace Nasljedivanje
{

    public class Imovina
    {   protected string vlasnik;
    }

    public class Gotovina : Imovina
    {   public long iznos; }

    public class Kuca : Imovina           {   public decimal vrijednost; }

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

            Gotovina bInzenjer = new Gotovina { vlasnik="Buduci Inzenjer", iznos=10000 };
            Console.WriteLine (bInzenjer.vlasnik);
            Console.WriteLine (bInzenjer.iznos);
            Kuca zgrada= new Kuca { vlasnik="Kampus",   vrijednost=2500000 };
            Console.WriteLine (zgrada.vlasnik);
            Console.WriteLine (zgrada.vrijednost);

        }
    }
}
```

19. Analizirajte sljedeći primjer. Procijeniti na osnovu analize koda izlazne rezultate.

```
namespace Delegate2
{
    class Program
    {
        public delegate void mojDelegat(float foo, float bar);

        class Matematika
        {
            public void Plus(float a, float b) { Console.WriteLine("Plus:" + (a + b)); }
            public static void Minus(float a, float b) { Console.WriteLine("Minus:" + (a - b)); }
        }

        public static void Puta(float a, float b) { Console.WriteLine("Puta:" + (a * b)); }

        static void Main(string[] args)
        {

            Matematika m = new Matematika();
            mojDelegat d1 = new mojDelegat(m.Plus);
            mojDelegat d2 = new mojDelegat(Matematika.Minus);
            mojDelegat d3 = new mojDelegat(Puta);

            d1(2, 3); // poziv metode (m.Plus) na koju pokazuje delegat d1
            d2(2, 3); // poziv metode (Matematika.Minus) na koju pokazuje delegat d2
            d3(2, 3); // poziv funkcije (Puta) na koju pokazuje delegat d3
            Console.WriteLine("-----");
        }
    }
}
```

RAZVOJ PROGRAMSKIH RJEŠENJA

```
//formiranje liste delegata
List<mojDelegat> l = new List<mojDelegat>();
l.Add(d1); l.Add(d2); l.Add(d3);
foreach (mojDelegat d in l) d(4, 2); // preko delegata pozivaju se metode m.Plus,
Matematika.Minus i Puta sa parametrima 4 i 2
Console.WriteLine("-----");

mojDelegat dx = d1; // dx pokazuje na istu funkciju kao i d1 (m.Plus)
dx(2, 3);

Console.WriteLine("-----");
dx += d2; // formira se lista poziva (invocation list) delegata
dx(4, 3); //izvršavaju se metode na koje pokazuje dx i d2 (m.Plus, Matematika.Minus)
}
}
```

20. Objasnite lambda izraz i lambda iskaz.

21. Šta je specificirano sa lambda izrazom $x \Rightarrow x * x$?

22. Razviti programsko rješenje koje će omogućiti iznajmljivanje putničkih i teretnih automobila. Automobili se opisuje osnovnim podacima tip automobila, broj šasije, tip goriva. Za putničke automobile je bitan i broj sjedišta i kategorija (obični, luksuzni). Za teretne automobile bitna je informacija da li posjeduje prikolicu i nosivost. Osnovna cijena iznajmljivanja za obična putnička vozila je 3 KM po danu, a za luksuzna 5 KM. Ukoliko automobil ima više od dva sjedišta cijena po danu se uvećava za 10%. Osnovna cijena iznajmljivanja za teretna vozila je 6 KM po danu. Ta cijena se uvećava za 20% po danu ukoliko teretno vozilo posjeduje prikolicu i nosivost mu je veća od 3 tone. Rješenje treba da evidentira u listu (može se uraditi direktno u programu preko konstruktora) za svaku vrstu automobila po 3 primjerka. Nakon toga omogućiti da osoba zainteresirana za iznajmljivanje unese broj dana za koji želi iznajmiti automobil i podatke o automobilu koji želi iznajmiti. Na osnovu unosa podataka od korisnika i liste unesenih automobila obavijestiti korisnika da li postoji traženi automobil i koja je cijena iznajmljivanja.

Prilikom dizajniranja klase uspostaviti hijerarhiju klase sa apstraktnom klasom na vrhu hijerarhije. Potrebno je primijeniti programsku dekompoziciju i napisati program u C#-u.

23. Razviti programsko rješenje koje će omogućiti 1) evidenciju podataka o reketima za tenis 2) iznajmljivanje reketa. Za sada se nude juniorski teniski reketi i reketi za amatera, a planira se i ponuda za profesionalne rekete. Za sve teniske rekete bitni su podaci: proizvođač reketa (Head, Wilson, Babolat), godina proizvodnje. Za juniorske rekete se evidentira podatak o težini i boji reketa. Za amaterske rekete bitan podatak je prodajna cijena reketa, veličina drške(3,4,5,6).

RAZVOJ PROGRAMSKIH RJEŠENJA

Iznajmljivanje juniorskih reketa se vrši tako da se za svaki dan iznajmljivanja plaća 1KM. Ukoliko je reket crvene boje i Wilson tipa cijena po danu se povećava za 2%. Amaterski reketi se iznajmljuju tako da se po danu iznajmljivanja plaća 1.5% od trenutne prodajne cijene reketa. Ukoliko je veličina drške manja od 5 obračunata cijena za svaki dan se povećava za 0.5%.

Rješenje treba da evidentira u listu (može se uraditi direktno u programu preko konstruktora) za svaki tip reketa po 3 različita primjerka (instance). Nakon toga omogućiti da osoba zainteresirana za iznajmljivanje unese broj dana za koji želi iznajmiti reket i potrebne podatke o reketu koji želi iznajmiti.

Na osnovu unesenih podataka od korisnika i liste unesenih reketa obavijestiti korisnika da li postoji traženi reket i koja je cijena iznajmljivanja.

Prilikom dizajniranja klase uspostaviti hijerarhiju klasa sa apstraktnom klasom na vrhu hijerarhije i definiranim interfejsom. Potrebno je primijeniti programsku dekompoziciju i napisati program u C#-u.

24. ETF.NET je novi i perspektivni ISP (Internet Service Provider) u Sarajevu. Kao takav, ima svoje korisnike i različite internet pakete koje nudi. ETF.NET želi čuvati minimalnu količinu podataka o svojim korisnicima- za fizička lica se čuva identifikacijski (id) broj korisnika, ime, prezime, datum rođenja i lokacija, dok se za pravna lica čuva id korisnika, naziv kompanije i lokacija.

Internet paketi opisuju se nazivom, specifikacijom brzine ("10/1 Mbps" - čuvati kao string) i razlikuju se prema načinu obračuna: postoje paketi sa neograničenim (flat) prometom kod kojih je iznos računan fiksan bez obzira na promet u tekućem mjesecu, te paketi sa ograničenim prometom. Kod njih se u slučaju da korisnik nije prešao limit naplaćuje samo osnovna cijena, dok se u slučaju prekoračenja primjenjuje formula $iznos = osnovna\ cijena + (promet - limit) \cdot koeficijent$. Koeficijent se odnosi na dodatnu naplatu KM/MB (1 MB = 1024 kB).

Korisnici su za internet pakete vezani ugovorima. Svaki ugovor opisan je svojim identifikacijskim brojem i referencira korisnika i internet paket za kojeg je korisnik izvršio pretplatu.

Potrebno je razviti programsko rješenje koje će omogućiti kreiranje novih flat ili ograničenih paketa, te unos novih fizičkih ili pravnih lica kao korisnika u sistem. Također, treba omogućiti kreiranje ugovora između ETF.NET i pravnih i fizičkih lica a u vezi sa pretplatom na neki od postojećih paketa u ponudi. Dodatno, programsko rješenje treba omogućiti da se za dati broj ugovora i uneseni načinjeni promet može izračunati iznos za naplatu kao i ispis usluga koje koristi određeni korisnik. Naši analitičari dali su i neke preporuke što se tiče implementacije ovog rješenja. Predlažu da posao odvojite u tri zadatka.

RAZVOJ PROGRAMSKIH RJEŠENJA

Prvi zadatak

Identificirajte klase i odnose među njima. Obavezno koristite nasljeđivanje i apstraktne bazne klase, polimorfizam i virtualne metode. Analitičari preporučuju da pazite na mjerne jedinice za opisivanje limita i mjesecnog prometa, kao i na to gdje i kako ćete realizovati unos korisnika i internet paketa.

Drugi zadatak

Analitičari i dizajneri naše kompanije zahtijevaju da razvijeni sistem klasa vezan uz korisnike i njihove ugovore bude što je manje moguće ovisan o konkretnom problemu ISP-a kako bi se razvijeni sistem mogao iskoristiti već u sljedećem projektu. Dizajneri ističu da je potrebno definirati ispravne interfejsе, ali ne žele reći koji su to interfejsi. Potrebno je formirati i dobro osmišljene biblioteke - .dll.

Treći zadatak

Sada implementirajte klase do kraja i napravite glavnu programsku petljу sa menijem u kojem su ponuđene opcije: 1) kreiranje novog paketa određenog tipa 2) kreiranje novog korisnika određenog tipa 3) kreiranje ugovora za korištenje paketa 4) opciju za izračunavanje računa na osnovu broja ugovora i datog napravljenog prometa 5) opciju za ispis usluga koje koristi korisnik sa datim identifikacijskim brojem.

Poglavlje 5: Grafički korisnički interfejs (GUI) i uvod u programiranje vodeno događajima

U okviru ovog poglavlja dat je pregled historijskog razvoja grafičkog korisničkog interfejsa, grafičkih kontrola, objašnjen način i svrha upravljanja događajima kroz sljedeće tematske jedinice:

1. Osnove i značaj korisničkog interfejsa
2. GUI kontrola: Windows (prozor) – osnovne karakteristike
3. Windows Form i dugme kontrole
4. Model upravljanja događajima

5.1. Osnove i značaj korisničkog interfejsa

Interakcija čovjek-kompjuter (eng. Human-Computer Interaction, skraćenica HCI) uključuje izučavanje, planiranje i dizajn interakcije između ljudi i kompjutera. HCI dizajneri moraju uzeti u obzir različite faktore kao što su: šta ljudi žele i očekuju, koja su fizička ograničenja i sposobnosti ljudi, koja je njihova percepcija i procesiranje informacija, šta je ljudima ugodno i atraktivno. Dizajneri moraju uzeti u razmatranje i tehničke karakteristike i ograničenja hardvera i softvera. Osnovni cilj interfejsa je da učine rad sa kompjuterskim sistemima jednostavnim, produktivnim i ugodnim.

Korisnički interfejs (*user interface*) ima dvije komponente: ulaz i izlaz (*input, output*). Ulaz je komponenta koja omogućava da osoba prenosi informacije (potrebe i želje) kompjuteru. Neke zajedničke ulazne komponente su tastatura, miš, prst - za ekrane osjetljive na dodir, glas - za govorne instrukcije.

Izlazna komponenta omogućava da kompjuterski sistem korisnicima prenosi i prikazuje informacije. Danas je najčešći izlaz na ekran (*screen*) uz nadopunu sa zvukom.

Često korisnički interfejs nije odgovarajuće dizajniran jer se samom dizajnu interfejsa ne posveti dovoljna pažnja i ne odvoji se dovoljno vremena za dizajn i razvoj korisničkog interfejsa. Osim toga ljudi imaju različite vizuelne percepcije.

RAZVOJ PROGRAMSKIH RJEŠENJA

Dobro dizajniran korisnički interfejs je veoma važan za korisnike. Danas su aktuelni grafički korisnički interfejsi (Graphical User Interface, skraćenica GUI). Izgled ekrana (*screen layout*), navigacija kroz sistem utiče na korisnike na različite načine. Najbolji interfejs je interfejs koji dozvoljava korisniku da se fokusira na informacije i zadatke. Ako je izgled ekrana i navigacija konfuzna i neefektivna, ljudi mnogo teže rade svoj posao i prave mnogo grešaka.

5.1.1. Istorija grafičkog korisničkog interfejsa

GUI je grafičko korisničko okruženje, to je način interakcije čovjeka s računarom kroz manipulaciju grafičkim elementima i dodacima uz pomoć tekstualnih poruka i obavijesti.

Neki GUI elementi koje koriste današnji korisnički interfejsi su:

- pointer - simbol koji se pojavljuje na ekranu i koji se pomjera da bi se selektirali objekti i komande,
- desktop* - područje na ekranu gdje su ikone grupirane,
- ikona - sličica na *desktopu*, tj. pozadini koja prikazuje komandu, fajl, prozor,...
- prozor (*windows*) – omogućava podjelu ekrana u različita područja,
- meni - dozvoljava izbor jedne od više opcija,
- dugmad (*button*) s tekstrom i/ili slikama- najčešće se koristi za pokretanje neke akcije,
- tekst polje - područje za unos teksta ili prikaz teksta,
- grafičke kontrole za odabir opcija (*check box, radio button, list box,...*)

U okviru poglavlja 5-8 detaljno se izučava većina GUI elemenata, sa primjerima implementacije kroz Visual Studio okruženje.

Prvi početak razvoja korisničkog interfejsa je period 1960 godine. Ivan Sutherland autor programa Sketchpad (1963 godine) se smatra prvim koji je uveo grafiku. Linije, krugovi i tačke su se mogli crtati na ekranu koristeći svjetlosno pero (*light pen*). Xerox je 1960 godine radio na razvoju uređaja za pokazivanje koji bi se držao u ruci.

Neki bitni datumi za razvoj grafičkog interfejsa, sa većim akcentom na Microsoft, zbog izabranog okruženja za rad su dati u tabeli 5.1.

RAZVOJ PROGRAMSKIH RJEŠENJA

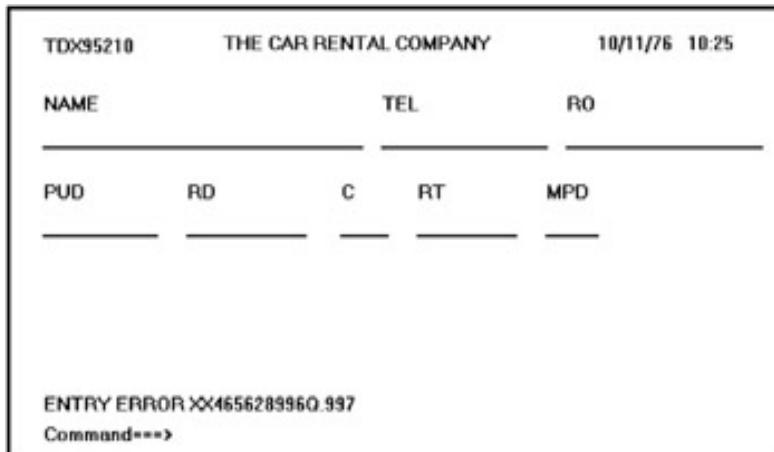
1973	Pionirski početak u Xerox Palo Alto Research Center. -Prvi koji su povezali sve elemente modernog GUI-a.
1981	Prvi komercijalni marketing u Xerox STAR. -Uvođenje pokazivača, selekcije i miša.
1983	Apple objavljuje Lisu. -Pojavljuju se padajući meniji.
1984	Apple objavljuje Macintosh. -Macintosh je prvi uspješan masovno prihvaćen sistem sa GUI-jem.
1985	Pojavljuje se Microsoft Windows 1.0. Commodore uvodi Amiga 1000.
1987	Window System postaje široko raspoloživ. Objavljen IBM's Presentation Manager - zamjena za DOS operativni sistem. Apple je objavio Macintosh II (prvi Macintosh u boji).
1988	NeXTStep proizvod na tržištu - simulira trodimenzionalni ekran.
1989	Pojavljuje se UNIX bazirani GUI. - AT&T i Sun Microsystems objavljuju Open Look. - DEC i Hewlett-Packard objavljuju Motif, za Open Software Foundation. Microsoft objavljuje Windows 3.0.
1992	Objavljen OS/2 Workplace Shell. Objavljen Microsoft Windows 3.1.
1993	Objavljen Microsoft Windows NT.
1995	Microsoft objavljuje Windows 95.
1996	IBM objavljuje OS/2 Warp 4. Microsoft uvodi NT 4.0.
1997	Apple objavljuje Mac OS 8.
1998	Microsoft objavljuje Windows 98.
1999	Apple objavljuje Mac OS X Server.
2000	Objavljen Microsoft Windows 2000. Objavljen Microsoft Windows ME.
2001	Objavljen Microsoft Windows XP.
2007	Objavljena Windows Vista.
2009	Windows 7 – trake su zamjenile menije u nekim aplikacijama, moglo su se postaviti ikone na task baru.
2012	Windows 8, interfejs optimiziran za mobilne uređaje, preko Vista 7 desktop interfejsa. Ukinut Start sa desktop moda.

Tabela 5.1: Hronološki pregled razvoja grafičkog korisničkog interfejsa

RAZVOJ PROGRAMSKIH RJEŠENJA

5.1.2. Kratka istorija dizajniranja ekrana

Ekran iz 1970 godine izgledao je kao na slici 5.1. Obično se sastojao od više polja čiji naslovi su uglavnom kriptovani i nerazgovijetni. Bio je vizuelno natrpan i često je imao komandno polje za koje je korisnik morao znati kako aktivirati. Nejasne poruke su često zahtijevale čitanje uputa da bi se shvatile. Korištenje ovakvog ekrana je zahtijevalo mnogo prakse i strpljenja. Većina prvih ekrana je bila monohromatska sa zelenim tekstom na crnoj podlozi.



Slika 5.1: Ekran iz 1970 godine [12]

Dekadu poslije postala su raspoloživa uputstva za dizajniranje tekst baziranih ekrana. Na slici 5.2 prikazan je primjer ekrana te dekade. Polja imaju jasnije opise u odnosu na ekrane prethodnih generacija i prikazane su komande na ekranu. Uočava se poravnanje i grupiranje elemenata. Poruke su također postale jasnije. Ali i ovi ekrani nisu bili bez mana. Instrukcije i napomene za korisnika pisali su se na ekranu u formi prompta ili kodova kao npr. PR i SC.

A screenshot of a computer terminal window titled "THE CAR RENTAL COMPANY". The interface is organized into sections: "RENTER >>" with fields for "Name" (with a long line) and "Telephone" (with a line ending in three dashes); "LOCATION >>" with fields for "Office" (with a long line), "Pick-up Date" (with a line ending in three dashes), and "Return Date" (with a line ending in three dashes); and "AUTOMOBILE >>" with fields for "Class" (with a line), "Rate" (with a line), and "Miles Per Day" (with a line). Below these sections is a note: "The maximum allowed miles per day is 150.". At the bottom of the window, a command prompt shows "Enter F1=Help F3=Exit F12=Cancel".

Slika 5.2: Ekran u 1980 godini [12]

RAZVOJ PROGRAMSKIH RJEŠENJA

Napredak grafike je doveo do još jedne prekretnice u dizajnu ekrana, kao što je prikazano na slici 5.3. Okviri za vizualno napredno grupiranje su postali raspoloživi. Dugmad (*buttons*) i meniji za implementiranje komandi su zamijenili funkcionalne tipke. Pojavljuju se različite veličine, stilovi i boje fontova. Pojavljuju se kontrole (*listbox*, *dropdown box*) na poljima koje omogućavaju izbor jedne od ponuđenih opcija.

The screenshot shows a window titled "THE CAR RENTAL COMPANY". It contains three main sections: "RENTER", "LOCATION", and "AUTOMOBILE". The "RENTER" section has fields for "Name" (with a long input field) and "Telephone" (with three small input fields). The "LOCATION" section has fields for "Office" (input field), "Pick-up Date" (a 4x2 grid of input fields), and "Return Date" (a 4x2 grid of input fields). The "AUTOMOBILE" section has fields for "Class" (input field), "Rate" (input field with a dropdown arrow), and "Miles Per Day" (input field with a dropdown arrow). At the bottom are four buttons: "OK", "Apply", "Cancel", and "Help".

Slika 5.3: Ekran u 1990 godini [12]

I na kraju slijedi prikaz ekrana sa elementima i dizajnom koji je primjetan na većini današnjih ekrana (slika 5.4).

The screenshot shows a web page for "Min Escrow". The top navigation bar includes "Početna", "Log in", and "Registracija". The left side features a "Login Forma" with fields for "Email" (user@asmal.com) and "Lozinka" (lozinka), and buttons for "Prijavi se" and "Zaboravili lozinku". The right side features a "Registracija" form with fields for "Email", "Potvrdite Email", "Lozinka", "Potvrdite Lozinku", "Ime", "Prezime", "Adresa", "Grad", "Država", "ZIP kod", and "Telefon", all with corresponding input fields. A note at the bottom left says: "Min Escrow je mini aplikacija za kupo-prodaju. Omogućava dvjema stranama, kupcu i prodavcu, da kreiraju i zaključu transakciju. Podržane su dva tipa transakcija: 1. Domenske transakcije, preko kojih možemo trgovati domenama. 2. Regularna roba, gdje možemo trgovati različitim robom. Da bi korisnici mogli koristiti aplikaciju, potrebno je da imaju kreiran korisnički račun. To mogu utvrditi tako što će se registrirati na linku 'Registracija'. Nakon registracije, korisnik se može ulogovati i koristiti aplikaciju." A green "Kreiraj account" button is located at the bottom right of the registration form.

Slika 5.4: Ekran 2000 godine

5.2. GUI kontrola: Windows (prozor) – osnovne karakteristike

Prozor je oblast na ekranu, određena granicama, koja određuje pogled na neku logički povezanu cjelinu. Ekran može sadržavati jedan, dva ili više prozora različitih dimenzija unutar svojih granica.

5.2.1. Karakteristike prozora

Prozor posjeduje mnoge karakteristike. Neke od njih su:

- Naziv.
- Veličina izražena kroz visinu i širinu (koje se mogu mijenjati).
- Stanje, aktivno ili neaktivno (sadržaj se može mijenjati samo u aktivnom prozoru).
- Vidljivost (prozor može biti u potpunosti vidljiv, djelimično ili potpuno sakriven ispod nekog drugog prozora).
- Lokacija.
- Prezentacija, tj. uređenost u odnosu na druge prozore. Može biti dijeljeni, preklapajući ili kaskadni.
- Funkcija, zadatak ili aplikacija kojoj je dodijeljen.

Tipičan prozor može biti sastavljen od velikog broja elemenata. Neki elementi se pojavljuju na svim prozorima, drugi samo na određenim tipovima prozora ili samo pod određenim uvjetima. U cilju konzistentnosti, zajednički elementi moraju uvijek biti na istom mjestu na prozoru.

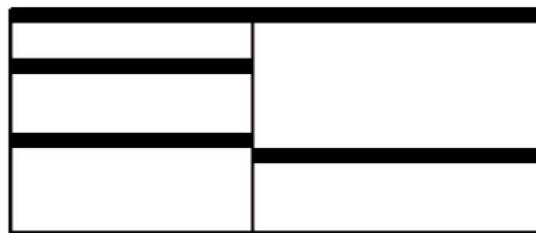
Npr: Kontrole za veličinu prozora  locirane u desnom uglu linije naziva prozora postavljaju se konzistentno na svim prozorima. Prva sa lijeve strane je kontrola za minimiziranja prozora. Srednja je kontrola za maksimalno uvećanje prozora. Kontrola x služi za zatvaranje prozora. Ako prozor ne podržava ove komande, ne treba ih ni prikazivati. Većina komponenti i karakteristika prozora uvodi se detaljnije u okviru Windows Formi.

5.2.2. Stilovi prezentacije prozora

Stil prezentacije prozora se odnosi na njegov odnos prema ostalim prozorima. Postoje dva osnovna stila, poznata kao dijeljeni i preklapajući.

Dijeljeni prozori

Dijeljeni prozori se pojavljuju u jednoj ravni na ekranu, prošireni maksimalno do granica prikaza (slika 5.5). Mnogi sistemi obezbjeđuju dvodimenzionalno dijeljenje prozora i njihovo podešavanje po širini i po visini.

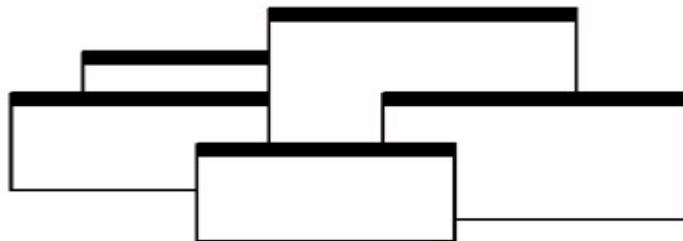


Slika 5.5: Dijeljeni prozori

Prednost dijeljenih prozora je što su uvijek vidljivi, čime su i sve informacije koje se prikazuju uvijek vidljive i dostupne. Jednostavniji su, prema raspoloživim studijima, za nove i neiskusne korisnike i doprinose boljim performansama korisnika, za zadatke koji zahtijevaju manje manipulacije sa prozorima. Jedan od nedostataka ovih prozora je što je ograničen broj prozora koji može biti prikazan u raspoloživoj oblasti na ekranu. Povećanjem broja prozora, prozori mogu postati veoma mali. Konfiguracija prozora može ne zadovoljavati korisnikove potrebe.

Preklapajući prozori

Preklapajući prozori, prikazani na slici 5.6, postavljaju se jedan preko drugog. Korisnik može kontrolirati ove prozore, kao i ravan u kojoj se pojavljuju. Većina računarskih sistema danas koristi ovaj stil prozora.



Slika 5.6: Preklapajući prozori

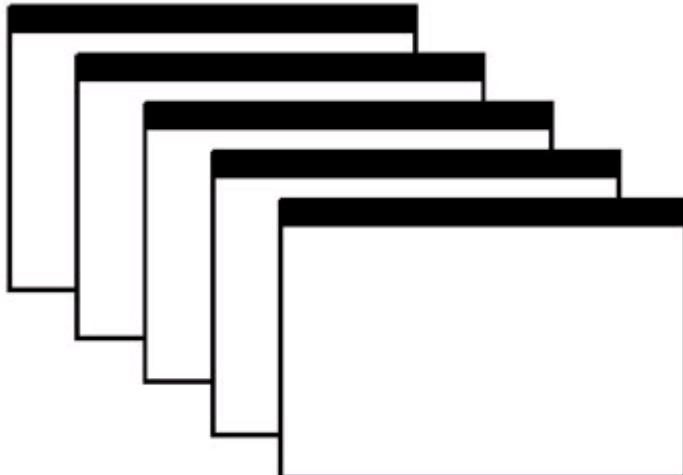
Prednosti ove vrste prozora su sljedeće:

- Vizuelno, njihov izgled je trodimenzionalan.
- Korisnik može da organizira prozore kako odgovara njegovim potrebama.
- Prozori mogu biti različitih veličina i postavljeni na stalnim pozicijama.
- Prostor na ekranu nije problem jer prozori mogu biti postavljeni jedan preko drugog.
- Dovode do boljih korisničkih performansi za zadatke koji zahtijevaju manipulaciju prozorima.

Neki od nedostataka ovih prozora su: prozor sa potrebnim informacijama može biti ispod drugih prozora, ekran može biti pretrpan, neiskusni korisnici se mogu teže snaći.

Kaskadni prozori

Specijalni tip preklapajućih prozora su kaskadni prozori. Kod ovog tipa prozora prozori se automatski aranžiraju, tako da je svaki prozor malo pomaknut u odnosu na drugi, kako je prikazano na slici 5.7.



Slika 5.7: Kaskadni prozori

Prednosti kaskadnog prikaza prozora su sljedeći:

- Prozor nikad nije potpuno pokriven.
- Pomak prozora u frontalni pogled je puno lakši.
- Obezbjedena je jednostavnost u vizuelnom predstavljanju.

Primarni i sekundarni prozori

Korisnički zadaci trebaju biti strukturirani kroz seriju prozora. Prozori se često dijele i na primarne i sekundarne prozore.

Primarni prozor

Primarni prozor se prvi pojavljuje na ekranu kad se pokrene neka aktivnost ili akcija. Treba da predstavlja okosnicu za funkcijeske komande i podatke i da omogući najviši nivo konteksta za ovisne, sekundarne prozore. Često se naziva i aplikacijski glavni prozor.

Sekundarni prozor

Sekundarni prozori su pomoćni prozori. Mogu ovisiti od primarnog prozora ili biti prikazani neovisno od primarnog prozora. Ovisni sekundarni prozor može biti prikazan samo sa komandnog interfejsa primarnog prozora. Najčešće je povezan sa jednim objektom podatka i pojavljuje se preko aktivnog prozora. Može se pomijerati i skrolirati. Mnogi sistemi dozvoljavaju upotrebu više sekundarnih prozora da bi se završio neki zadatak. Neovisni sekundarni prozor može biti otvoren ili zatvoren neovisno od primarnog prozora.

Iako sekundarni prozori imaju mnoge karakteristike primarnog prozora, oni se razlikuju od primarnog prozora po ponašanju i upotrebi. Sekundarni prozori se koriste da se izvrše dodatni i podčinjeni zadaci. Microsoft Windows posjeduje npr. slijedeće sekundarne prozore: dijalog prozore (*dialog boxes*), prozore za postavljanje osobina (*property sheets*), prozore za poruke (*message boxes*), prozore za boje (*palette windows*), iskakajuće prozore (*pop-up*) prozore.

5.3.Windows Forme i dugme kontrola

Forma je grafički element koji se koristi za prezentaciju prozora, kreiranje sadržaja prozora i upravljanje ponašanjem prozora. Forma se sastoji od kontrola sa grafičkom predstavom (dugme, labela, tekst polje,...) i kontrola koje nemaju vizuelnu predstavu i koje se često nazivaju komponente (npr. komponenta za pristup bazi podataka, validator podataka,...). Uz formu i sve kontrole forme vežu se događaji (*event*). Događaji se uglavnom generiraju pomjeranjem miša ili upotrebom tastature i pri tome se izvršavaju odgovarajuće akcije koje su specifično napisane od strane programera.

Windows Forms aplikacija sastoji se od jedne ili više formi na kojima su razne kontrole. Detalji vezani za kontrole, komponente i događaje iznose se u ovom i sljedeća tri poglavlja. Microsoft Windows .NET Framework pruža skup klasa za izgradnju GUI aplikacija u okviru `System.Windows.Forms` imenovanog prostora. U okviru udžbenika prikazati će se samo neke od tih klasa.

5.3.1. Kreiranje forme iz konzolne aplikacije

Da bi se u konzolnoj aplikaciji koristile klase koje omogućavaju rad sa formama i kontrolama na formi potrebno je uključiti sljedeće reference: `System;` `System.Windows.Forms;System.Drawing.`

Npr. Ukoliko je potrebno da se iz konzolne aplikacije kreira forma koja ima naziv „Forma iz konzolne aplikacije“ veličine 300, 300, to bi se moglo postići kodom 5.1:

```
using System;
using System.Drawing;
using System.Windows.Forms;

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

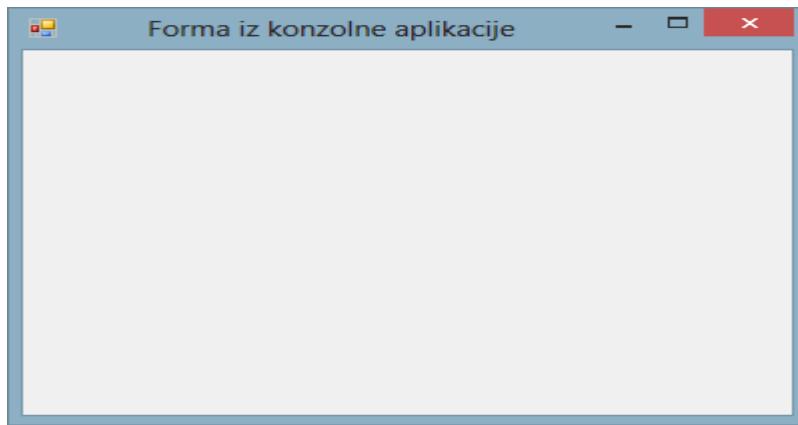
            Form f = new Form(); // kreiranje objekta (f) tipa klase Form
            f.Size = new Size(300,300) ;// postavljanje veličine forme
            f.Text = "Forma iz konzolne aplikacije";
            // postavljanje naslova forme
            Application.Run(f);

        }
    }
}
```

Kod 5.1: Kreiranje forme iz konzolne aplikacije

U kodu 5.1 za rad sa formama koristi se klasa `Form`, koja ima svoje attribute (npr. `Size`, `Text`) i metode.

Izvršavanjem iznad prikazanog koda dobija se forma (prozor):



Izgled forme kreirane iz konzolne aplikacije

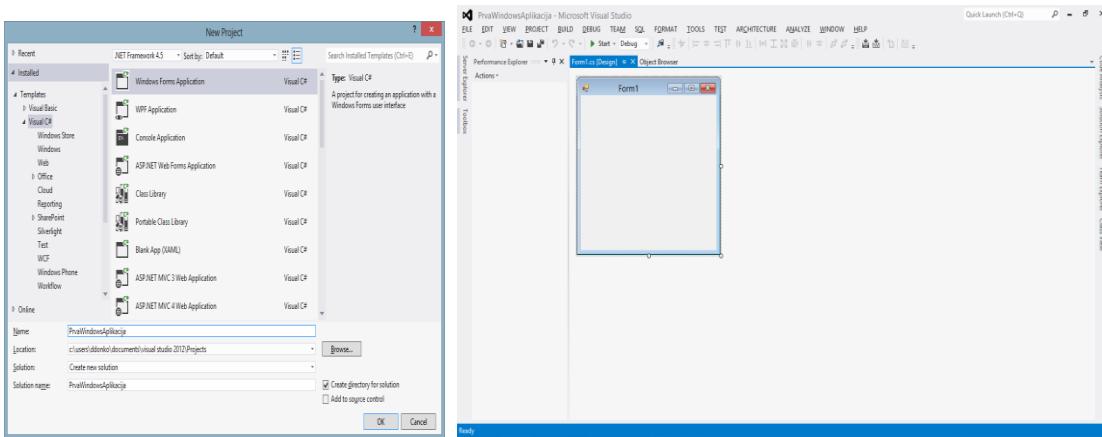
5.3.2. Kreiranje formi sa dizajnerom

Osim što se može pisati kod za generiranje formi korištenjem biblioteka može se koristiti i dizajner za kreiranje korisničkog interfejsa. Microsoft Visual Studio .NET podržava konstrukciju grafičkog GUI-a korištenjem dizajnera. Kreiranjem Windows aplikacije postavlja se početna forma na ekran. Dizajner omogućava jednostavni *drag-drop* komponenti od `Toolboxa` (grafički prikaz svih kontrola) na formu. Editor za svojstva (`Properties editor`) omogućava postavljanje raznih osobina forme (boja, font, naslov, okvir,...). Na osnovu onoga što se uradi pomoću dizajnera generira se automatski kod, koji kreira formu za vrijeme izvršavanja.

Slijedi kratki opis kako se kreira i prikazuje jednostavna forma, pomoću dizajnera. Objasnjava se na primjeru Visual Studio .NET platforme ali i mnogi drugi vizualni alati rade na sličan način.

1. Prilikom kreiranja projekta bira se Windows Forms Application (slika 5.8a). Nakon izbora projekta i dodjele imena dobija se prozor za dizajniranje forme koji je prikazan na slici 5.8b:

RAZVOJ PROGRAMSKIH RJEŠENJA

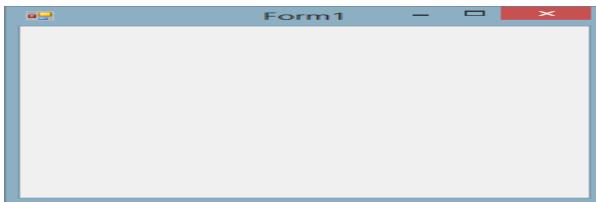


Slika 5.8a: Kreiranje projekta

5.8b: Početni prozor forme u dizajn modu

Prozor predstavlja jednu formu na koju se mogu dodati razne kontrole. Ovoj formi mogu se postaviti i razne osobine.

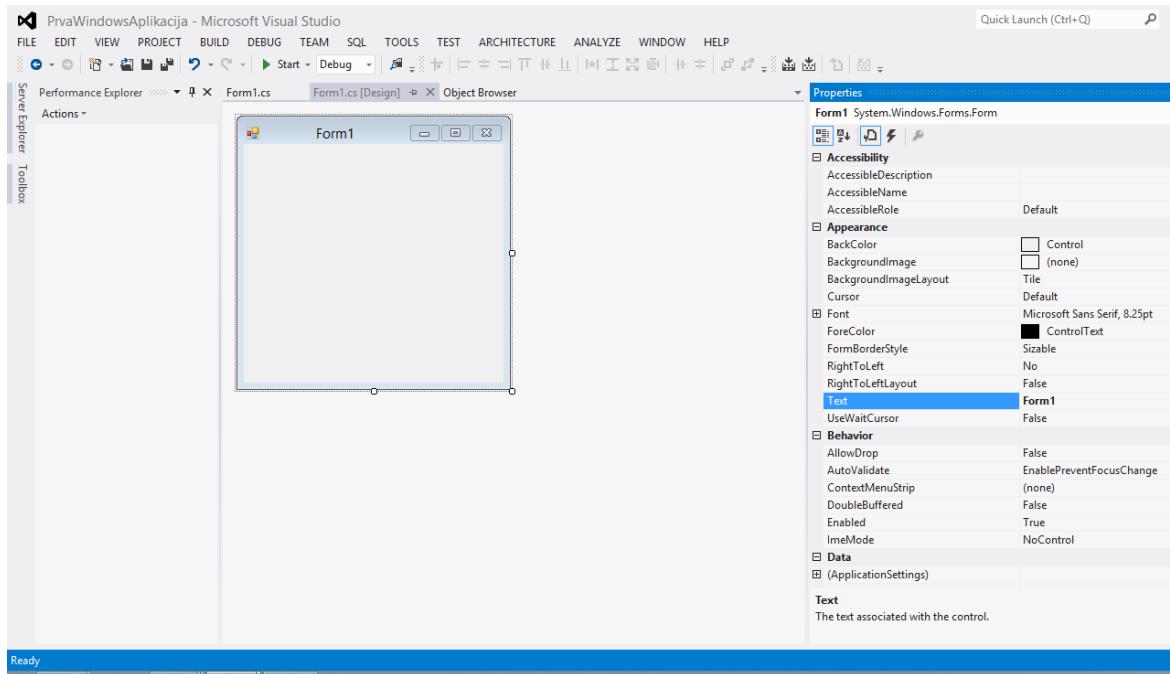
Izvršavanjem projekta dobija se forma slična formi kreiranoj iz iznad prikazane konzolne aplikacije:



Korištenje prozora za postavljanje osobina forme

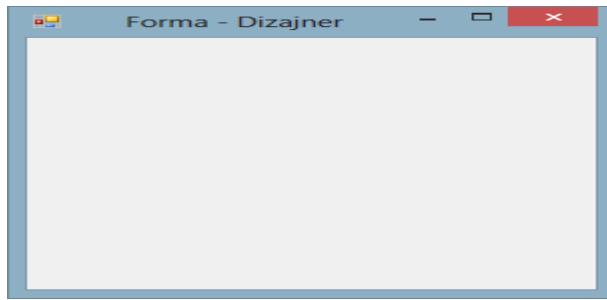
Za dodavanje osobina formi može se pisati kod, ili koristiti Properties editor koji omogućava grafički pristup najčešće korištenim osobinama. Na slici 5.9 prikazan je Properties editor sa osobinama za formu. Izborom Text osobine može se promijeniti naziv forme (Form1 u Forma-Dizajner), kao što je prikazano na slici ispod.

RAZVOJ PROGRAMSKIH RJEŠENJA



Slika 5.9: Prozor za postavljanje osobina

Na sličan način se mogu mijenjati i druga svojstva – npr: `FormBorderStyle` (stil okvira) za koju je vezana lista ponuđenih vrijednosti. Poslije izvršenih promjena nakon pokretanja aplikacije dobija se forma sa izgledom:



Relacije između formi su slične već opisanim relacijama prozora: svaka forma može kreirati drugu formu, forme se mogu neovisno jedna od druge minimizirati, zatvarati, može postojati i roditelj/dijete (*parent/child*) relacija među formama, aplikacijska glavna forma.

Uz formu se vežu i metode, kao i događaji-dešavaju se pri nekoj korisničkoj interakciji sa formom ili pri nekoj aktivnosti same forme (npr. pri podizanju forme dešava se događaj `Load`).

RAZVOJ PROGRAMSKIH RJEŠENJA

Tabela 5.2 prikazuje osnovne zajedničke **form osobine, metode i dogadaje**.

Form osobine (properties), metode (methods) i dogadaji(events)	Opis
Osobine	
AutoScroll	Boolean vrijednost koja dozvoljava ili nedozvoljava pomjeranja (<i>scrollbars</i>) sadržaja prozora.
FormBorderStyle	Stil okvira (<i>border</i>) za formu.
Font	Font teksta za prikaze na formi.
Text	Tekst u naslovu forme.
Metode	
Close	Zatvara formu i oslobađa sve resurse koje se koristi za kontrole i komponente forme.
Hide	Skriva formu, ali ne uništava formu i ne oslobađa njene resurse.
Show	Prikazuje formu.
Dogadaji	
Load	Dešava se prije nego što se forma prikaže.

Tabela 5.2. osobine (*properties*), metode (*methods*) i dogadaji (*events*) `Form` klase

Generirani program/kod

Za Windows aplikaciju kreiranu dizajnerom automatski se generira kod. Ispod je prikazan kod 5.2 za projekat `PrvaWindowsAplikacija`.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace PrvaWindowsAplikacija
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Kod 5.2: Generirani kod za projekat PrvaWindowsAplikacija

Dizajnirana forma kodom 5.2 je instanca klase `Form1`, koja je naslijeđena od `Form` klase koja se nalazi u `System.Windows.Form`. Klasa `Form1` se implementira kao `partial` klasa.

Automatski generiran kod 5.3 za `InitializeComponent()` sadrži iskaze za postavljanje osobina.

```
private void InitializeComponent()
{
    this.SuspendLayout();

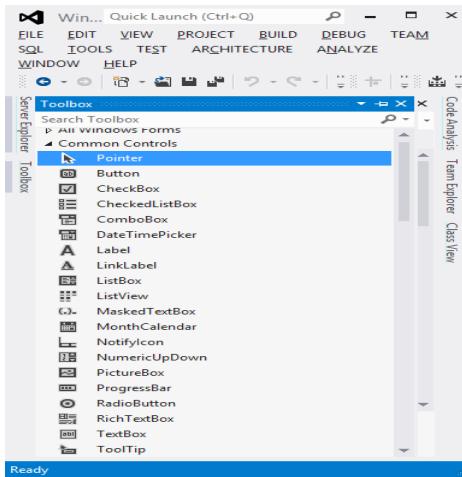
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.BackColor = System.Drawing.SystemColors.Control;
    this.ClientSize = new System.Drawing.Size(398, 261);
    this.FormBorderStyle =
System.Windows.Forms.FormBorderStyle.Fixed3D;
    this.Name = "Form1";
    this.Text = "Forma - Dizajner";
    this.Load += new System.EventHandler(this.Form1_Load);
    this.ResumeLayout(false);

}
```

Kod 5.3: Automatski generiran kod za postavljanje osobina forme

5.3.3. GUI kontrole

Da bi se kreirala Windows aplikacija, prvo se kreira forma (Windows Form), zatim se postavljaju njene osobine kako je objašnjeno u prethodnoj sekciji. Nakon toga se dodaju kontrole na formu i postavljaju njihove osobine, što se objašnjava u ovoj sekciji. Uz kontrole se implementiraju metode koje odgovaraju na događaje uzrokovane interakcijom sa korisnikom (objašnjava se u sljedećoj sekciji). Kontrole i komponente se mogu dodavati programskim kodom, korištenjem odgovarajućih klasa ili korištenjem Toolboxa (slika 5.10) u sastavu dizajnera formi.



Slika 5.10: Toolbox: Komponente i kontrole za Windows Forms

Forma je ustvari kontejner za kontrole i komponente. Da bi se dodala kontrola na formu, selektira se komponenta/kontrola sa Toolboxa i postavlja se na formu. Visual Studio generira kod koji instancira objekat tipa klase za tu kontrolu i postavlja osnovne osobine objekta. Ako se komponenta ili kontrola promijeni, izbriše sa forme generirani kod za kontrolu se mijenja ili briše. Kontrole su locirane u `System.Windows.Forms` imenovanom prostoru. U nastavku slijedi kratki pregled jedne od najčešće pojavljivane kontrole na formi a to je kontrola dugme (*button*) u cilju ilustracije postavljanje te kontrole na formu i pisanja odgovarajuće metode u slučaju interakcije sa tom kontrolom.

5.3.4. Dugme (*button*) kontrola

Osnovna dugme (*button*) kontrola je kontrola pravougaonog oblika sa labelom/tekstom unutar granica pravougaonika koja indicira željenu akciju pri čemu se labela može sastojati od teksta i grafike. Najčešće se koristi za: pokretanje neke akcije/komande (komandni *buttoni*), prikaz opcija (*toolbar*), prikaz *pop-up* menija.

RAZVOJ PROGRAMSKIH RJEŠENJA

Primjer komandnih dugmadi koji uzrokuju izvršenje akcije:   

Primjer trake dugmadi (*toolbar*) bez labela sa ilustrativnom slikom koja odgovara namjenjenoj komandi izvršavanja preko dugmeta:



Dugme koje vodi kaskadnom dijalogu uključuje (...) nakon labele je , dok dugme koje vodi ka meniju se označava sa  ili dijalogu sa .

Prednosti dugme kontrole su: uvijek vidljivi, jednostavni za korištenje, mogu biti logički organizirani, omogućavaju brzi pristup željenoj akciji, pružaju dodatno značenje akciji koja će biti izvršena, mogu posjedovati 3-D pojavu.

Neki nedostaci dugme kontrole su što koriste dosta prostora na ekranu, zahtjeva se pomjeranje pokazivača da bi se odabralo odgovarajuće dugme, njihov broj je ograničen.

Sve vrste dugmadi su izvedene iz `Button` klase (namespace `System.Windows.Forms`), koja definira osnovne zajedničke karakteristike vezane za `Button` kontrolu. U tabeli 5.3 su prikazane neke često korištene osobine i događaji vezani za klasu `Button`.

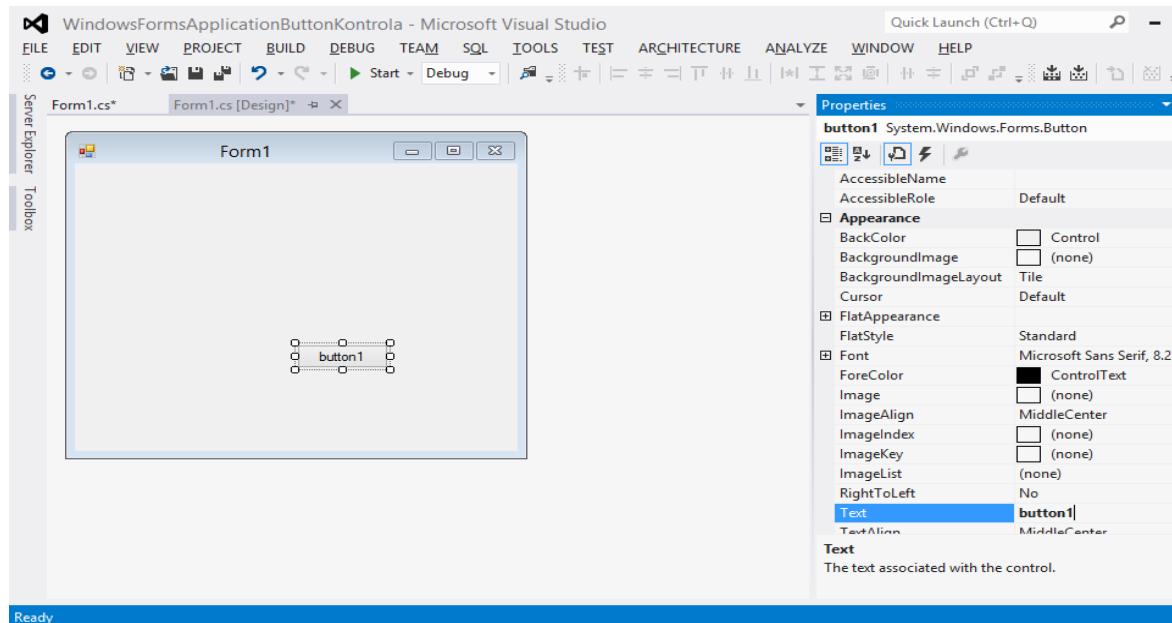
Dugme (<code>Button</code>) Osobine događaji	i Opis
Osobine	
Text	Specificira tekst na <code>Button</code> kontroli.
<code>FlatStyle</code>	Specificira izgled dugmeta: <code>Flat</code> (prikaz bez trodimenzionalnog izgleda), <code>Popup</code> (promjena izgleda pri pomjeranju miša preko kontrole), <code>Standard</code> (trodimenzionalni oblik kontrole, podrazumijevana vrijednost) i <code>System</code> (pojava dugmeta pod kontrolom operativnog sistema).
Događaj	
<code>Click</code>	Aktivira se klikom na dugme.

Tabela 5.3: Često korištene osobine i događaji dugme kontrole

RAZVOJ PROGRAMSKIH RJEŠENJA

Postavljanje kontrola na formu

Sa Toolboxa se odabira kontrola i prebacuje na formu. Na slici 5.11 je prikazana forma sa dugme kontrolom i editorom za postavljanje osobina dugme kontrole.



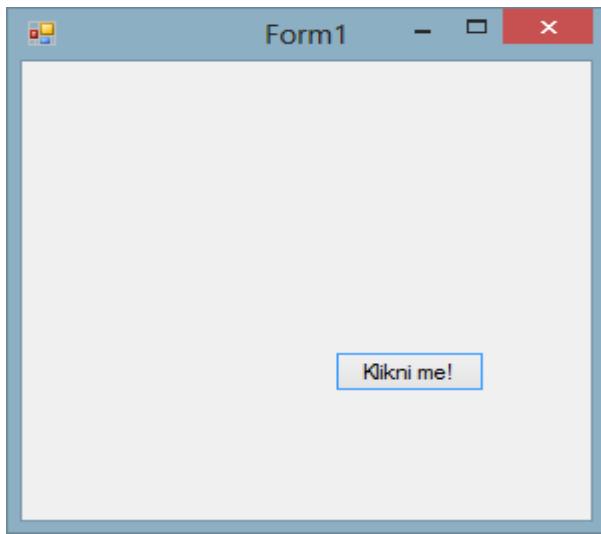
Slika 5.11: Dugme na formi i editor za postavljanje osobina

Naprimjer, ukoliko se želi napisati tekst Klikni me! koji će opisati namjenu dugmeta (trenutno je to tekst `button1`) tada se mijenja osobina `Text`.

Objekat (varijabla) koji odgovara dugme kontroli ima naziv `button1` (vidi se na Properties) editoru. Preporučljiva je promijena naziva objekta. Vrši se pomoću `Name` osobine. Konvencija imenovanja je da prve riječi naziva kontrole opisuju namjenu kontrole, i da se taj naziv završava se sa tipom kontrole. U našem slučaju dobar naziv za objekat bi bio `klikButton`, `porukaButton`, `prukaKlikButton`.

Izvršavanjem Windows Form aplikacije prikazane iznad dobija se forma kao na slici ispod.

RAZVOJ PROGRAMSKIH RJEŠENJA



Windows Form aplikacija sa dugme kontrolom

Na ovoj formi dugme kontrola ima samo vizuelno namjenu. Ukoliko bi željeli da se implementira programski kod koji će se izvršiti kada korisnik klikne na dugme potrebno je naučiti kako upravljati događajima (*eventima*).

5.4. Model upravljanje događajima (Event Handling Model)

Korisnik u interakciji sa aplikacijskim korisničkim interfejsom (GUI-om) indicira zadatke koje je potrebno izvršiti. Korisnička interakcija koja uzrokuju da aplikacija izvrši neki zadatak predstavlja događaj. Ta korisnička interakcija može biti npr: klik na dugme, kucanje teksta u tekstualno polje, izbor stavke iz menija, zatvaranje prozora, pomjeranje miša. Metoda koja izvršava neki zadatak (programski kod) kao odgovor na događaj naziva se metoda za upravljanje događajem (*event handler*). Proces odgovaranja na događaje je poznat kao proces upravljanja događajima (*event handling*). Grafički korisnički interfejs (GUI) je vođen događajima (*event driven*).

Metoda za upravljanje događajima (*event handler*) je obični delegat (u poglavlju 4 obrađeni su delegati) sa dva parametra.

Definicija metode za upravljanje događajem u C# je:

```
public delegate void EventHandler(  
    Object sender,  
    EventArgs e  
)
```

Svaka metoda za upravljanje događajem (*event handler*) prima 2 parametra. Prvi parametar se naziva `sender` i to je referenca na objekt koji generira događaj. Drugi parametar, je referenca na događaj koji je tipa `EventArgs` i koji se tipično naziva `e`. Ovaj objekat sadrži dodatne informacije o događaju koji se dešava. `EventArgs` je bazna klasa svih klasi koje predstavljaju informacije o događajima. Metode za upravljanje događajima ne vraćaju povratnu vrijednost. Događaji su dizajnirani za izvršavanje koda baziranog na akciji i vraćaju kontrolu glavnom programu.

5.4.1. Programiranje događaja za dugme kontrolu

U sekciji 5.3.1. pokazano je kako se korištenjem klase `Form` kreira forma. U ovoj sekciji će se pokazati kako se bez dizajnera na formu dodaje kontrola dugme i kako se odgovara na događaj uzrokovani klikom na dugme. Proces odgovaranja kontrole na događaj radi tako da kada se pokrene neki događaj kontrole (npr. klik na dugme) poziva se odgovarajuća metoda (registrovana kao delegat kojeg poziva taj događaj) i koja izvršava kod za taj događaj (npr. nakon pritiska dugmeta ispisuje poruku).

RAZVOJ PROGRAMSKIH RJEŠENJA

Primjer dat sa kodom 5.4 na formu veličine 300x300 sa naslovom "Forma - ilustracija događaja klik na dugme" postavlja kontrolu dugme sa tekstrom Klikni unutar nje, veličine 50x30 na lokaciju 120x120. Programirana je i metoda (dugme_pritisnuto) za upravljanje dogadajem koji se desi ukoliko se klikne na dugme. U slučaju aktiviranja događaja pomoću MessageBox.Show (detaljnije objašnjeno u poglavlju 6) se u okviru novog prozora piše poruka.

```
using System;
using System.Windows.Forms;

class Program {

    static void dugme_pritisnuto(Object o, EventArgs e) // event handler-metoda
        //izvršava se kada se desi događaj-klik na dugme
    {
        MessageBox.Show("Pisem poruku");
    }

    static void Main() {
        Form f = new Form();
        f.Size = new System.Drawing.Size(300,300);
        f.Text = "Forma - ilustracija događaja klik na dugme";

        // 1.kreiranje dugmeta-buttona, klikButton je instanca klase Button
        Button klikButton = new Button();

        //2. Postavljanje osobina za kreirano dugme (teksta, lokacije, veličine)
        klikButton.Text = "Klikni";
        klikButton.Location = new System.Drawing.Point(120, 120);
        klikButton.Size = new System.Drawing.Size(50,30);

        //3. Definiranje događaja sa značenjem da kada se klikne na dugme pozovi ----
        // metodu dugme_pritisnuto
        klikButton.Click += new EventHandler(dugme_pritisnuto);

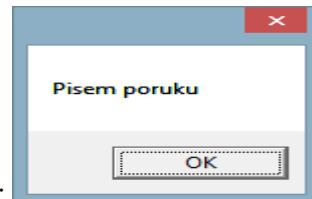
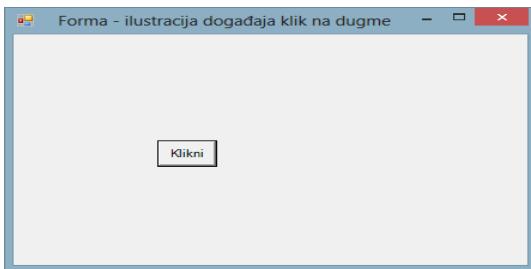
        f.Controls.Add(klikButton); // 4. dodavanje dugmeta na formu

        Application.Run(f);
    }
}
```

Kod 5.4: Programske funkcije za kreiranje forme, dugmeta i događaja Click na dugme

RAZVOJ PROGRAMSKIH RJEŠENJA

Izvršavanjem koda 5.4 dobija se forma sa sljedećim izgledom:



Nakon klika na dugme pojavljuje se novi prozor sa porukom:

Upravljanje događajem u primjeru iznad u suštini radi tako da kada se pritisne (klikne) dugme na formi, Windows aplikaciji se pošalje poruka da je pritisnuto dugme. Aplikacija prima tu poruku i pokreće događaj sa nazivom `Click()`, događaj `Click` prolazi kroz listu registrovanih *handlera* na delegate i ustanovljava da ima samo jedan registrovan delegat na metodu `dugme_pritisnuto` te ga poziva i prosljeđuje mu parametre.

5.4.2. Kreiranje događaja sa dizajnerom

Pomenuti `Click` događaj se može u dizajn modu kreirati dvostrukim klikom na postavljeno dugme na formi. Automatski generirani kod za taj slučaj je prikazan sa kodom 5.5.

```
namespace WindowsFormsApplication4x
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
        }
    }
}
```

Kod 5.5: Automatski generiran kod za događaj `Click` na dugme

RAZVOJ PROGRAMSKIH RJEŠENJA

Programski kod (u ovom slučaju 5.6) koji se treba izvršiti dodaje se u metodu za upravljanje događajem:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Pisem poruku");
}
```

Kod 5.6: Metoda za upravljanje događajem Click

Konvencija koju Visual Studio primjenjuje je da se metode za upravljanja događajima (*event handler*) imenuje kao:

controlName_eventName tj. nazivKontrole_nazivDogađaja (npr: `button1_Click`).

Osnovna podrazumijevana metoda za upravljanje događajem neke kontrole uglavnom se dobije dvostrukim klikom na kontrolu postavljenu na formu. Tipično, kontrole mogu generirati više različitih tipova događaja za koje se vežu različite metode. Već je kreirana `Click` metoda za dugme kontrolu. `Click` je podrazumijevani (*default*) događaj za dugme kontrolu. Aplikacija može također, zahtijevati i druge događaje za to dugme, naprimjer `MouseHover` događaj, koji se dešava kada pointer miša ostane pozicioniran preko dugme kontrole.

Dodatne metode za upravljanje događajima se mogu uspostaviti i preko prozora za postavljanje osobina (Properties Window - Events ikona). Za svaki događaj može se napisati metoda za upravljanje događajem. Najvažnije je da se shvati da su GUI objekti ustvari instance klase, koje imaju svoje osobine i metode i koje se mogu koristiti kao i bilo koje druge klase (npr. *Uposlenik*).

Rezime:

U okviru ovog poglavlja dati su osnovni pojmovi vezani za grafički korisnički interfejs. Urađen je kratki hronološki pregled bitan za pojavu grafičkog interfejsa kao i hronološki pregled ekrana. Kreiranje GUI kontrola i komponenti može se programirati pomoću odgovarajućih klasa iz konzolne i Windows aplikacije, a može se koristiti i dizajner koji je dostupan u okviru mnogih okruženja. U ovom poglavlju je prikazano kreiranje forme i postavljanje dugmeta na formu na oba načina. Grafički korisnički interfejs služi za interakciju korisnika sa programskim rješenjem. Postizanje interakcije može se odvijati različitim događajima, koji kada se dese povezuju se sa metodama za upravljanje događajem. Pored forme i dugmeta postoji još veliki broj kontrola koje se izučavaju u narednim poglavljima.

Pitanja i zadaci za samostalni rad:

- 1.Šta je GUI, koja je njegova osnovna namjena?
- 2.U tabeli 5.1. dat je kratki historijski pregled bitnih proizvoda koji su uticali na razvoj korisničkog interfejsa. Samostalno, istražite koje su prednosti donijeli i koji su još bitni proizvodi za razvoj današnjeg GUI-a.
3. Navedite neke GUI kontrole.
4. Objasnite šta je forma, koje su njene osnovne karakteristike (osobine) i koji je osnovni podrazumijevani događaj vezan za formu?
- 5.Objasnite šta je to programiranje vođeno događajima?
- 6.Koji je najčešći način nastanka ('okidanja') događaja?
- 7.Objasniti šta je metoda za upravljanje događajem i koji su njeni osnovni parametri?
- 8.Objasnite pojam upravljanja događajima u kontekstu pisanja programskih rješenja.
- 9.Navedite preporuke imenovanja za varijable koje se vežu sa GUI kontrolama i metodama za upravljanje događajima.
- 10.Analizirajte osobine i događaje vezane za kontrolu `Form` i `Button` kroz Visual Studio okruženje.
11. Kreirajte formu po želji sa dva dugmeta za koja se vežu događaji `Click` iz konzolne aplikacije (sve bez dizajnera) i iz Windows Forms aplikacije (uz upotrebu dizajnera).

Poglavlje 6: Pregled GUI kontrola

U okviru ovog poglavlja prikazane su GUI kontrole: labela, radio dugmad, *check box*, tekst polje, *group box*, te za njih navedene osnovne osobine i događaji uz odgovarajuće primjere. Obradena je i interakcija sa korisnikom i događaji tastature i miša kroz sljedeće osnovne tematske jedinice:

- 1.GUI kontrole
- 2.Interakcija sa korisnikom

6.1. GUI kontrole

Slijedi prikaz labele, tekst polja, kontrola za izbor jedne ili više opcija i kontrola za grupiranje.

6.1.1.Labela

Labela je kontrola koja se koristi da pruži razne opise i informacije na formi. Korištenje labela uglavnom se sastoji od kreiranja i postavljanja njihovih osobina. Tabela 6.1 pokazuje najčešće korištene osobine `Label` klase. Uz labelu se može postaviti i slika.

Label Osobine	Opis
Font	Font teksta labele.
Image	Slika uz labelu.
Text	Tekst labele.
TextAlign	Poravnanje teksta na labeli horizontalno (lijevo, centrirano ili desno) i vertikalno (vrh, sredina ili dno).

Tabela 6.1: Dio osobina `Label` kontrole i klase

RAZVOJ PROGRAMSKIH RJEŠENJA

6.1.2. TextBox kontrola

Namjena tekst polje (*textbox*) kontrole je da dozvoli prikaz, unos, uređivanje tekstualnih informacija. Najčešće se koriste za unos podataka koje je teško kategorizati. Podržana je sa `TextBox` klasom. Tekst polje može biti jednolinijsko (*single-line*) ili višelinijsko (*multiple-line*) polje, što se određuje sa `Multiline` karakteristikom. Jednolinijsko tekstualno polje može se koristiti naprimjer za: unos/ispis vrijednosti za string polja klase, prikaz rezultata, unos imena datoteke, unos komande.

Višelinijsko tekstualno polja može se koristiti za kreiranje ili čitanje poruka elektronske pošte, prikaz i uređivanje tekstualnih datoteka i ostale slične namjene.

U tabeli 6.2 je lista često korištenih osobina i događaja za kontrolu `TextBox`.

TextBox osobine i događaji	Opis
Osobine	
AcceptsReturn	Ako je <code>true</code> u višelinijskom <code>TextBox</code> u, pritiskom na <code>Enter</code> kreira se nova linija.
Multiline	Ako je <code>true</code> , <code>TextBox</code> se sastoji od više linija. Podrazumijevana vrijednost je <code>false</code> .
PasswordChar	Ako je ova osobina postavljena na karakter, <code>TextBox</code> postaje <i>password</i> polje, i specificirani karakter maskira svaki karakter koji korisnik kuca, u suprotnom <code>TextBox</code> prikazuje tekst koji se kuca.
ReadOnly	Ako je <code>true</code> , <code>TextBox</code> ima sivu pozadinu, i tekst se ne može mijenjati. Podrazumijevana vrijednost je <code>false</code> .
ScrollBars	Za višelinijsko tekst polje, ova osobina indicira koje pokretne trake se pojavljuju (<code>None</code> , <code>Horizontal</code> , <code>Vertical</code> ili <code>Both</code>).
Text	Sadržaj <code>TextBox</code> kontrole.
Događaji	
TextChanged	Generira se kada se tekst mijenja u <code>TextBox</code> u (tj. kada korisnici dodaju ili brišu karaktere).

Tabela 6.2: Osobine i događaji `TextBox` kontrole

RAZVOJ PROGRAMSKIH RJEŠENJA

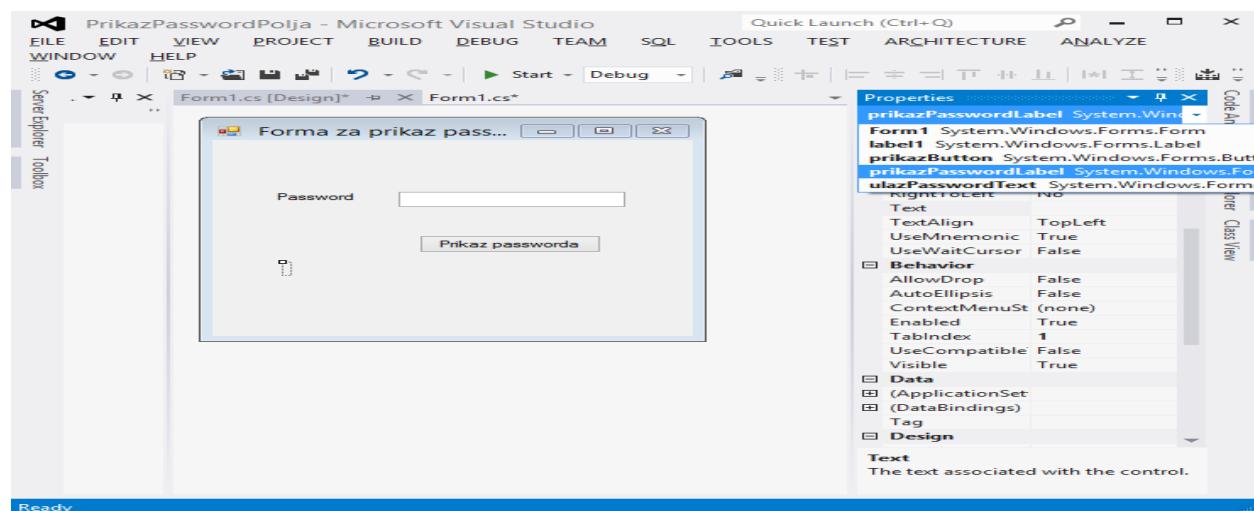
System.Windows.Forms pruža klase za rad sa tekstualnim poljima. Pored TextBox kontrole postoji i RichTextBox kontrola koja omogućava dodatne opcije za formatiranje.

Primjer: Dizajnirati formu koja će omogućiti unos kriptovanog (*password*) tekst polja. Omogućiti da se sadržaj polja ispiše na formu nakon klika na dugme koje se također nalazi na formi.

Opis rješenja:

Forma treba imati tekst polje –PasswordChar koje se koristi za unos teksta, label polje za prikaz unesenog teksta i dugme sa programiranim događajem na klik, koji će inicirati prikazivanje teksta. Ulazni fokus na formi treba biti na tekstualnom polju. Koraci implementacije rješenja su:

1. Kreiran je projekat Windows tipa sa nazivom PrikazPasswordPolja.
2. Na formu je sa Toolboxa dodana kontrola TextBox. Naziv kontrole je promijenjen u ulazPasswordText. Osobina PasswordChar je postavljena na *. Sa TabIndex osobinom postavlja se fokus na ovu kontrolu.
3. Ispred teksta kontrole dodana je labela, Text osobina labele je promijenjena u Password.
4. Na formu je dodana kontrola Button. Naziv kontrole je promijenjen u prikazButton, a Text osobina je promijenjena u Prikaz passworda.
5. Na formu je dodana kontrola Label. Naziv labele je promijenjen u prikazPasswordLabel, Text osobina kontrole nije postavljena (izbrisana je *default* vrijednost). Slika 6.1. prikazuje formu i dodane kontrole u dizajner modu.



Slika 6.1: Forma sa kontrolama u dizajn modu

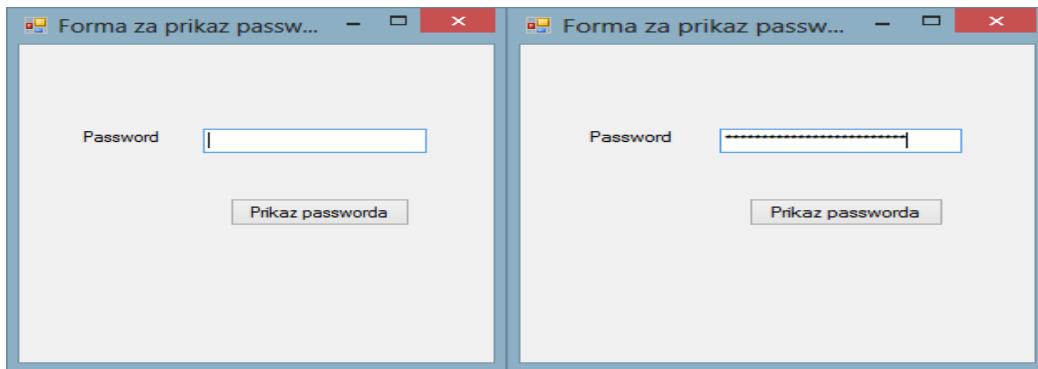
RAZVOJ PROGRAMSKIH RJEŠENJA

6. Programiran je događaj koji se aktivira kada korisnik pritisne na dugme Prikaz passworda. Metoda za upravljanje događajem koji se desi klikom na to dugme ispisuje na labelu uneseni tekst-*password* (kod 6.1).

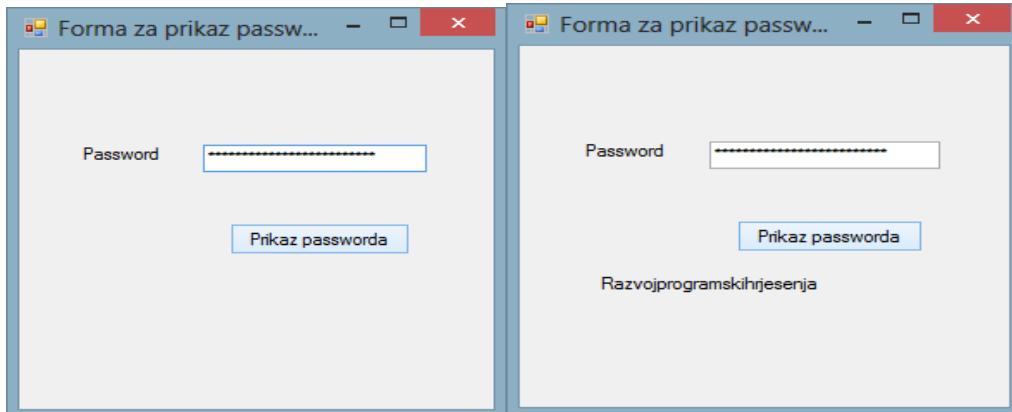
```
private void prikazButton_Click(object sender, EventArgs e)
{
    prikazPasswordLabel.Text = ulazPasswordText.Text;
}
```

Kod 6.1: Metoda za upravljanje Click događajem dugmeta za prikaz passworda

Nakon izvršavanja ovog projekta mogući scenarij odnosno koraci izvršavanja su:



1.Fokus na formi je tekst polje 2.Korisnik unosi *password* u tekst polje



3.Korisnik klikne na dugme Prikaz passworda 4. *Password* se prikazuje na labeli

RAZVOJ PROGRAMSKIH RJEŠENJA

6.1.3. CheckBox i RadioButton kontrole

Pored osnovnog tipa dugmeta postoje još dva tipa dugmadi stanja: CheckBox i RadioButton. Isto kao i klasa Button, klase za CheckBox i RadioButton su izvedene iz klase ButtonBase.

CheckBox

CheckBox je najčešće u obliku malog kvadrata iza kojeg je opisni tekst. Najčešće se formiraju grupe checkbox dugmadi i koriste se za izbor jedne ili više opcija. Ispod je prikazana forma sa tri checkboxa sa inicijalnim stanjima-oznakama: UnChecked, Checked i Indeterminate. Stanja se mijenjaju klikom na kontrolu i pri tome se aktivira događaj CheckedChanged.



Lista često korištenih CheckBox osobina i događaja je u tabeli 6.3.

CheckBox osobine i događaji	Opis
Osobine (properties)	
Checked	Indicira da li je CheckBox selektiran (postoji check mark) ili nije (prazan je). Osobina vraća Boolean vrijednost.
CheckState	Indicira stanje <i>check boxa</i> , koje je iz liste vrijednosti CheckState (Checked, Unchecked ili Indeterminate). Indeterminate se koristi kada je neodređeno stanje.
Text	Specificira tekst prikazan desno od CheckBoxa.
Događaji	
CheckedChanged	Generira se kada se Checked osobina mijenja. Ovo je <i>default</i> događaj za CheckBox kontrolu.
CheckStateChanged	Generira se kada se CheckState osobina mijenja.

Tabela 6.3: Osobine i događaji CheckBox kontrole

RAZVOJ PROGRAMSKIH RJEŠENJA

Primjer: Na dizajniranoj formi za unos i prikaz password polja potrebno je da metoda za upravljanje klik događajem dugmeta Prikazi Password postavi na formu *check box* Povecaj. Klikom na *check box* povećava se tekst na labeli koja prikazuje *password*.

Opis rješenja: Potrebno je izmijeniti metodu za `prikazButton_Click`. Dodaje se kod 6.2 koji dodaje *check box* kontrolu.

```
private void prikazButton_Click(object sender, EventArgs e)
{
    prikazPasswordLabel.Text = ulazPasswordText.Text;

    // Kreira inicijalni check box (checkBox1).
    CheckBox checkBox1 = new CheckBox();

    // Postavljanje oblika check boxa
    checkBox1.Appearance = Appearance.Normal;

    // Postavljanje lokacije za check box kontrolu
    checkBox1.Location = new System.Drawing.Point(186, 186);

    checkBox1.Text="Povecaj" ;

    // Omogućava da se može vršiti klik na check box
    checkBox1.AutoCheck = true;

    // Povezivanje check box kontrole sa događajem
    // događaj se dešava kada se desi promjena stanja check boxa
    checkBox1.CheckedChanged += new
System.EventHandler(this.checkBox1_CheckedChanged);

    // Dodaje kontrolu na formu
    Controls.Add(checkBox1);

}
```

Kod 6.2: Dodavanje *check boxa* (`checkBox1`) i povezivanje sa događajem `CheckedChangedState`

Kod 6.3 prikazuje metodu za upravljanje događajem koji se desi nakon promjene stanja *check boxa*.

RAZVOJ PROGRAMSKIH RJEŠENJA

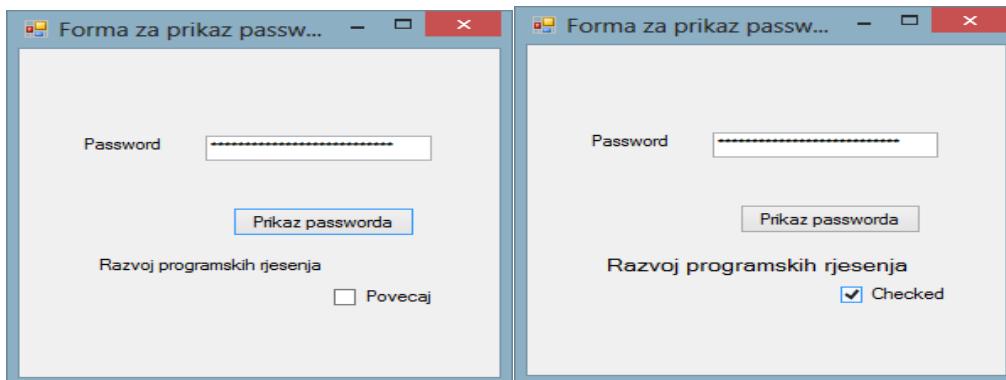
```
private void checkBox1_CheckedChanged( object sender, EventArgs e)
{
    float currentSize;

// konvertovanje (cast) sender objekta u objekt klase CheckBox
    CheckBox checkbox = sender as CheckBox;

// kada se selektira (check) check box povećaj font na labeli passworda
    if (checkbox.Checked)
    {
        checkbox.Text = "Checked";
        currentSize = prikazPasswordLabel.Font.Size;
        currentSize += 2.0F;
        prikazPasswordLabel.Font = new Font(prikazPasswordLabel.Font.Name,
currentSize, prikazPasswordLabel.Font.Style);
    }
    else
        checkbox.Text = "UnChecked";
}
```

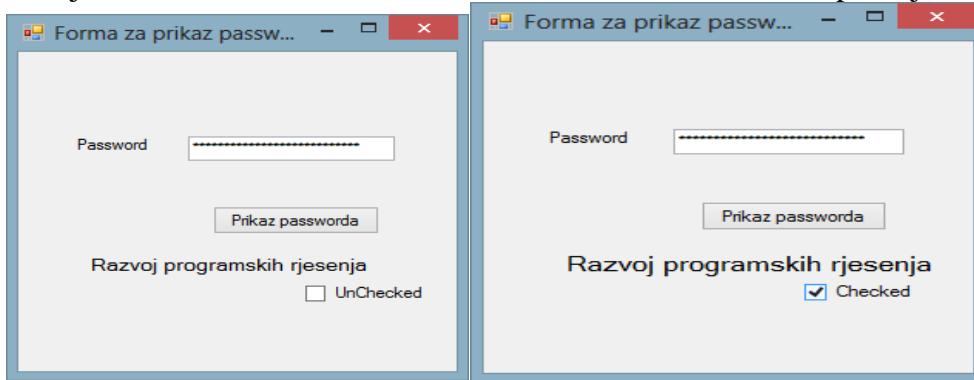
Kod 6.3: Metoda za upravljanje događajem CheckedChangedState – mijenja veličinu teksta ako je *check box* selektiran (Checked)

Mogući scenarij izvršavanja koda 6.3:



1. Unos *passworda*, klik na dugme.
Prikazuje se uneseni tekst i *check box*.

2. Klik (selekt) na *check box*. Tekst se poveća,
labela uz *check box* se promijenila.



3. Klik (deselekt) *check box*, mijenja
se labela uz *check box* i veličina teksta.

4. Isto kao pod 2.

RAZVOJ PROGRAMSKIH RJEŠENJA

RadioButton

Radio dugmad (definirani sa klasom `RadioButton`) su slični sa `CheckBox` kontrolom, jer također imaju dva stanja (on-selektiran ili off-nije selektiran). Radio dugmad se pojavljuju kao grupa, u kojoj samo jedno radio dugme iz te grupe može biti istovremeno selektirano. Selektiranjem jednog radio dugmeta drugi označeni u grupi se deseletiraju. Zbog toga, radio dugmad se koriste da se prikaže skup više međusobno isključivih opcija. Na slici ispod je dat primjer ove kontrole.



Radio dugmad (instance `RadioButton` klase) u okviru `GroupBox` kontrole

Da bi radio dugmad formirala logički povezanu grupu, pojedinačna dugmad moraju biti dodana u okviru `GroupBox` ili `Panel` kontrole (objašnjeno u nastavku). Često korištene osobine i događaji klase `RadioButton` su u tabeli 6.4 (ispod).

RadioButton	osobine i dogadaji	Opis
Osobine		
<i>Checked</i>		Indicira da li <code>RadioButton</code> selektiran.
<i>Text</i>		Specificira tekst <code>RadioButton</code> kontrole.
Dogadaji		
<i>CheckedChanged</i>		Generira se svaki put kada <code>RadioButton</code> promijeni stanje.

Tabela 6.4: Osobine i događaji `RadioButton` kontrole

RAZVOJ PROGRAMSKIH RJEŠENJA

Upravljanje sa događajem koji se aktivira promjenom stanja radio dugmadi je slično kao kod *check box* kontrole. Primjer je dat u prilogu 2.

Radio dugmad i *check box* kontrole nude jednostavni pristup opcijama i kao takvi izuzetno su prihvaćeni od strane korisnika.

Neki nedostaci ovih kontrola je što nisu podesne za veliki broj izbora, pri tome je teže odabrati opciju i koriste dosta prostora na ekranu.

6.1.4. Formiranje grupa kontrola – `GroupBox` i `Panel`

`GroupBox` i `Panel` se koristi za aranžiranje kontrola na formi. `GroupBox` i `Panel` se tipično koristi za grupiranje više kontrola slične funkcionalnosti. Sve kontrole u okviru kontrola `GroupBox` ili `Panel` se pomjeraju zajedno kada se `GroupBox` ili `Panel` pomjera.

Primarna razlika između ove dvije kontrole je da `GroupBox` može prikazivati naslov (tekst) i ne uključuje pomične trake, dok `Panel` kontrola može uključiti pomične trake i ne uključuje naslov, a njen okvir se može mijenjati i sa `BorderStyle` osobinom. U tabelama 6.5 i 6.6 su prikazane neke od osobina `GroupBox` i `Panel` kontrola.

GroupBox osobine	Opis
Controls	Skup kontrola koje <code>GroupBox</code> sadrži.
Text	Specificira naslov koji se prikazuje na vrhu <code>GroupBox</code> kontrole.

Tabela 6.5: `GroupBox` osobine

Panel osobine	Opis
AutoScroll	Indicira da li se pokretne trake pojavljuju kada je <code>Panel</code> malen za prikazivanje svih kontrola. Podrazumijevana vrijednost je <code>false</code> .
BorderStyle	Postavlja okvir <code>Panel</code> kontrole. Podrazumijevana vrijednost je <code>None</code> ; druge opcije su <code>Fixed3D</code> i <code>FixedSingle</code> .
Controls	Skup kontrola koje <code>Panel</code> sadrži.

Tabela 6.6: `Panel` osobine

6.2. Interakcija sa korisnikom

Program treba obrađivati na specifičan način pojedine pritiske tipki na tastaturi ili klikove mišem. U tu svrhu se pišu metode za upravljanje događajima koje odgovaraju na te akcije. Također program treba ostvariti i interakciju putem posebno prilagođenih poruka korisniku. To se ostvaruje sa prikazivanjem poruka (npr. metodom `MessageBox.Show()`) i kreiranjem posebno prilagođenih okvira za dijalog. U ovoj sekciji se obrađuju pomenuti aspekti.

6.2.1. MessageBox.Show() metoda

.NET Framework obezbeđuje klasu `MessageBox`, u okviru imenovanog prostora `System.Windows.Forms` koja pruža statičku metodu `Show` da prikaže prozor poruke (*message box*). Metoda (`MessageBox.Show()`) se može koristiti za prikaz teksta, prikaz dodatnih ikona uz tekst, kao i za prikaz dugmadi putem kojih se može ostvariti dodatna komunikacija između programskog rješenja i korisnika. Naslov, poruka, dugmad i ikone koji se prikazuju u prozoru poruke se određuju sa parametrima koji se predaju ovoj metodi.

`MessageBox.Show()` metoda ima više parametara, sa prva 4 parametra mogu se dobiti razne varijante prikaza. Prvi parametar je tekst poruke, drugi naziv prozora, treći parametar je namijenjen za prikaz dugmadi uz poruku, a zadnji (četvrti) parametar je namijenjen za prikaz ikone uz tekst poruke. Ukoliko `MessageBox` metoda sadrži dugmad često se naziva i *dijalog box*.

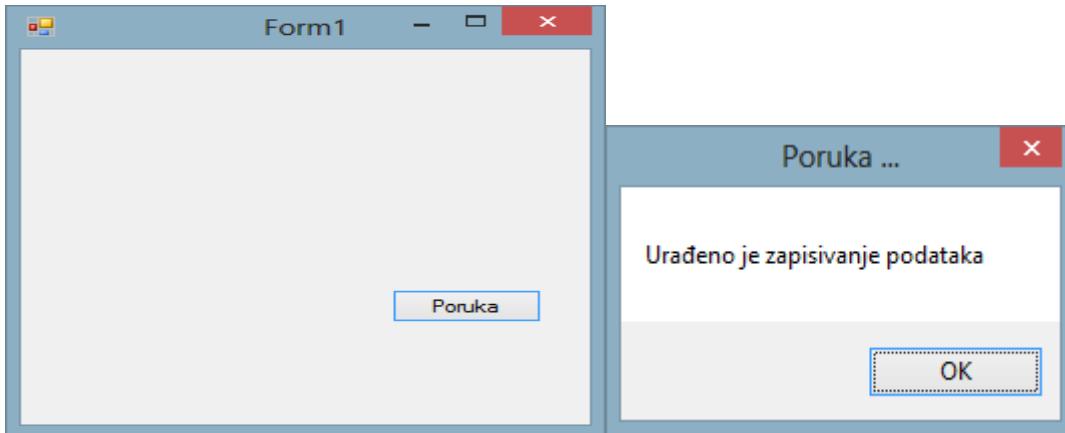
Npr. da bi se prikazao:

- *message box* (prozor poruke) sa nazivom prozora i tekstrom poruke koristi se metoda `MessageBox.Show` i parametri:

```
MessageBox.Show(tekstporuke, naziv)
```

Za scenarij prozora i poruka prikazanih ispod korišten je kod 6.4. U kodu je metoda za upravljanje klik događajem, unutar koje je korištena `MessageBox.Show` metoda sa dva parametra.

RAZVOJ PROGRAMSKIH RJEŠENJA



Poruka u okviru prozora sa nazivom Poruka... dobijena je nakon izvršenog događaja klika na dugme Poruka.

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Urađeno je zapisivanje podataka", "Poruka ...");
}
```

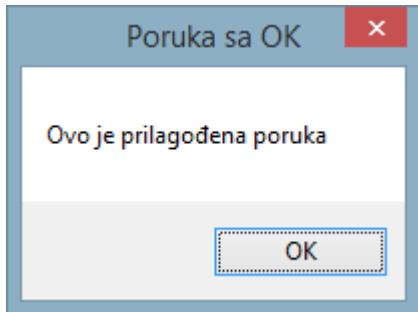
Kod 6.4: Metoda `MessageBox.Show` sa parametrima za prikaz poruke i naziva prozora

- *message box* (prozor poruke) s nazivom prozora, tekstrom poruke, dodatnim dugmadima koristi se metoda `MessageBox.Show` sa parametrima:

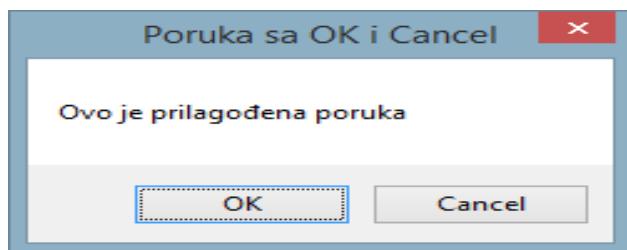
`MessageBox.Show(tekstporuke, naziv,dugmad)`

Dugmad su određena sa osobinom `MessageBoxButtons` klase `MessageBox` uz koju se veže lista vrijednosti: `AbortRetryIgnore`, `OK`, `OKCancel`, `YesNoCancel`, `YesNo`, `RetryCancel`. Ove vrijednosti određuju dugmad koja se mogu pojaviti uz poruku. Dodatnim parametrom `MessageBox.Show` metode može se postaviti koje je dugme u fokusu prilikom pojavljivanja poruke. Ispod su prikazani prozori poruka i odgovarajuća `MessageBox.Show` metoda.

RAZVOJ PROGRAMSKIH RJEŠENJA



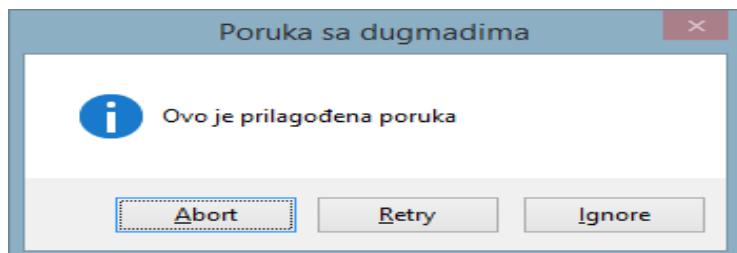
```
MessageBox.Show("Ovo je prilagođena poruka", "Poruka sa OK", MessageBoxButtons.OK);
```



```
MessageBox.Show("Ovo je prilagođena poruka", "Poruka sa OK i Cancel", MessageBoxButtons.OKCancel);
```

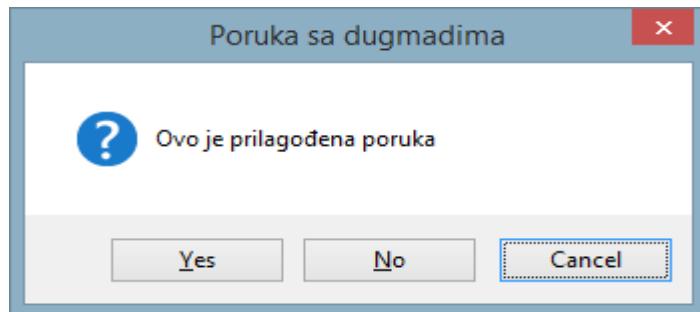
- *message box* (prozor poruke) sa nazivom, dugmadima i dodatnim ikonama koristi se `MessageBox.Show` metoda sa parametrima:
`MessageBox.Show(tekstporuke, naziv, dugmad, ikona)`

Ikone su određene osobinom `MessageBoxIcon` klase `MessageBox` uz koju se veže lista vrijednosti: Asterik, Error, Information, Question, Stop, Warning, Ikona se bira u ovisnosti od tipa poruke i njene važnosti za korisnika. Ispod su prikazani prozori poruka i odgovarajuća `MessageBox.Show` metoda sa parametrom i za ikone.

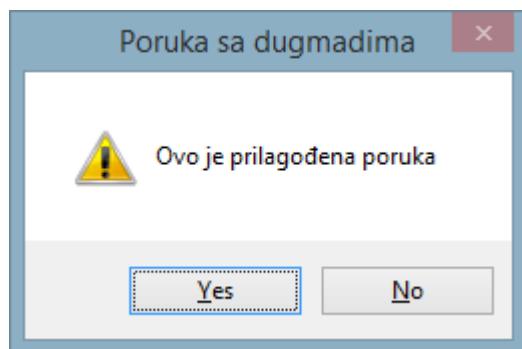


```
MessageBox.Show("Ovo je prilagođena poruka", "Poruka sa dugmadima", MessageBoxButtons.AbortRetryIgnore, MessageBoxIcon.Information);
```

RAZVOJ PROGRAMSKIH RJEŠENJA



```
MessageBox.Show("Ovo je prilagođena poruka", "Poruka sa dugmadima",
    MessageBoxButtons.YesNoCancel, MessageBoxIcon.Question,MessageBoxDefaultButton.Button3);
```



```
MessageBox.Show("Ovo je prilagođena poruka", "Poruka sa dugmadima",
    MessageBoxButtons.YesNo, MessageBoxIcon.Warning);
```

Prilikom dizajniranja prozora poruka preporučuje se da se:

- opštim porukama dodijeli ikona `Information`,
- osnovno (fokus) dugme postaviti da ima tamnu ivicu,
- ukoliko korisnik nema kontrolu nad onim što se desilo ne treba omogućiti dugme `Cancel`.

Veoma je značajno (i nije tako jednostavno) obezbjediti odgovarajuće poruke korisnicima u odgovarajućim trenucima. Osim razmatranja ikona i dugmadi koja će biti prikazana u poruci, treba pratiti sljedeće preporuke:

- koristiti formalan ton,
- izbjegavati upotrebu skraćenica,
- tekst treba biti razumljiv,
- ne pisati previše dugačke poruke,
- poruke sa pitanjima postaviti što jednostavnije,
- izbjegavati stroga tehničke izraze.

RAZVOJ PROGRAMSKIH RJEŠENJA

6.2.2. Interakcija korisnika sa tastaturom i mišem

Interakcija sa korisnikom može se ostvariti sa tastaturom i mišem pomoću brojnih događaja.

Upravljanje događajima tastature (Keyboard-Event Handling)

Događaji upravljanja tipkama (*key events*) se dešavaju kada se tipke tastature pritisnu ili opuste. Ovi događaji se mogu vezati za svaku kontrolu koja je naslijedena iz `System.Windows.Forms.Control`. Postoje tu tri događaja `KeyPress`, `KeyUp` i `KeyDown`. `KeyPress` događaj se dešava kada korisnik pritisne tipku koja predstavlja ASCII karakter. Specifična tipka (*key*) se može odrediti sa osobinom `KeyChar` koja je argument nekog događaja.

`KeyPress` događaj ne indicira da li su modifikatorske tipke (npr. `Shift`, `Alt` i `Ctrl`) bile pritisnute kada se događaj desio. Ako je ta informacija važna, `KeyUp` ili `KeyDown` događaji se mogu koristiti. `KeyEventArgs` argument za svaki od ovih događaja sadrži informacije o modifikatorskim tipkama. Tabela 6.7 daje važne informacije o događajima vezanim za interakciju sa tastaturom. Mnoge osobine vraćaju vrijednosti iz `Keys` liste, koja obezbeđuje konstante koji specificiraju različite tipke na tastaturi.

Događaji tastature i argumenti događaja	
Događaji tastature sa argumentima događaja tipa <code>KeyEventArgs</code> .	
<code>KeyDown</code>	Generira se kada je tipka inicijalno pritisnuta.
<code>KeyUp</code>	Generira se kada je tipka opuštena.
Key Event sa argumentom događaja tipa <code>KeyPressEventArgs</code> .	
<code>KeyPress</code>	Generira se kada je tipka pritisnuta.
Klasa <code>KeyPressEventArgs</code> – osobine	
<code>KeyPressEventArgs</code> klasa je klasa argumenata povezana sa <code>KeyPress</code> događajem. Ova klasa predstavlja karaktere tastature pritisnute od strane korisnika. Dio je <code>System.Windows.Forms</code> prostora, naslijedena je od <code>System.EventArgs</code> klase.	
<code>KeyChar</code>	Vraća ASCII karakter tipke koja je pritisnuta.
<code>Handled</code>	Indicira da li se <code>KeyPress</code> događaj obrađuje.
<code>KeyEventArgs</code> klasa je klasa argumenata događaja povezanih sa <code>KeyDown</code> i <code>KeyUp</code> događajima i tipkama tastature koje su pritisnute ili opuštene od strane korisnika. Dio je <code>System.Windows.Forms</code> prostora, i naslijedena je od <code>System.EventArgs</code> klase.	

RAZVOJ PROGRAMSKIH RJEŠENJA

Dogadaji tastature i argumenti dogadaja

Alt	Indicira da li je Alt tipka pritisnuta.
Control	Indicira da li je Ctrl tipka pritisnuta.
Shift	Indicira da li je Shift tipka pritisnuta.
Handled	Indicira da li je dogadjaj obrađen.
KeyCode	Vraća kod tipke kao vrijednost iz Keys liste vrijednosti. Ne uključuje informacije o modifikatorskim tipkama. Koristi se za testiranje specifičnih tipki.
KeyData	Vraća kod za kombinaciju pritisnutih tipki uključujući i modifikatorske tipke. Ova osobina sadrži sve informacije o pritisnutoj tipki.
KeyValue	Vraća kod tipke kao int vrijednost.
Modifiers	Vraća Keys vrijednost indicirajući svaku pritisnuto modifikatorsku tipku (Alt, Ctrl i Shift).

Tabela 6.7: Dogadaji tastature i njihovi argumenti

Kod 6.5 prikazuje metodu koja upravlja dogadjajem KeyUp povezanim sa tekstualnim poljem (textBox2) na formi.

```
private void textBox2_KeyUp(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.F1)
        MessageBox.Show("pritisnuta F1 tipka");
    if (e.KeyCode == Keys.D1)
        MessageBox.Show("pritisnut broj 1");
    if ((e.Modifiers == Keys.Shift) && (e.KeyCode == Keys.F2))
        MessageBox.Show("pritisnuta Shift + F2 tipka");
}
```

Kod 6.5: Metoda za upravljanje dogadjajem KeyUp otkriva koja je tipka pritisnuta

RAZVOJ PROGRAMSKIH RJEŠENJA

Upravljanje događajima miša

Ova sekcija objašnjava kako se upravlja događajima uzrokovanim mišem prilikom klika mišem, pritiska nekog dugmeta na mišu i prilikom njegovog pomjeranja. Mišem prouzrokovani događaji se mogu vezati za svaku kontrolu koja je izvedena iz klase `System.Windows.Forms.Control`. Za mnoge događaje uzrokovane mišem, informacije o događajima se predaju metodi za upravljanje događajem preko objekta klase `MouseEventArgs`, a delegira se `MouseEventHandler` za upravljanje događajem.

Klasa `MouseEventArgs` sadrži informacije vezane za događaj prouzrokovan mišem, kao što su koordinate (x,y) pointera miša, koje je dugme pritisnuto- desno, lijevo ili srednje (`Right`, `Left` ili `Middle`) i broj koliko je puta izvršen klik. U tabeli 6.8 su prikazane neki uobičajni događaji mišem i njihovi argumenti.

Događaji prouzrokovani mišem i njihovi argumenti	
Događaji mišem sa argumentom tipa <code>EventArgs</code>	
<code>MouseEnter</code>	Dešava se kada miš ‘uđe’ na granice kontrole.
<code>MouseLeave</code>	Dešava se kada miš napusti granice kontrole.
Događaji mišem sa argumentima tipa <code>MouseEventArgs</code>	
<code>MouseDown</code>	Dešava se kada se pritisne tipka miša dok je cursor miša unutar granica kontrole.
<code>MouseMove</code>	Dešava se kada se miš pomjera unutar granica kontrole.
<code>MouseUp</code>	Dešava se kada se opusti tipka miša kada je cursor na granicama kontrole.
Klasa <code>MouseEventArgs</code> osobine	
<code>Button</code>	Specificira koje je dugme miša pritisnuto (<code>Left</code> , <code>Right</code> , <code>Middle</code> ili <code>none</code>).
<code>Clicks</code>	Broj puta koliko je pritisnuta tipka miša.
<code>X</code>	x-koordinata unutar kontrole kada se događaj desi.
<code>Y</code>	y-koordinata unutar kontrole kada se događaj desi.

Tabela 6.8: Događaji mišem i njihovi argumenti

RAZVOJ PROGRAMSKIH RJEŠENJA

Rezime:

U okviru ovog poglavlja nastavljeno je sa pregledom GUI kontrola, osobina i događaja vezanih uz kontrole. Objasnjene su kontrole za prikaz teksta (Label), unos teksta (TextBox), izbor opcija (RadioButton, CheckBox), kontrole za logičko povezivanje kontrola u grupe (GroupBox, Panel). Već je uveden dovoljan broj kontrola za kreiranje formi. U sljedećem poglavlju će se prikazati još jedan dio kontrola. U ovom poglavlju je objasnjen i način kreiranja prozora poruka i način upravljanja događajima uzrokovanim sa tipkama tastature i mišem.

Pitanja i zadaci za samostalni rad:

1. Za svaku uvedenu kontrolu analizirajte sve dostupne osobine i događaje. Uradite više jednostavnih primjera koji sadrže upotrebu osobina i događaja kontrole.
2. Kako se može ostvariti funkcionalna interakcija sa korisnikom?
3. Koja je osnovna namjena `MessageBox.Show()` metode? Navedite njene parametre.
4. Koji su osnovni savjeti vezani za poruke namijenjene korisniku?
5. Kako se postiže interakcija sa tastaturom, koji su vezani događaji?
6. Koji su događaji koji se koriste za manipulaciju ulazom miša?
7. Opišite labelu kao kontrolu - namjena, osobine.
8. Opišite tekst polje kao kontrolu – namjena, osobine, osnovni podrazumijevani događaj.
9. Opišite *check box* kao kontrolu – namjena, osobine, osnovni podrazumijevani događaj.
10. Opišite *radio dugme* kao kontrolu – namjena, osobine, osnovni podrazumijevani događaj.
11. Opišite namjenu panel i *group box* kontrole?
12. Za dugme kontrolu – napisati događaj prouzrokovani klikom na njega, koji će samo napisati informativnu poruku, da se događaj desio, korištenjem `MessageBox` klase i njene odgovarajuće metode.
13. Kreirati formu sa dugmetom i pri prelasku mišem preko dugmeta obojiti pozadinu u plavo.
14. Na kreiranu formu u zadatku 13 dodati novo tekst polje na formu. Omogućiti korisniku unos teksta u dodanu kontrolu. Implementirati ispis tog teksta preko prozora poruke nakon što se pritisne dugme postavljeno na formi.
15. Na kreiranu formu u zadatku 13 dodati labelu i napisati metodu koja će tekst iz tekst polja ispisati na toj labeli nakon unosa ili promjene teksta.

Poglavlje 7: Pregled dodatnih GUI kontrola

U okviru ovog poglavlja dat je pregled GUI kontrola koje omogućavaju prikaz slika (**PictureBox** kontrola), prikaz i izbor opcija (**ListBox**, **ComboBox** kontrola), izbor linkova (**LinkLabel** kontrola), izbor datuma (**MonthCalendar** kontrola), prikaz i izbor tabova (**TabControl** kontrola), izbor ranga vrijednosti (**NumericUpDown** kontrola) kroz tematske jedinice koje odgovaraju nazivima kontrola.

7.1. Kontrola za prikazivanje slike **PictureBox**

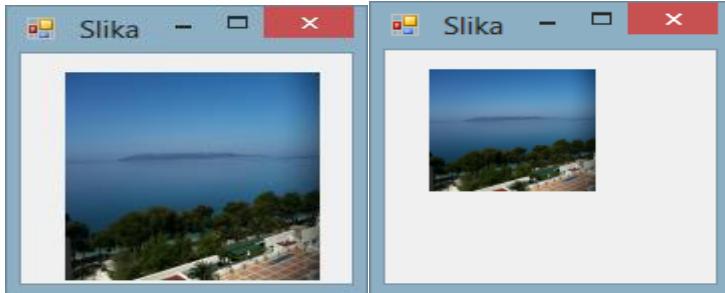
PictureBox kontrola služi za prikazivanje slika u određenom formatu (bitmap, GIF (Graphics Interchange Format) i JPEG (Joint Photographic Experts Group),...). Tabela 7.1 prikazuje važnije osobine i događaje **PictureBox** kontrole.

PictureBox	
Osobine i događaji	Opis
Osobine	
Image	Specificiranje slike koja se prikazuje u PictureBox kontroli.
SizeMode	Nabrojiva lista koja kontrolira veličinu i poziciju slike. Vrijednosti su Normal (default), StretchImage , AutoSize i CenterImage . Normal postavlja sliku u lijevi ugao PictureBox kontrole, CenterImage postavlja sliku u sredinu. Ove dvije opcije ‘otkidaju’ sliku ako je previše velika. StretchImage mijenja veličinu slike da bi se smjestila u PictureBox , dok AutoSize kontrola mijenja veličinu PictureBox kontrole da bi se smjestila slika.
Događaj	
Click	Desi se kada se klikne na kontrolu.

Tabela 7.1: Često korištene osobine i događaji **PictureBox** kontrole

RAZVOJ PROGRAMSKIH RJEŠENJA

Primjer: Na slikama ispod prikazana je faza izvršavanja aplikacije koja sadrži formu sa kontrolom za prikaz slike. Prilikom dvostrukog klika na sliku, slika se smanji.



Opis rješenja:

Korištenjem dizajnera (može i upotrebom odgovarajućih klasa) kreirana je kontrola `PictureBox` (`gradPictureBox`). Opcija `SizeMode` postavljena je na `StretchMode`.

Izvršeno je dodjeljivanje i programiranje metode za upravljanje događajem dvostrukog klika `gradPictureBox` kontrole (kod 7.1). Unutar metode mogu se primijetiti i dodatne osobine `PictureBox` kontrole koje se tiču veličine, visine i širine same kontrole (`Size`, `Height`, `Width`).

```
private void gradPictureBox_DoubleClick(object sender, EventArgs e)
{
    gradPictureBox.Size = new Size(gradPictureBox.Size.Height - 50,
gradPictureBox.Size.Width-50);
}
```

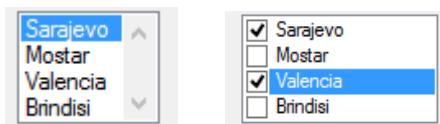
Kod 7.1: Metoda za upravljanje događajem dvostrukog klika na sliku

7.2. Izbor opcija: *List* i *Combo box* kontrole

List box i *combo box* su osnovne kontrole Windows aplikacija koje služe za prikaz liste opcija/izbora. Osim prikaza ove kontrole omogućavaju izbor jedne ili više stavki u ponuđenoj listi opcija. Podržane su sa `ComboBox` i `ListBox` klasama izvedenim iz `ListControl` klase. Slijedi detaljniji opis ovih kontrola.

List box kontrola

List box kontrola omogućava prikaz više opcija/izbora i izbor jedne ili više opcije od ponuđenih u listi. Osobine ove kontrole date su detaljnije u tabeli 7.2. Postoji i posebna varijanta ove kontrole a to je `CheckedListBox` kontrola koja pored svake opcije ima i *check box*.



List box i *Checked list box* kontrola

Opcije list kontrola mogu se popuniti u okviru dizajnera korištenjem `Items` opcije i vezanog `Collection` editora. Često je potrebno da se iz programskog koda popuni lista opcija. Kod 7.2 je dio koda kojim se popunjavaju stavke `ListBox` kontrole.

```
private void Form1_Load(object sender, EventArgs e)
{
    List<String> gradovi = new List<String>() { "Sarajevo", "Mostar",
"Valencia", "Brindisi" };
    sifarnikListBox.DataSource = gradovi;
}
```

Kod 7.2: Popunjavanje liste sa opcijama/izborima

Prednosti *list* i *combo box* kontrola ogledaju se u mogućnosti prikaza više opcija, liste su vidljive i podsjećaju korisnika na raspoložive opcije.

Ove kontrole mogu biti teške za korištenje ukoliko je lista opcija uređena na neočekivan način, kao i ukoliko se često zahtjeva pomijeranje liste da bi se vidjele ponuđene opcije.

Tabela 7.2 (ispod) prikazuju neke važnije karakteristike `ListBox` klase i kontrole.

RAZVOJ PROGRAMSKIH RJEŠENJA

ListBox osobine, metode i događaji	Opis
Osobine	
Items	Kolekcija stavki (opcije izbora) koje se smještaju u <code>ListBox</code> kontrolu.
MultiColumn	Koristi se za podjelu <code>ListBox</code> kontrole u više kolona. Više kolona eliminira vertikalne pokretne trake.
SelectedIndex	Vraća indeks selektirane stavke. Ako stavka nije selektirana osobina vraća -1. Ako se selektira više stavki, ova osobina vraća indeks samo jedne stavke. Zbog ovog razloga, ako je potrebna selekcija više stavki koristi se osobina <code>SelectedIndices</code> .
SelectedIndices	Vraća kolekciju koja sadrži indekse za sve selektirane stavke.
SelectedItem	Vraća reference na selektirane stavke. Ako je selektirano više stavki, vraća stavku sa najmanjim brojem indeksa.
SelectedItems	Vraća kolekciju selektiranih stavki.
SelectionMode	Određuje broj stavki koje mogu biti selektirane. Vrijednosti <code>None</code> (ne može se selektirati stavka), <code>One</code> (može se selektirati jedna stavka), <code>MultiSimple</code> (može se selektirati više stavki) i <code>MultiExtended</code> (za selekciju mogu se koristiti <code>Shift</code> , <code>Ctrl</code> i strelice).
Sorted	Postavkom ove vrijednosti na <code>true</code> vrši se alfabetsko sortiranje stavki.
Metode	
ClearSelected	Poništavanje odabranih (selektiranih) izbora.
GetSelected	Prima indeks stavke i vraća <code>true</code> ako je ta stavka selektirana.
Događaji	
SelectedIndexChanged	Aktivira se kada korisnik selektira novu stavku u listi.

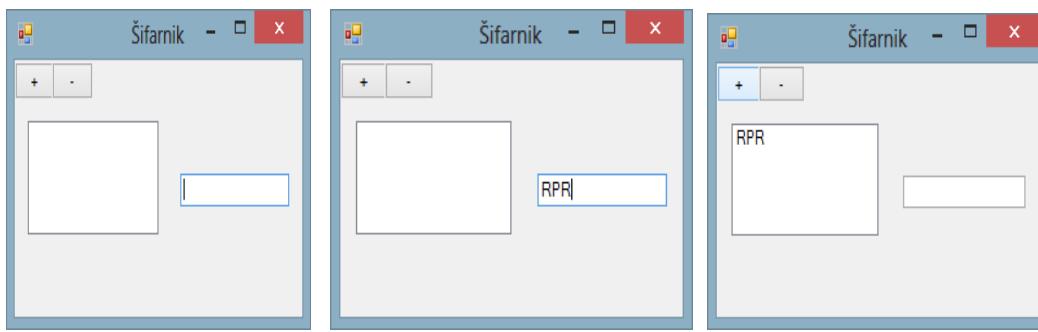
Tabela 7.2: `ListBox` osobine, metode i događaji

RAZVOJ PROGRAMSKIH RJEŠENJA

Primjer: Potrebno je razviti Windows aplikaciju sa dizajniranom formom pomoću koje će se moći dodavati i brisati stavke iz liste.

Opis rješenja: Forma će sadržavati `ListBox` kontrolu, tekstualno polje za unos stavki, i dvije dugme kontrole označene sa + i -, koje će respektivno generirati događaje za unos i brisanje stavke iz liste.

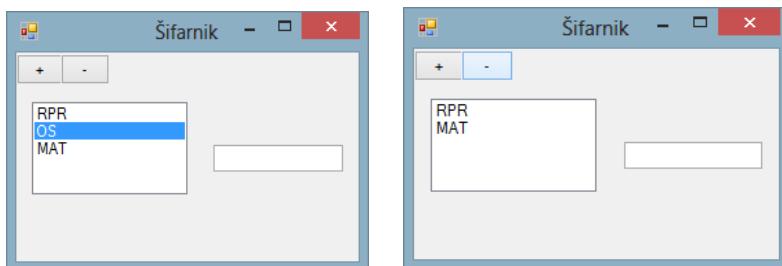
Nakon kreiranja aplikacije i odgovarajućih kontrola jedan od mogućih scenarija izvršavanja same aplikacije je dat na slici ispod.



Početna forma

Unos šifre u tekstualno polje

Pritisnuto dugme +



Izbor opcije za brisanje

Pritisnuti dugme -

Na ovaj način bi korisnik mogao unositi odgovarajuću listu a ista bi se mogla koristiti u okviru programa na formama gdje je potreban izbor na osnovu kreirane liste.

Kod 7.3 i kod 7.4 prikazuju metode koje su programirane da upravljaju pomenutim događajima za klik na dugmad označenu sa + i -.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
private void dodajButton_Click(object sender, EventArgs e)
{
    if (ulazniTextBox.Text.Length > 0)
    {
        sifraListBox.Items.Add(ulazniTextBox.Text);
        ulazniTextBox.Clear();
    }

}
```

Kod 7.3: Metoda za upravljanje događajem klik dugmeta + unosi nove stavke u listu

```
private void brisiButton_Click(object sender, EventArgs e)
{
    if (sifraListBox.SelectedIndex != -1)
        sifraListBox.Items.RemoveAt(sifraListBox.SelectedIndex);
}
```

Kod 7.4: Metoda za upravljanje događajem klik dugmeta – briše stavke iz liste

Primjer: Potrebno je kreirati listu boja. Omogućiti izbor jedne boje iz `ListBoxa` i promijeniti pozadinsku boju forme u skladu sa izborom.

U ovom zadatku boja se opisuje pomoću klase `Boja`. Na osnovu instanci klase `Boja` se formira lista vrijednosti. Kod 7.5 daje jedno od mogućih rješenja za traženi zadatak.

```
public class Boja
{
    public String Naziv { get; set; }
    public Color SistemskaBoja { get; set; }

    public Boja(String Naziv, Color SistemskaBoja)
    {
        this.Naziv = Naziv;
        this.SistemskaBoja = SistemskaBoja;
    }

    public override string ToString()
    {
        return Naziv;
    }
}

// nastavak na sljedećoj stranici
```

RAZVOJ PROGRAMSKIH RJEŠENJA

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();

//Kreiranje liste vrijednosti bojaListBox i njeno popunjavanje sa opcijama Crvena, Zelena)

        List<Boja> boje = new List<Boja>()
        {
            new Boja("Crvena",Color.Red),
            new Boja("Zelena",Color.Green)
        };
        bojaListBox.DataSource = boje;
        bojaListBox.SelectedIndexChanged += bojaListBox_SelectedIndexChanged;

    }

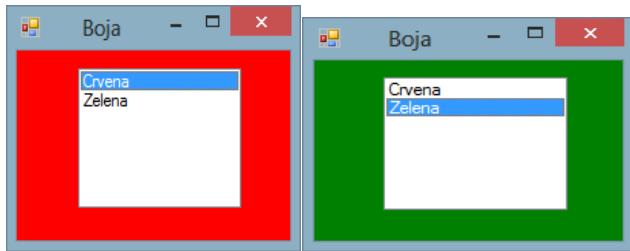
//Metoda za upravljanje događajem koji se desi uslijed promjene odabira boje iz liste
    private void bojaListBox_SelectedIndexChanged(object sender, EventArgs e)
    {

        if (bojaListBox.SelectedIndex != -1)
            this.BackColor = (bojaListBox.SelectedItem as Boja).SistemskaBoja;

    }
}
```

Kod 7.5: ListBox kontrola-kreiranje kontrole i metoda za upravljanje događajem SelectedIndexChanged

Scenarij izvršavanja gornjeg programskog rješenja:



Odabir boje iz liste uzrokuje promjenu pozadinske boje

RAZVOJ PROGRAMSKIH RJEŠENJA

ComboBox kontrola

ComboBox kontrola predstavlja kombinaciju kontrole za unos teksta i upravo opisanog ListBoxa. Otuda i naziv "combo" (više u jednom). Pri tome ComboBox dolazi u tri različite varijante: u prvoj varijanti se radi o TextBoxu ispod kojeg se nalazi ListBox. Ova varijanta omogućava ili da se unese proizvoljan sadržaj ili da se kao sadržaj odabere nešto ponuđeno kroz ListBox. Druga varijanta predstavlja jednostavnu modifikaciju prve gdje ListBox nije stalno prisutan ispod TextBoxa već se prikazuje samo na zahtijev. Treća varijanta ComboBoxa ne dozvoljava unos proizvoljnog sadržaja preko tekstu kontrole, već omogućava samo odabir jedne od ponuđenih opcija.

ComboBox klasa ima osnovni skup metoda i karakteristika isti kao ListBox kontrola. Neke karakteristične osobine za ovu kontrolu date su u tabeli 7.3.

ComboBox osobine i događaji	Opis
Osobine	
DropDownStyle	Određuje tip ComboBox kontrole –Simple, DropDown , DropDownList. <ul style="list-style-type: none">ComboBoxStyle.DropDown, tekst polje se može mijenjati, korisnik mora kliknuti na strelicu da prikaže listu. Ovo je unaprijed definiran stil.ComboBoxStyle.DropDownList, isto kao DropDown, sa izuzetkom da tekst polje se ne može mijenjati.ComboBoxStyle.Simple, tekst polje se može mijenjati i lista je uvijek vidljiva.
Items	Kolekcija stavki (izbora) vezanih za ComboBox kontrolu.
MaxDropDownItems	Specificira maksimalan broj stavki (između 1 i 100) koje drop-down lista može prikazati. Ako je broj stavki veći od maksimalnog broja stavki koje se mogu prikazati, pojavljuju se pokretne trake.
Događaji	
SelectedIndexChanged	Aktivira se kad god korisnik odabere novu stavku u drop-down listi.

Tabela 7.3: ComboBox osobine i događaji

RAZVOJ PROGRAMSKIH RJEŠENJA

Jedna od mogućih primjena ComboBoxa je za kreiranje polja za pretragu koja imaju *autosuggest* funkcionalnost (u stilu Google tražilice koja odmah tokom unosa pojma pretrage nudi opcije koje zadovoljavaju do tada unesen tekst).

Primjer: Kreirati formu koja predstavlja jednostavnu tražilicu po osnovu imena i prezimena osoba koje su ranije poznate u sistemu uz omogućenu automatsku *suggest* funkcionalnost.

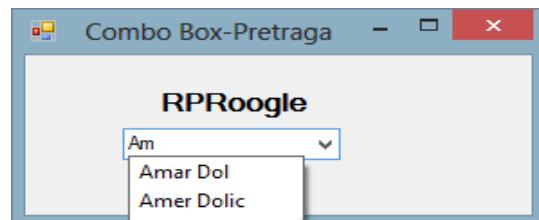
Opis rješenja: Nakon kreiranja Windows Form aplikacije i dodavanja ComboBox kontrole na početnu formu preko Properties editora se mogu promijeniti osobine vezane za automatsku sugestiju (AutoCompleteMode postaviti na Suggest).

Slijedi programskim putem popunjavanje opcija za ComboBox kontrolu.

```
public Form1()
{
    InitializeComponent();
    List<String> poznateOsobe = new List<String>() { "Amer Dolic", "Amir Dol",
    "Aldin Donic", "John Doeić" };
    osobePretragaComboBox.DataSource = poznateOsobe;
}
```

Kod 7.6: Dodavanje liste opcija za *combo box* poznateOsobe

Scenarij izvršavanja ovog programskog rješenja:



Pored događaja ComboBox kontrole koji su kombinacija ključnih događaja TextBox kontrole (TextChanged koji se aktivira nakon što se sadržaj uređivača izmjeni) i ListBoxa (SelectedIndexChanged, SelectedValueChanged), postoje i specifični događaji za ovu kontrolu. To su DropDown događaj koji se aktivira nakon što se prikaže dropdown listbox i DropDownClosed koji se aktivira nakon što se sakrije dropdown listbox.

RAZVOJ PROGRAMSKIH RJEŠENJA

7.3. `NumericUpDown` kontrola

Ukoliko je potrebno da se ograniči korisnički ulaz na izbor specifičnog ranga numeričkih vrijednosti, koristi se `NumericUpDown` kontrola. Ova kontrola se pojavljuje kao `TextBox` sa dva ‘mala dugmeta’ (strelice) na desnoj strani. Korisnik može unositi numeričke vrijednosti u ovu kontrolu ili klikom na strelice povećavati i smanjivati vrijednosti kontrole. Najveća i najmanja vrijednost u rangu je specificirana sa `Maximum` i `Minimum` osobinama.

Tabela 7.4 opisuje često korištene osobine i događaje klase `NumericUpDown`.

NumericUpDown osobine i događaji	Opis
Osobine	
Increment	Specificira za koliko se tekući broj u kontroli mijenja kada se klikne strelica prema gore ili dolje.
Maximum	Najveća vrijednost u rangu vezanom za kontrolu.
Minimum	Najmanja vrijednost u rangu vezanom za kontrolu.
DecimalPlaces ThousandsSeparator	DecimalPlaces određuje broj mesta iza decimalnog zareza. ThousandsSeparator opcija služi za navođenje separatora za određivanje prikaza velikih vrijednosti.
Value	Numerička vrijednost trenutno prikazana na kontroli.
Događaji	
ValueChanged	Ovaj događaj se dešava nakon promjene vrijednosti kontrole.

Tabela 7.4: `NumericUpDown` osobine i dogadaji

Primjer: Napisati programsko rješenje koje omogućava da se unosi težina u kilogramima pomoću `NumericUpDown` kontrole i izračunavanje cijene po formuli $cijena = \text{težina_u_kg} * 3.5 + 7$. Onemogućiti unos negativnih težina, kao i težina većih od 10 kilograma. Inkrement postaviti na 0.1 kg, rezoluciju na 1 decimalu.

RAZVOJ PROGRAMSKIH RJEŠENJA

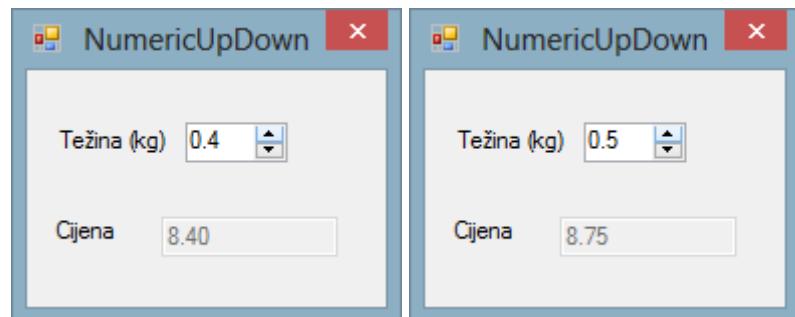
Opis rješenja: Kreirana je Windows Forms aplikacija. Na početnu formu je postavljena NumericUpDown kontrola. Preko Properties editora mogu se promijeniti osobine kontrole da bi se zadovoljili zahtjevi iskazani u zadatku.

Metoda za upravljanjem događajem uslijed promijene vrijednosti NumericUpDown kontrole sadrži programsku logiku vezanu za izračunavanje cijene. Prikazana je kodom 7.7.

```
private void ulaznaTezinaNumericUpDown_ValueChanged(object sender, EventArgs e)
{
    Decimal unesenaTezina = ulaznaTezinaNumericUpDown.Value;
    cijenaTextBox.Text = (unesenaTezina * 3.5m + 7).ToString();
}
```

Kod 7.7: Metoda za upravljanje događajem `ulaznaTezinaNumericUpDown_ValueChanged`

Scenarij izvršavanja:



7.4. MonthCalendar kontrola

Mnoge aplikacije moraju izvršavati kalkulacije zasnovane na datumu i vremenu. .NET Framework obezbeđuje dvije kontrole koje dozvoljavaju aplikaciji dobijanje informacija o datumu i vremenu MonthCalendar i DateTimePicker kontrole.



MonthCalendar kontrola

RAZVOJ PROGRAMSKIH RJEŠENJA

MonthCalendar kontrola prikazuje mjesecni kalendar na formi. Korisnik može izabrati datum iz tekućeg prikazanog mjeseca, ili može koristiti obezbjeđeni link da bi navigirao na drugi mjesec. Podrazumijevani događaj za ovu kontrolu je `DateChanged`, koji se generira kada se novi datum izabere. MonthCalendar osobine i događaji su sumirani u tabeli 7.5.

MonthCalendar osobine i događaji	Opis
Osobine	
FirstDayOfWeek	Postavlja koji dan sedmice će se prvi prikazivati za svaku sedmicu kalendarja.
MaxDate	Posljednji datum koji može biti selektiran.
MaxSelectionCount	Maksimalan broj izbora-datuma koji mogu biti selektirani istovremeno.
MinDate	Prvi datum koji može biti selektiran.
MonthlyBoldedDates	Datumi koji će se prikazivati u bold stilu na kalendaru.
SelectionEnd	Posljedni selektirani datum.
SelectionRange	Selektirani datumi.
SelectionStart	Prvi selektirani datum.
Događaj	
DateChanged	Generira se kada se selektira datum na kalendaru.

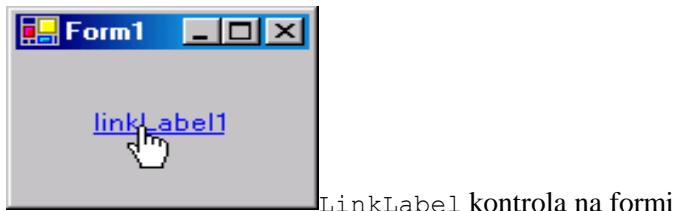
Tabela 7.5: MonthCalendar osobine i događaji

7.5. LinkLabel kontrola

LinkLabel kontrola prikazuje linkove na druge resurse, kao npr: fajl ili Web stranice. LinkLabel se pojavljuje kao podvučen tekst (plavo obojen). Kada se miš pomjera preko linka, pointer mijenja oblik (npr. u ruku); to je slično ponašanje kao *hyperlink* na Web stranici. Link može mijenjati boju da indicira da li je link novi, već posjećen ili aktivan. Kada se klikne na LinkLabel generira se `LinkClicked` događaj. Klasa LinkLabel je izvedena iz klase `Label` i zbog toga nasljeđuje sve njene osobine i funkcionalnosti. Tabela 7.6 prikazuje osnovne osobine i događaje LinkLabel kontrole.

LinkLabel osobine i događaji	Opis
Osobine	
ActiveLinkColor	Specificira boju aktivnog linka.
LinkArea	Specificira koji dio teksta na LinkLabel kontroli je dio linka.
LinkBehavior	Specificira ponašanje linka, kao npr: izgled linka kada je miš preko kontrole.
LinkColor	Specificira originalnu boju linka prije nego je posjećen. Inicijalna boja je postavljana sa sistemom i obično je plava.
Text	Specificira tekst kontrole.
VisitedLinkColor	Specificira boju posjećenih linkova.
Događaj	(Argument događaja LinkLabelLinkClickedEventArgs)
LinkClicked	Generira se kada se klikne na link.

Tabela 7.6: LinkLabel osobine i događaji

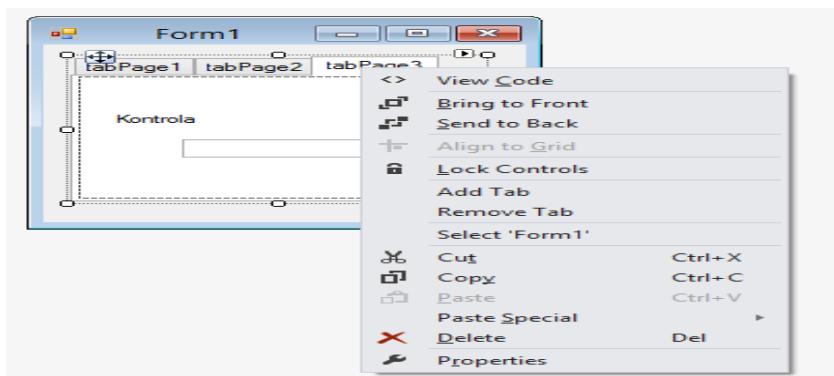


LinkLabel kontrola na formi

7.6. TabControl kontrola

TabControl kontrola sadržiTabPage objekte, koji su slični kontrolama Panel i GroupBox. Ova kontrola dozvoljava da se specificiraju informacije (kontrole) na odvojenim tabovima na formi. TabControl upravlja povezanim skupom tab stranica (TabPage). Samo jedan tabPage može biti istovremeno prikazan. Tabela 7.7 prikazuje neke od osobina i događaja kontrole TabControl.

Slika ispod prikazuje TabControl u dizajner modu. Jednostavno je dodavanje i brisanje novih stranica (TabPage). Na svaku stranicu se dodaju kontrole.



Dodavanje tabova u kontrolu TabControl se postiže sljedećim programskim iskazima:

```
this.tabControl1.Controls.Add(this.tabPage1);  
  
this.tabControl1.Controls.Add(this.tabPage2);  
  
this.tabControl1.Controls.Add(this.tabPage3);
```

Dodavanje kontrola na stranicu tri se postiže sa programskim iskazima:

```
this.tabPage3.Controls.Add(this.label1);  
  
this.tabPage3.Controls.Add(this.textBox1);
```

RAZVOJ PROGRAMSKIH RJEŠENJA

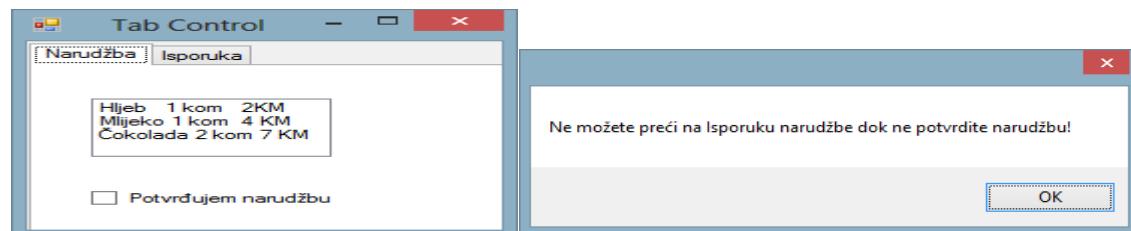
TabControl osobine i dogadaji	Opis
ImageList	Specificira slike koje se prikazuju na tabovima.
ItemSize	Specificira veličinu taba.
SelectedIndex	Indeks selektirane tab stranice.
SelectedTab	Selektirana tab stranica.
TabCount	Vraća broj tab stranica.
TabPage	Kolekcija tab stranica unutar tab kontrola.
Događaj	
SelectedIndexChanged	Generira se kada se SelectedIndex promijeni (tj., izabere se druga stranica -TabPage).

Tabela 7.7: TabControl osobine i događaji

Primjer: Razviti programsko rješenje koje će omogućiti potvrdu narudžbe i njenu isporuku. Isporuka se može uraditi tek nakon potvrde narudžbe.

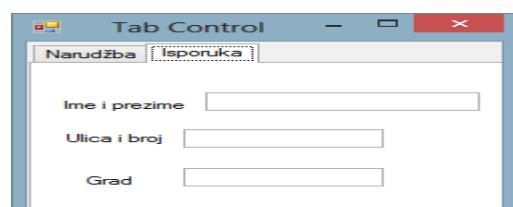
Rješenje: Na početnoj formi Windows aplikacije kreira se TabControl sa dva taba imenovana: Narudžba, Isporuka. Na stranici Narudžba nalazi se spisak odabranih proizvoda i *check box* kojim je potrebno potvrditi narudžbu. Na stranici Isporuka se unose podaci o klijentu kojem se vrši isporuka.

Scenarij izvršavanja:



Aktivna stranica Narudžba

Pokušaj prelaska na stranicu Isporuka bez potvrde.



RAZVOJ PROGRAMSKIH RJEŠENJA

Nakon potvrde – markiranja *check boxa* na stranici Narudžba prelazi se na stranicu Isporuka.

Da bi se ostvarila kontrola prelaska sa jedne stranice na drugu potrebno je programirati događaj prodajaTabControl_Deselecting. Kod 7.8 prikazuje metodu za upravljanje pomenutim događajem.

```
private void prodajaTabControl_Deselecting(object sender,
TabControlCancelEventArgs e)
{
    if (e.TabPage == prodajaTabControl.TabPages[0])
    {
        if (!potvrdaNarudzbaCheckBox.Checked)
        {
            MessageBox.Show("Ne možete preći na Isporuku narudžbe dok ne potvrdite
narudžbu!");
            e.Cancel = true;
        }
    }
}
```

Kod 7.8: Metoda za upravljanje događajem TabControl_Deselecting

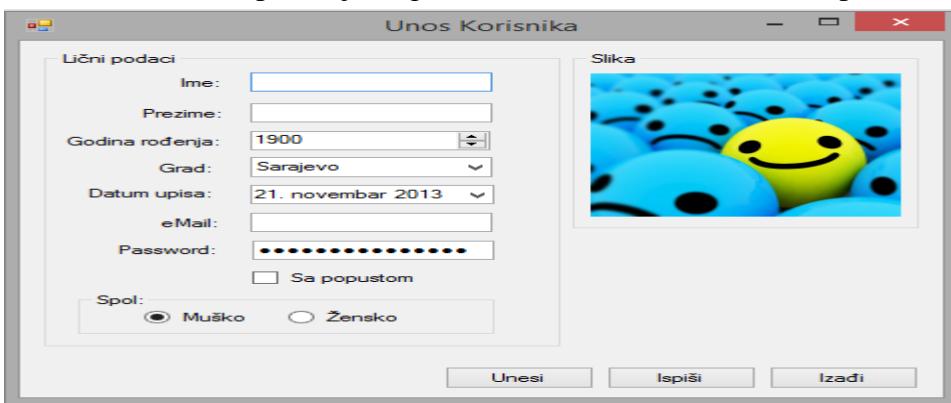
Unutar metode prvo se provjerava da li je trenutno selektirana tab stranica ona stranica koja je potrebna, a nakon toga se provjerava da li je korisnik pristao na pravila i uvjete korištenja. Ako nije, ne dozvoljava se promjena aktivne tab stranice, u suprotnom promjena se dopušta.

Rezime:

U okviru ovog poglavlja prikazane su dodatne GUI kontrole za prikaz slika, izbor opcija, ranga numeričkih vrijednosti, kontrole za rad sa datumom. Prikazana je i kontrola za kreiranje stranica na odvojenim tabovima koja se često koristi pri dizajnu programskih rješenja. Za detaljnije upoznavanje kontrola, njihovih osobina i metoda veoma je važno proučavanje dokumentacije za ove kontrole koje su sastavni dio većine programskih okruženja.

Pitanja i zadaci za samostalni rad:

1. Objasnite kontrolu za prikazivanje slika.
2. Opišite `ListBox` kontrolu – namjenu, osobine, osnovni događaj.
3. Opišite `ComboBox` kontrolu-namjenu, osobine, osnovni događaj.
4. Koja je razlika između *list box* i *combo box* kontrole?
5. Opišite `NumericUpDown` kontrolu-namjenu, osobine, osnovni događaj.
6. Opišite `LinkLabel` kontrolu-namjenu, osobine, osnovni događaj .
7. Opišite `TabControl` kontrolu-namjenu, osobine, osnovni događaj.
8. Navedite kontrole za rad sa datom i vremenom.
9. Koje bi kontrole koristili za dizajniranje forme na kojoj će biti polja: Skladište (mogu biti dva tipa: Maloprodajno, Veleprodajno), Šifra proizvoda, Naziv Proizvoda (dodjeljuje se iz unaprijed predefinirane liste od 1000 proizvoda), Količina, Cijena bez PDV i Cijena sa PDV-om (računa se kao Cijena bez PDV * 17%).
Forma će služiti za unos. Kojom kontrolom i događajem na toj kontroli će se izvršiti spašavanje unesenih podataka.
10. Koje bi kontrole koristili ukoliko je potrebno dizajnirati formu na kojoj će biti polja: Tip uposlenika (mogu biti dva tipa: Nastavno osoblje, Ostalo osoblje), ime uposlenika, slika uposlenika. Skicirajte formu, kontrole, navedite naziv kontrola i eventualne događaje koji se uz njih vežu.
11. Kreirati Windows Forms aplikaciju sa početnom formom kao na slici ispod:



RAZVOJ PROGRAMSKIH RJEŠENJA

Aplikacija treba omogućiti korisniku da unosi osnovne informacije o osobi. Unesene podatke smjestiti u listu osoba. Potrebno je poštovati pravila programske dekompozicije.

Korisnik pritiskom na dugme "Unesi" unosi novu osobu (predstavljenu sa klasom Osoba) u listu osoba nakon čega aplikacija prikazuje poruku da je korisnik uspješno unesen te mijenja naslov forme u "Unos korisnika #" gdje # predstavlja broj unesenih korisnika.

Pritiskom na dugme "Ispiši" korisniku se prikazuje nova forma. Konstruktoru nove forme se prosljeđuje lista osoba, koja je unesena na glavnoj formi. Tu listu osoba treba ispisati na novoj formi. Ispis uraditi tako što se preklopi metoda `ToString` klase Osoba.

```
public override String ToString() { return Ime+ " " +Prezime; }
```

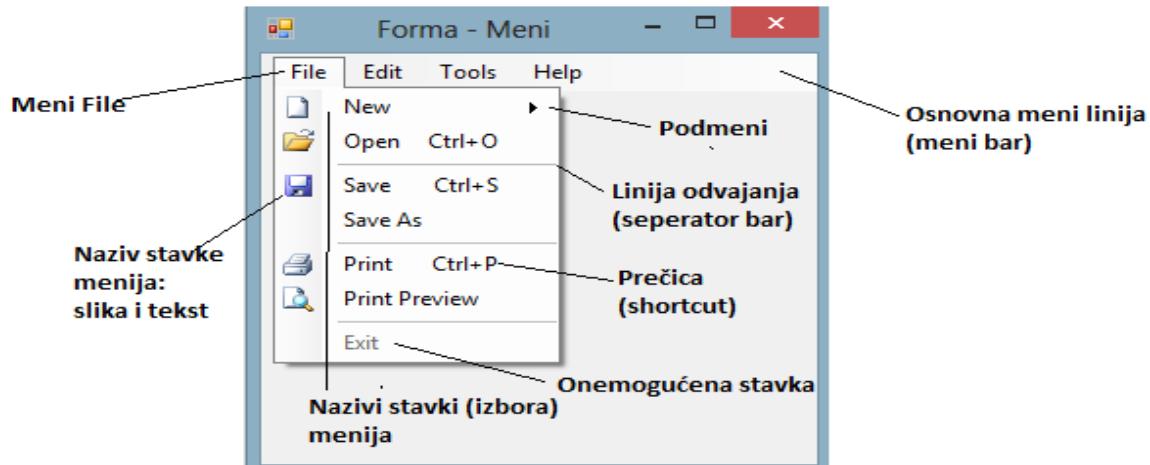
Predavanje 8: GUI kontrola – Meni i MDI aplikacije

U okviru ovog poglavlja prikazana je svrha i vrste meni kontrole, način kreiranja menija, osobine i događaji vezani za meni kroz sljedeće tematske jedinice:

1. Osnovni koncepti vezani za GUI kontrolu meni
2. MDI (Multiple Document Interface) aplikacije

8.1. Osnovni koncepti vezani za GUI kontrolu meni

Korisnicima programskih rješenja potrebno je omogućiti jednostavan prilaz svim dostupnim izborima/alternativama na odgovarajućoj tačci korištenja sistema. Već u prethodnim poglavljima prikazane su GUI kontrole koje se mogu koristiti za izbor opcija. Neke od njih su dugmad, radio dugmad, *check boxovi*, liste izbora. Postoji i specijalizirana GUI komponenta koja omogućava kreiranje strukturiranih izbornika-menija. To je meni (*menu*) komponenta koja je u okviru Visual Studioa omogućena sa `MenuStrip` klasmom. Primjer menija kreiranog pomoću navedene klase i predefiniranog menija u Visual Studio okruženju dat je na slici 8.1.



Slika 8.1: Primjer menija sa naznačenim elementima

Najviši nivo menija se naziva osnovna meni linija-menu bar i na njoj se nalaze nazivi pojedinačnih izbora/menija koji predstavljaju osnovne funkcije sistema (File, Edit, Tools,...). Meni bar može imati različite orijentacije. Najčešće je to horizontalna (na slici 8.1) ili vertikalna orijentacija.

RAZVOJ PROGRAMSKIH RJEŠENJA

Svaki meni se uglavnom sastoji od niza stavki, odnosno izbora (npr. `File` meni se sastoji od stavki/izbora `New`, `Open`, `Save`,...). Nazivi stavki mogu biti tekstualni, slika ili kombinacija slike i teksta (npr. opcija `Save` je označena i sa tekstrom i slikom). Meniji sa slikama (ikonama) pomažu prepoznavanju zadatka i bržoj selekciji izbora.

I sama stavka menija može sadržavati skup izbora koji se nazivaju podmeniji ili *drop-down* meniji. Npr. `New` stavka sadrži podmeni od dvije stavke `Project`, `Web` sajt. Ukoliko *drop-down* meniji i sami sadrže podmenije formiraju se kaskadni meniji. Kaskadni meniji se koristi za grupiranje pojedinih alternativa u logičku cjelinu što doprinosi jednostavnijoj strukturi menija. Na samom meniju se mogu također formirati grupe opcija korištenjem linije separacije. Meniji mogu imati pomicne trake, ali se ne preporučuje njihovo korištenje.

Naziv menija, podmenija, stavke menija trebaju jednoznačno opisati njihovu namjenu. Organizacija menija i podmenija treba da pruži najefektivniju sekvencu koraka za izvršavanje nekog zadatka. Redoslijed opcija treba da je u skladu sa prirodnim redoslijedom izvršavanja nekog zadatka. Lista stavki izbora može se urediti i da je u skladu sa učestalosti korištenja ili važnosti stavki. Preporučljivo je da se naznače stavke menija koje su već selektirane i aktivne. Pretežno se aktivni izbori obilježavaju sa indikatorom (*check mark*) sa lijeve strane.

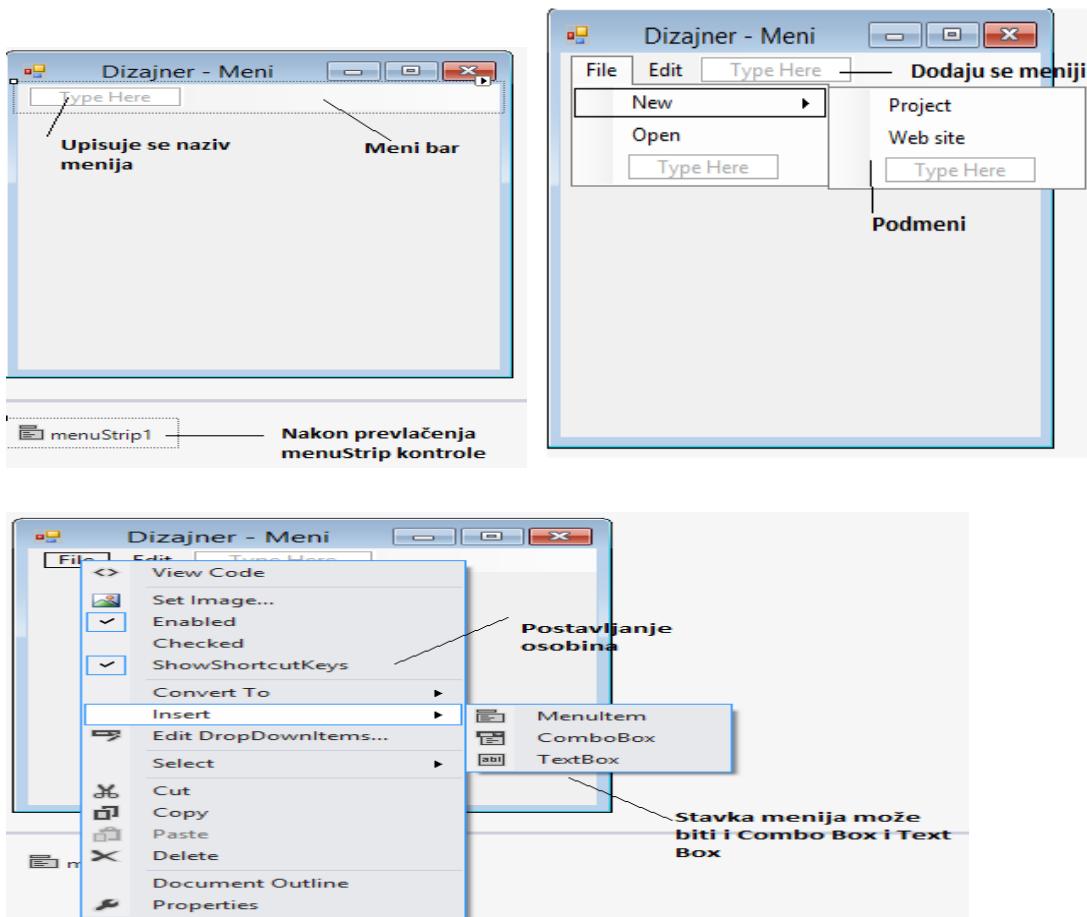
Opcije na meniju koje nisu dozvoljene za korištenje određenom korisniku ili u nekoj tačci izvršavanja sistema treba onemogućiti (npr. `Exit` opcija). Preporučljivo je sivom bojom prikazati neaktivne izbore.

Za često korištene akcije, da bi se ubrzalo njihovo izvršavanje, dobro je omogućiti ekvivalente sa tastature (npr. tipke `CTRL + O` za `Open`). Koristiti standardne kombinacije ekvivalenta sa tastature ukoliko one postoje za određene akcije. U opisu pored naziva menija koristiti znak plus da se naznači da tipke moraju biti korištene istovremeno.

8.1.1.Dizajner i meni

U okviru dizajnera da bi se kreirao meni na formi, sa `Toolboxa` se povlači `MenuStrip` kontrola na formu. To kreira meni bar ispod naslovne linije forme i postavlja `MenuStrip` ikonu u prostor za komponente. Nakon toga u dizajner modu mogu se dodavati odgovarajući generalni meniji i njihove opcije izbora, što je prikazano na slici 8.2. Meniji, kao i druge kontrole, imaju osobine, metode i događaje kojima se može pristupati preko prozora za osobine i programskim kodom.

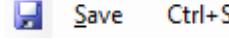
RAZVOJ PROGRAMSKIH RJEŠENJA



Slika 8.2: Kreiranje menija sa dizajnerom

Osnovni događaj `menuStrip` kontrole je `Click` događaj koji se dešava nakon izbora neke od ponuđenih opcija. Najčešće akcije kao odgovor na ovaj događaj su prikazivanje nove forme koja se koristi za obradu neke funkcije programskog rješenja, prikazivanje dijalog prozora i prozora za postavljanje osobina. Osnovne meni osobine koje su izložene i u tekstu iznad sumirane su u tabeli 8.1.

RAZVOJ PROGRAMSKIH RJEŠENJA

	Opis
MenuStrip osobine	
MenuItems	Kolekcija izbora na nivou glavnog meni bara (File>Edit,...).
LayoutStyle	Specificira orientaciju menija. Neke od opcija su HorizontalStackOverflow, VerticalStackOverflow, Table.
ToolStripMenuItem Properties (Svojstva koja se odnose na stavke menija.)	
DropDownItems	Specificira kolekciju stavki za meni. U okviru dizajnera koristi se Collection editor za njihovo nabranje.
Checked	Indicira da li je stavka menija aktivna (checked). Inicijalna podrazumijevana vrijednost je false-nije aktivna. 
CheckOnClick CheckState	Određuje da li je moguća promjena stanja stavke prilikom klika na stavku. Određuje stanje stavke prilikom aktiviranja menija (checked, unchecked).
Enabled	Određuje da li je stavka menija omogućena. Podrazumijevana vrijednost je true i znači da je stavka omogućena, sa false se stavka onemogućava i prikazuje se kao siva. 
Visible	Određuje vidljivost stavke na meniju. Inicijalna podrazumijevana vrijednost je true i znači da je stavka vidljiva.
DisplayStyle	Određuje stil prikaza naslova. Neke od opcija su ImageAndText (naziv menija može biti slika i tekst), Text (naziv menija može biti samo tekst), Image (naziv menija je označen sa slikom). Meni sa DispalyStyle.ImageAndText 
Image	Specificira lokaciju slike koja se koristi za naziv stavke.
ShortcutKeys	Specificira tipke prečice (shortcut keys) za stavku menija.

RAZVOJ PROGRAMSKIH RJEŠENJA

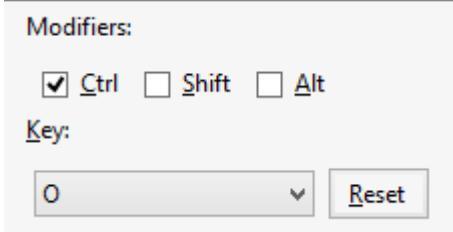
	Opis
	
ShowShortcutKeys	Indicira da li se prečica prikazuje pored teksta stavke menija. Inicijalna podrazumijevana vrijednost je <code>true</code> sa značenjem da se prikazuje. <code>Open Ctrl+O</code>
Text	Specificira tekst (naziv) za stavku menija.
ToolStripMenuItem Događaji	
Click	Generira se kada se klikne na stavku menija ili kada se koristi prečica za meni.
CheckState	Generira se kada se promijeni stanje izbora menija označenog sa atributom <code>Checked</code> .

Tabela 8.1: MenuStrip i ToolStripMenuItem osobine i događaji

8.1.2. Dinamičko dodavanje menija na formu

Slijedi primjer dinamičkog (uz pomoć klase se piše programski kod koji se aktivira za vrijeme izvršavanja) dodjeljivanja menija na formu. Potrebno je kreirati meni `File` koji sadrži tri stavke izbora `New`, `Open`, `Close`. Opcije izbora `New` i `Open` potrebno je linijom separacije odvojiti od izbora `Close`. Prečica za `Open` stavku je `CTRL+O`. Uz stavku `Open` je vezan i događaj `Click` koji kada se aktivira ispisuje poruku koja sadrži naziv izbora i veličinu kontrole. Kod 8.1 odgovara opisanom zadatku.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
using System.Windows.Forms;
namespace ProgramiranjeMenijaK
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            // 1. kreiranje glavne meni linije-meni bara i dodavanje stavke File
            MenuStrip meniGlavni = new MenuStrip();
            ToolStripMenuItem meniFile = new ToolStripMenuItem("File");
            Controls.Add(meniGlavni);

            //2.Kreiranje stavki menija (New, Open) za meni File
            ToolStripMenuItem meniFileNew = new ToolStripMenuItem("New");
            ToolStripMenuItem meniFileOpen = new ToolStripMenuItem("Open");

            //3. povezivanje Open stavke sa događajem Click
            // i postavljanje prečice za stavku Open
            meniFileOpen.Click += new System.EventHandler(this.meniFileOpen_Click);
            meniFileOpen.ShortcutKeys = Keys.Control | Keys.O;

            //4. kreiranje linije odvajanja
            ToolStripSeparator meniSeparator = new ToolStripSeparator();

            //5.Kreiranje stavke Close
            ToolStripMenuItem meniFileClose = new ToolStripMenuItem("Close");

            // 6.Dodavanje kreiranih stavki i linije odvajanja u File meni
            meniFile.DropDownItems.AddRange(new System.Windows.Forms.ToolStripItem[]
            { meniFileNew, meniFileOpen,meniSeparator,meniFileClose});

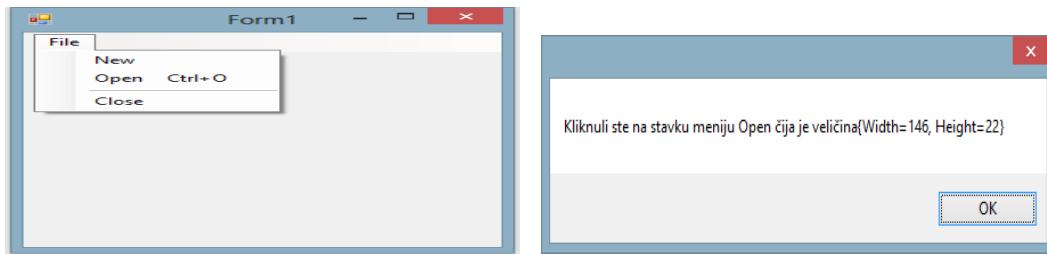
            // 7. Dodavanje glavne meni linije na formu
            meniGlavni.Items.Add(meniFile);
        }

        // Metoda za upravljanje događajem Click Open stavke menija
        private void meniFileOpen_Click(object sender, EventArgs e)
        {
            ToolStripMenuItem stavka = sender as ToolStripMenuItem;
            MessageBox.Show("Kliknuli ste na stavku meniju " + stavka.Text +
                " čija je veličina" + stavka.Size );
        }
    }
}
```

Kod 8.1: Dinamičko (rad sa klasama) kreiranje menija na formu

RAZVOJ PROGRAMSKIH RJEŠENJA

Na slikama ispod prikazan je mogući scenarij izvršavanja.



Otvoren meni File, izabrana stavka Open, aktiviran događaj Click stavke Open

8.1.3. Kontekstni meniji

Kontekstni meniji (*context menu*) se vežu uz pojedine GUI kontrole. Nazivaju se i *pop-up* meniji. Aktiviraju se desnim klikom na kontrolu. Koriste se da pruže dodatni skup izbora koji je specifičan u određenoj tačci korištenja sistema. U Visual Studio okruženju kreiraju se sa ContextMenuStrip kontrolom, koja ima većinu karakteristika kao i ToolStrip kontrola. ContextMenuStrip može se kreirati u okviru dizajnera i povezati sa nekom kontrolom preko ContextMenuStrip osobine te kontrole, ili se može napisati programski kod i izvršiti povezivanje kontekstnog menija i kontrole za vrijeme izvršavanja programa.

Primjer: Dinamički kreirati kontekstni meni sa dvije stavke izbora: Povećaj i Smanji. Povezati kontekstni meni sa slikom koja se nalazi na formi.

Opis rješenja: Pomoću dizajnera je kreirana forma i slika (pictureBox1). Dio koda kojim je kreiran kontekstni meni i dodjeljen slici obilježen je sa kod 8.2:

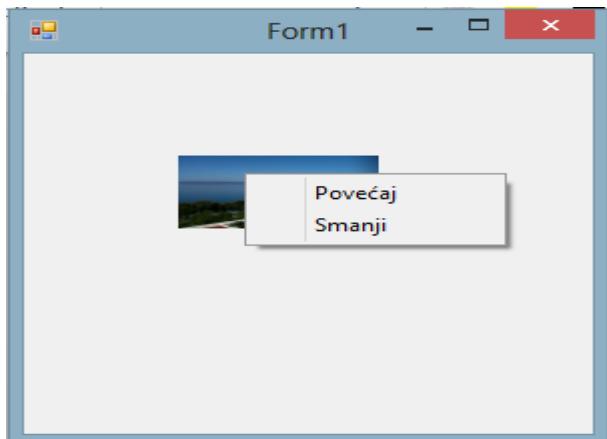
```
using System;
using System.Windows.Forms;
namespace ContextMenuProgramiranje
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            // 1. Deklaracija kontekstnog menija cm i stavki menija
            ContextMenuStrip cm = new ContextMenuStrip();
            ToolStripMenuItem meniFileNew = new ToolStripMenuItem("New");
            ToolStripMenuItem meniFileOpen = new ToolStripMenuItem("Open") ;

            // 2.dodavanje stavki New, Open u konteksni meni cm
            cm.Items.AddRange(new System.Windows.Forms.ToolStripItem[] {
            meniFileNew,meniFileOpen}) ;

            // 3. Povezivanje konteksnog menija cm sa slikom
            pictureBox1.ContextMenuStrip = cm;
        }
    }
}
```

Kod 8.2: Kreiranje kontekstnog menija i povezivanje sa slikom

Scenarij izvršavanja:



Aktiviran kontekstni meni na slici

Prednosti kontekstnog menija su pojavljivanje po potrebi u radnoj oblasti, kada nisu potrebni ne koriste prostor prozora, a njihova vertikalna orientacija je pogodna za skeniranje i grupiranje.

RAZVOJ PROGRAMSKIH RJEŠENJA

Nedostaci su da korisnik mora znati da postoje i kako se aktiviraju. Često mogu prekriti i bitne informacije na formi (radnoj površini). Preporučuje se da ovi meniji imaju maksimalno do deset stavki.

8.1.4. Prozor meni

Prozor meni (*window menu*) je meni namijenjen da prikaže drugi prozor koji se obično koristi kao posrednik za izvršavanje neke akcije. To može biti namjenski kreirani prozor za setovanje opcija, dijalog *box* prozor ili neka forma specifična za domenski problem. Obično se ovi meniji naglašavaju sa tri tačke (...) nakon naziva prozora.

Postoji više raspoloživih dijalog *box* komponenti. Često korišteni dijalog *boxovi* su `ColorDialog` koji se koristi za izbor boje i postavljanje boje na formu, `FontDialog` koristi se za izbor i postavljanje fonta, `PrintDialog` koristi se za izbor štampača i štampanje dokumenta. Nadalje tu su i dijalog *boxovi* koji omogućavaju rad sa direktorijima i fajlovima (`FolderDialogBox`, `OpenFileDialog`, `SaveFileDialog`).

Primjer: Kreirati formu sa dva tekstualna polja koja se koriste za unos predmeta i broja studenata i dugmetom Potvrdi. Na formi je potrebno omogućiti i meni Stil sa opcijama Font i Boja, sa ciljem izbora fonta i pozadinske boje.

Opis rješenja: Kreirane su sve potrebne kontrole korištenjem dizajnera.

Za meni opciju Font i Boja napisane su metode za upravljanje događajem klik koje će prikazati dijalog *boxove* za Font i Boju. Kod 8.3 prikazuje metodu za upravljanje događajem klik stavke menija Font.

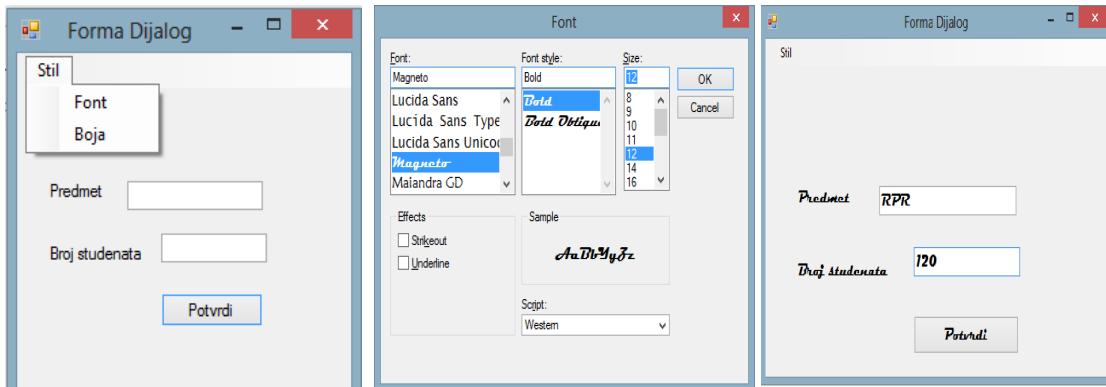
```
private void fontToolStripMenuItem_Click(object sender, EventArgs e)
{
    FontDialog fontDialog1 = new FontDialog();
    if (fontDialog1.ShowDialog() == DialogResult.OK)
        Font = fontDialog1.Font;

}
```

Kod 8.3: Metoda za upravljanje događajem klik stavke menija Font prikazuje `FontDialog` *box*

RAZVOJ PROGRAMSKIH RJEŠENJA

Scenarij izvršavanja:



Početna forma sa otvorenim menijem Stil,

Font dijalog prozor pojavljuje se nakon izbora Font opcije,

Početna forma sa promijenjenim fontom

Izborom Stil opcije Boja pojavio bi se dijalog *box* za izbor boje i izvršila bi se metoda za upravljanjem događajem `Click` koja bi prikazala standardni dijalog *box* za odabir boje (kod 8.4).

```
private void bojaToolStripMenuItem_Click(object sender, EventArgs e)
{
    ColorDialog bojaDialog = new ColorDialog();

    if (bojaDialog.ShowDialog() == DialogResult.OK)
        BackColor = bojaDialog.Color;
}
```

Kod 8.4: Metoda za upravljanje događajem klik stavke menija Boja prikazuje `ColorDialog` *box*

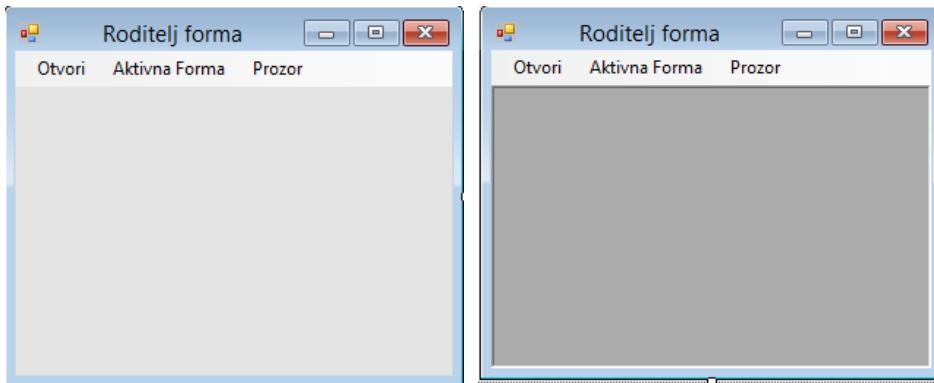
8.2.MDI (Multiple Document Interface) aplikacije

SDI (Single Document Interface) aplikacije dozvoljavaju korisniku obavljanje zadataka preko jednog prozora. Mnoge novije aplikacije dozvoljavaju korisniku da rade sa više prozora u isto vrijeme i takve aplikacije se nazivaju – MDI (Multiple Document Interface) aplikacije. Primjer MDI aplikacije npr. Microsoft Word, koji omogućava rad sa više dokumenata istovremeno preko više otvorenih prozora.

Glavni aplikacijski prozor MDI programa se naziva roditelj prozor (*parent windows*), a svaki prozor unutar aplikacije je označen kao dijete (*child*) prozor. MDI aplikacije mogu imati više funkcionalno različitih prozora u svojstvu djece. Maksimalno jedno dijete (jedan prozor) može biti aktivno u jednom trenutku. Djeca prozori se ponašaju kao i bilo koji drugi prozori u pogledu zatvaranja, minimiziranja, promjene veličine.

RAZVOJ PROGRAMSKIH RJEŠENJA

Da bi forma postala MDI roditelj, treba joj dati i to svojstvo, postavljanjem njene `IsMdiContainer` osobine na `true`. Pri tome kontrola `Form` mijenja izgled kao na slici 8.3.



Slika 8.3: SDI i MDI forma

Nakon kreiranja roditelj forme, kreira se forma dijete i povezuje se sa roditelj formom preko `MdiParent` osobine.

Programski kod kojim se dodaje dijete na roditelj kontrolu `Form` je:

```
PredmetDijete1Forma dijete1 = new PredmetDijete1Forma();  
dijete1.MdiParent = this; // povezivanje dijete forme sa roditelj formom  
dijete1.Show(); // prikaži dijete
```

Kod za kreiranje djeteta je obično u okviru neke metode za upravljanje događajem, koja kreira novi prozor kao odgovor za korisničku akciju. U većini slučajeva, roditelj `Form` kreira dijete, tako da je referenca na roditelja `this`.

Kada se kreira MDI aplikacija, dobro je uključiti meni koji prikazuje listu prozora koji su otvoreni sa označenim aktivnim prozorom. To pomaže bržem izboru dijete prozora. Osobina `MdiWindowListItem` klase `MenuStrip` se koristi za prikaz liste MDI formi djece (ilustrirano ispod primjerom).

Primjer: Kreirati MDI roditelj formu koja sadrži 2 meni izbora na meni baru. Prvi meni Otvori, sadrži stavke Predmet i Student. Drugi meni, Prozor, obezbeđuje opcije za formu prikaza aktivne djece u kaskadnom, horizontalnom, vertikalnom obliku i prikazuje otvorene prozore.

Opis rješenja: Prvo se kreira roditelj forma sa gore navedenom strukturom menija. U `Properties` prozoru, postavlja se svojstvo roditelj forme `IsMdiContainer` na `true`. Nakon toga kreiraju se djeca forme Predmet i Student. Programskim kodom u okviru klik događaja

RAZVOJ PROGRAMSKIH RJEŠENJA

stavke Predmet i Student povezuje se ove forme sa roditelj formom. Prozor meni se koristi za prikaz otvorenih prozora i za prikaz stavki za uređenje djece u kaskadnom, vertikalnom i horizontalnom obliku.. Kod 8.5 prikazuje roditelj klasu i povezivanje MDI forme sa djecom.

```
using System;
using System.Windows.Forms;
namespace MDI
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            // Dodjeljivanje menija za prikaz liste djece-formi
            menuStrip1.MdiWindowListItem = prozorToolStripMenuItem;
        }

        // Povezivanje Predmet forme sa stavkom menija Otvori - Predmet
        private void predmetToolStripMenuItem_Click(object sender, EventArgs e)
        {
            PredmetDijete1Forma dijete1 = new PredmetDijete1Forma();
            dijete1.MdiParent = this;
            dijete1.Show();           // prikaži dijete
        }

        // Povezivanje Student forme sa stavkom menija Otvori - Student
        private void studentToolStripMenuItem_Click(object sender, EventArgs e)
        {
            StudentDijete2Forma dijete2 = new StudentDijete2Forma();
            dijete2.MdiParent = this;
            dijete2.Show();           // prikaži dijete
        }
}
```

Kod 8.5: Formiranje menija i povezivanje MDI forme sa djecom

Metode za upravljanje uređenjem prozora pozivaju metodu `LayoutMdi` sa parametrima iz `MdiLayout` nabrojive liste date su sa kodom 8.6.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
// Metode za upravljanje događajima Prozor-Kaskadni, Horizontalni, Vertikalni
private void kaskadniToolStripMenuItem_Click(object sender, EventArgs e)
{
    this.LayoutMdi(MdiLayout.Cascade);
}

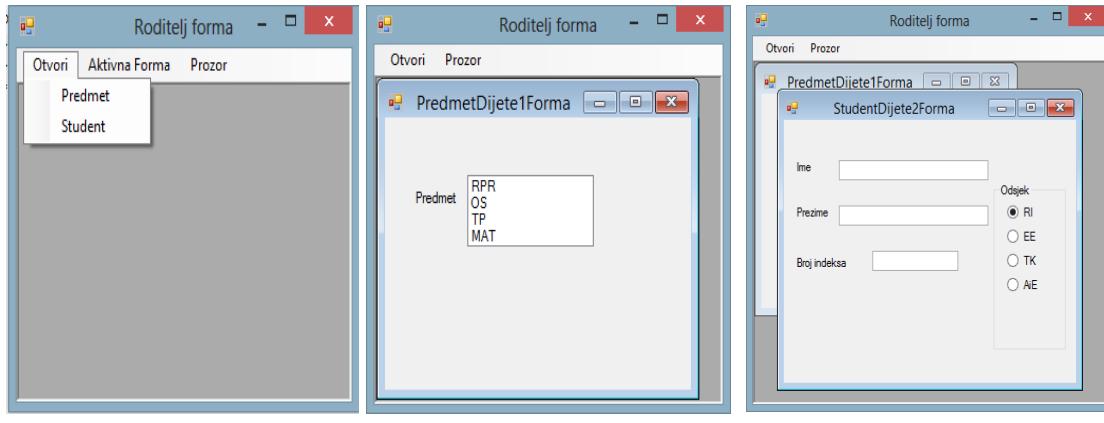
private void horizontalniToolStripMenuItem_Click(object sender, EventArgs e)
{
    this.LayoutMdi(MdiLayout.TileHorizontal);
}

private void vertikalniToolStripMenuItem_Click(object sender, EventArgs e)
{
    this.LayoutMdi(MdiLayout.TileVertical);
}

}
```

Kod 8.6: Metode za uređivanje prozora kaskadno, horizontalno i vertikalno

Mogući scenarij izvršavanja prikazanog programskog rješenja:

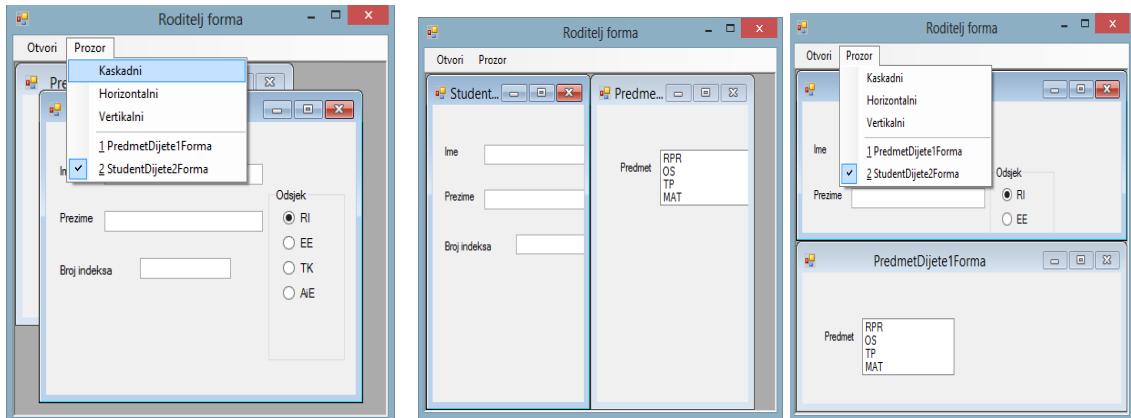


Osnovna forma

Otvoren prozor za
formu Predmet

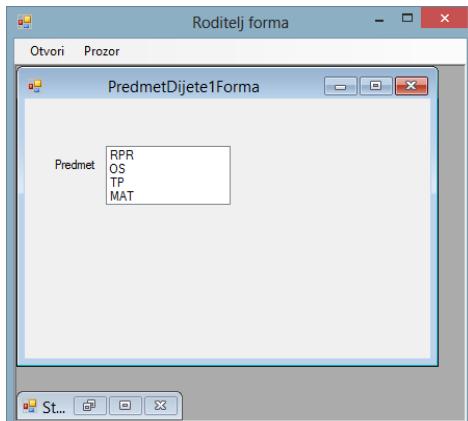
Otvoren prozor za
formu Student

RAZVOJ PROGRAMSKIH RJEŠENJA



Prikaz formi kaskadno, vertikalno, horizontalno. Prozor meni prikazuje otvorene prozore i aktivan prozor.

Djeca prozori mogu se neovisno minimizirati, maksimizirati i zatvarati. Kada se roditelj minimizira ili zatvori, dijete prozor se također minimizira ili zatvori. Kada se dijete prozor minimizira ili maksimizira, njegova naslovna linija prikazuje ikone, koje se mogu koristiti da se dijete prozor vrati u prethodno stanje.



Minimiziran prozor Student

U prikazanom kodu za roditelj klasu koristi se i osobina `MdiWindowsListItems` koja sadrži listu djece-formi vezanih uz roditeljsku MDI formu. Korisna je i osobina `ActiveMdiChild` koja vraća podatke o trenutnom aktivnom djjetetu ili `null` ako nema aktivne dijete forme.

```
// određivanje aktivne dijete forme
Form aktivnoDijete = this.ActiveMdiChild;
```

Aktivna kontrola aktivne dijete forme sadržana je u `aktivnoDijete.ActiveControl`.

RAZVOJ PROGRAMSKIH RJEŠENJA

Rezime:

U okviru ovog poglavlja objašnjenja je uloga i vrste menija. Objasnjena je i razlika između SDI i MDI aplikacija. Dati su primjeri koji ilustriraju ove koncepte. Postoji još GUI kontrola koje nisu prikazane. Neke od njih kao naprimjer `TreeView`, `DataGridView` će se prikazati u narednim poglavljima a neke čitaoci će samostalno upoznati. Za korisnika programskog rješenja od izuzetne važnosti je raspored, oblik, boje i drugi dizajn faktori. Naredno poglavlje se bavi principima dobrog dizajn korisničkog interfejsa.

Pitanja i zadaci za samostalni rad:

1. Objasnite namjenu meni kontrole.
2. Objasnite ulogu i poziciju osnovne meni linije (meni bara).
3. Koje orientacije može imati meni bar?
4. Koje elemente (stavke) može imati meni?
5. Objasnite strukturu podmenija.
6. Objasnite strukturu kaskadnog menija.
7. Navedite neke preporuke za imenovanje stavki izbora u okviru menija.
8. Navedite neke preporuke oko organizacije menija.
9. Kako se na meniju formiraju grupe opcija?
10. Da li meniji mogu imati pomicne trake?
11. Šta se preporučuje da se postavi uz meni stavke za izbore koji se često koriste.
12. U okviru .NET koja kontrola se koriste za formiranje menija?
13. Koji je osnovni (*default*) događaj za meni kontrolu?
14. Navedite pet osobina .NET meni kontrole. Objasnite njihovu namjenu?
15. Objasnite šta je kontekstni meni?
16. Kako se aktiviraju kontekstni meniji?
17. Koja kontrola u okviru .NET se koristi za kreiranje kontekstnih menija?
18. Koja je namjena prozor menija?
19. Šta je dijalog-*box*?
20. Navedite često korištene dijalog *boxove*?
21. Prikažite primjerom korištenje menija, podmenija, kontekstnog menija, prozor menija i dijalog *boxa*.
22. Koja je razlika između MDI i SDI organizacije aplikacijskog prozora?
23. Kako se naziva glavni aplikacijski prozor u MDI organizaciji, a kako ostali prozori?
24. Koja .NET osobina je bitna ukoliko je potrebno da prozor postane glavni MDI prozor?

Poglavlje 9: Dizajniranje korisničkog interfejsa

U okviru ovog poglavlja razmatraju se faktori i daju se smjernice za dobro dizajniran korisnički interfejs kroz tematske jedinice:

1. Uloga i značaj dobro dizajniranog korisničkog interfejsa
2. Utjecaj ljudske psihologije na dizajn korisničkog interfejsa
3. Smjernice za izbor i raspoređivanje GUI kontrola

9.1. Uloga i značaj dobro dizajniranog korisničkog interfejsa

Dobro dizajniran korisnički interfejs je vrlo bitan korisnicima. Kroz interfejs korisnici uviđaju mogućnosti sistema. Za mnoge korisnike to predstavlja i sam sistem. Sam izgled i navigacija sistema su vrlo bitni i utječu na rad na mnogo načina. Ukoliko su zbumujući i neefikasni, ljudi će imati poteškoća prilikom obavljanja svakodnevnih poslova i praviti će više grešaka. Loš dizajn sistema može učiniti da korisnici više ne žele da koriste sistem i da ga odbace. Također može izazvati frustracije i stres, a finansijski gubici kompanije i korisnika mogu biti ogromni. Prosječni korisnik će prije odabratи sistem koji se lakše koristi, nego onaj koji je po svojoj funkcionalnosti možda i kvalitetniji. Obzirom da je korisnički interfejs važan segment jednog softverskog rješenja, potrebno je razviti dobar korisnički interfejs koji će pomagati korisnicima u izvršavanju njihovog rada. Sva razmatranja oko dizajna korisničkog interfejsa su ilustrirana na primjeru ogledne aplikacije Videoteka koja je rađena u sklopu [24].

9.2. Utjecaj ljudske psihologije na dizajn korisničkog interfejsa

Dobar dizajn počinje sa razumijevanjem ljudske psihologije kao fundamentalnog principa u dizajnu interfejsa. Mnogi autori vodiča za dizajn imali su barem malo osnovnog znanja o ljudskoj psihologiji koja je od velikog značaja za primjenu na dizajn kompjuterskih sistema. Bitno je razumjeti najvažnije aspekte ljudske psihologije koja je temelj vodiča za dizajn korisničkog interfejsa. U okviru ove sekcije opisane su osnovne psihološke karakteristike ljudi koje imaju primjenu u dizajnu korisničkog interfejsa: percepcija, pamćenje, učenje i objašnjen je način utjecaja njihovih karakteristika na dizajn korisničkog interfejsa. U okviru opisa karakteristika percepcije objašnjeni su Gestalt principi i prikazana je njihova primjena.

RAZVOJ PROGRAMSKIH RJEŠENJA

9.2.1. Percepcija

Percepcija je ljudska svijest i razumijevanje elemenata i objekata u okruženju kroz fizički osjećaj čula, uključujući vid, sluh, miris itd.

Ljudska percepcije ovisi od tri faktora: iskustva korisnika, trenutnog konteksta, ciljeva i planova korisnika.

Ovi faktori utiču i na sam dizajn. Naprimjer, kada korisnik nešto traži ukoliko to što traži je na različitom mjestu od uobičajnog ili izgleda drugačije, postoji mogućnost da pogriješi. Dovoljno je pri navigaciji između prozora promijeniti položaj dugmeta za prethodni i sljedeći prozor da se korisnik zbuni. Zbog toga, potrebno je prikazivati informacije i kontrole na konzistentnim lokacijama. Kontrole i prikaz podataka koji rade istu funkciju trebaju biti postavljene na istoj poziciji na različitim stranicama i formama korištenjem iste boje, fonta i sjene. Korisnici dolaze sa ciljevima koje žele postići. Dizajneri bi trebali razumjeti ove ciljeve. Oni utječu na percepciju – bitno je da su potrebne informacije dostupne i da se mapiraju na cilj korisnika, tako da ih oni primjećuju i koriste.

9.2.2. Gestalt principi

Početkom 20. stoljeća grupa njemačkih psihologa željela je da objasni kako funkcioniра ljudska percepcija. Primijetili su mnoge bitne vizuelne fenomene. Ljudski vizuelni sistem nameće strukturu i spoznaje cijele oblike, figure i objekte umjesto rastavljenih ivica, linija i područja. Njemačka riječ za oblik je *Gestalt*, tako da su ove teorije postale poznate pod nazivom Gestalt principi vizuelne percepcije. Gestalt principi opisuju kako um organizira vizuelne podatke.

Gestalt principi su još uvijek na snazi,ako ne kao temeljno objašnjenje ljudske percepcije, barem kao okvir koji to opisuje. Oni također pružaju korisni temelj za smjernice za grafički dizajn i korisnički interfejs. Najvažniji Gestalt principi koji imaju implikaciju za dizajn korisničkog interfejsa su: udaljenost, sličnost, simetrija, slika/pozadina.

Gestalt princip: Udaljenost

Relativna udaljenost između objekata na ekranu utječe na korisničko perceptivno opažanje da li i kako su objekti organizirani u podgrupe. Objekti koji su blizu jedni drugima (relativno u odnosu na druge objekte) izgledaju kao grupa, za razliku od onih koji su udaljeni.

Prema principu udaljenosti, elementi na ekranu mogu biti vizuelno grupirani postavljanjem bliže jedni drugima nego drugim kontrolama, čak i bez vidljivih granica. Mnogi eksperti u grafičkom dizajnu preporučuju ovaj pristup da bi se izbjegla vizuelna pretrpanost korisničkog interfejsa.

Na slici 9.1 označeno sa brojem 1 prikazano je vizuelno grupiranje dugmadi za akcije dodavanja i brisanja bez vidljivih granica u aplikaciji Videoteka.

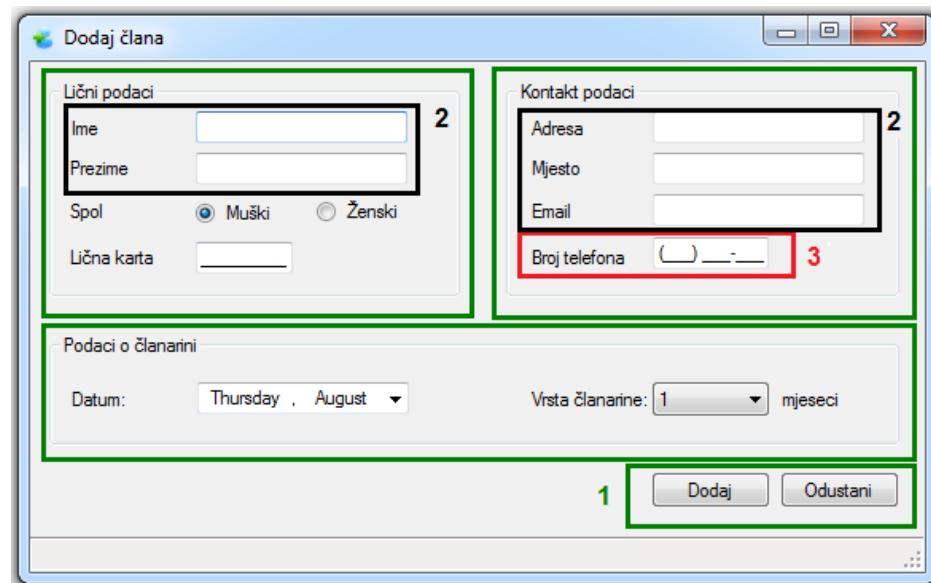
RAZVOJ PROGRAMSKIH RJEŠENJA

Gestalt princip: Sličnost

Ljudske oči i um vide objekte kao pripadnike iste grupe ukoliko dijele istu vizuelnu osobinu, kao što je boja, veličina, oblik, svjetlost i orijentacija. Na slici 9.1 označeno sa 2 prikazana je upotreba principa sličnosti u aplikaciji Videoteka. Tekstualna polja imaju istu širinu i visinu.

Gestalt princip: Simetrija

Princip simetrije tvrdi da ljudi pokušavaju da analiziraju i rastavljaju kompleksne scene na način koji smanjuje kompleksnost. Podaci obično imaju više načina interpretacije, ali naš vid automatski organizira i interpretira podatke tako što ih pojednostavljuje i daje im simetriju. U aplikaciji Videoteka ovaj princip prikazan je prilikom unosa broja telefona novog člana. Iako su zgrade bliže drugim znakovima, zbog principa sličnosti ljudi ih uglavnom vide kao par simetričnih zagrada, što je prikazano na slici 9.1 označeno sa 3. S obzirom da je ovim poljem predstavljen format unosa telefonskog broja, korištenje ovog strukturiranog polja olakšava korisniku unos.



Slika 9.1: Prikaz principa udaljenost, sličnost i simetrija

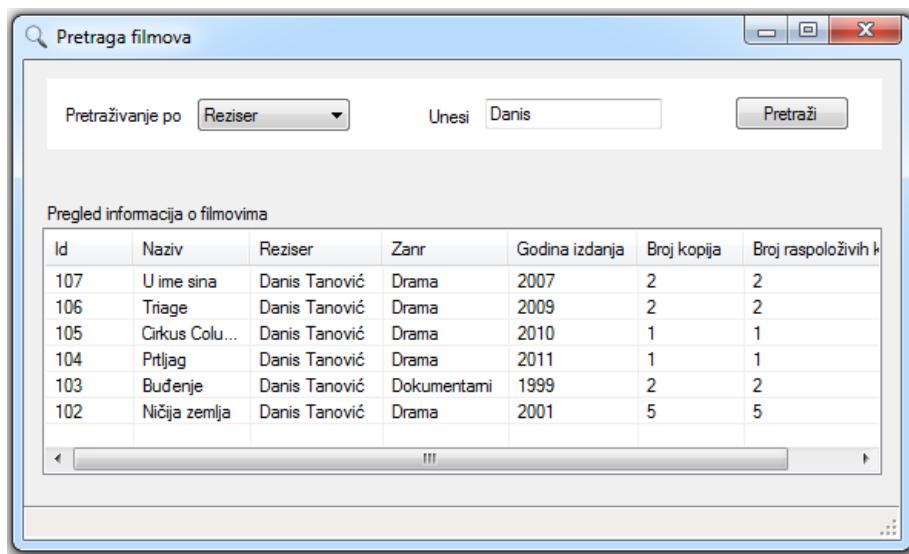
Gestalt princip: Figura/pozadina

Ljudski um razdvaja vizuelno polje u figuru (prednji plan) i pozadinu (zadnji plan). Prednji plan se sastoji od elemenata scene koji su objekat primarne ljudske pažnje, a pozadinu sačinjava sve ostalo. Na slici 9.1 prikazana je primjena principa figura/pozadina u aplikaciji Videoteka prilikom prikaza informacija o sistemu. Tekst na sivoj pozadini postaje prednji plan i centar pažnje.

9.2.3. Pamćenje

Prilikom dizajniranja raznih pretraga i pregledavanja rezultata treba imati u vidu osobine vezane za kratkoročno pamćenje. Broj odvojenih informacija koje se obrađuju u kratkoročnom pamćenju je ograničen na 3 do 4 predmeta, a zadržavanje informacija je vrlo kratko i smatra se da traje 5 do 30 sekundi. Kada korisnici žele da izvrše pretragu oni upišu željeni pojam, započnu pretragu i pregledaju rezultate. Zbog ograničenja kratkoročnog pamćenja, pri pojavi rezultata pažnja korisnika se preusmjerava od onoga što su upisali kao pojam pretrage prema rezultatima, što često dovodi do toga da korisnici prilikom pregledanja rezultata zaborave upisani pojam za pretraživanje. Zbog toga je bitno da se uneseni pojmovi ne brišu prilikom prikazivanja rezultata.

U aplikaciji Videoteka upisani pojmovi pretrage i izabrane opcije su prikazani i nakon završene pretrage i prikaza rezultata, što je prikazano na slici 9.2.



Slika 9.2. Pretraga filmova u aplikaciji Videoteka

Također, karakteristike kratkoročnog pamćenja imaju implikaciju i u dizajniranju uputa za izvršavanje određenog zadatka. Interaktivni sistemi trebali bi prikazivati informacije ili upute tako da ih korisnici prate dok ne izvrše zadatak. Zbog kratkoročnog pamćenja nije moguće zapamtiti sve upute, što bi dizajneri trebali imati u vidu.

Učenje i pamćenje su dva nadopunjajuća faktora prilikom procesa učenja. Učenje je proces prijenosa informacija iz kratkoročnog pamćenja u dugoročno.

Korisnici ispočetka kada izvode neku aktivnost, to rade sa oprezom. Poslije u praksi taj proces bio trebao da postane automatiziran. Da bi se ovo postiglo, potrebno je poznavati faktore koji utječu na učenje i usvajanje korištenja interaktivnih sistema. Dosljednost u dizajnu sistema je vrlo bitna i utječe na učenje. Jednom naučene akcije mogu se primjenjivati na sličnim zahtjevima, što omogućava brže i jednostavnije učenje.

RAZVOJ PROGRAMSKIH RJEŠENJA

Naprimjer, u aplikaciji Videoteka, zbog osobine dosljednosti sistema jednom naučeno pretraživanje članova videoteke može biti primijenjeno na pretraživanje filmova.

Mnogi dizajneri korisničkog interfejsa počinju dizajn sistema planirajući izgled, što nije ispravan pristup. Ovaj pristup rezultira sistemom koji ne sadrži bitne funkcionalnosti, sadrži bespotrebne funkcionalnosti i težak je za korištenje i učenje. Preporučuje se da dizajneri najprije odrede karakteristike korisnika, koncepte samog sistema i veze između njih, pa tek onda dizajniraju kako će se ti koncepte grafički predstaviti korisniku.

9.3. Smjernice za izbor i raspoređivanje GUI kontrola

U prethodnim poglavljima obrađene su osnovne GUI komponente. U ovoj sekciji slijedi rezime o tim komponentama sa preciziranim smjernicama o njihovoj upotrebi.

9.3.1. Izbor kontrola

Meni

Potrebno je obezbijediti glavni meni koji je početni i sadrži osnovne opcije sistema, čime se obezbeđuje da sve sistemske akcije počinju njegovom upotrebotom.

Broj izbora menija koji se prikazuje potrebno je ograničiti. Ukoliko elementi nisu logički grupirani, potrebno je ograničiti izbore do 4, maksimalno do 8. Ukoliko se radi o logički grupiranim elementima, potrebno je ograničiti izbore do 18, maksimalno do 24.

Stavke menija bi trebale biti logički grupirane u grupe koje su međusobno isključive i različite. Broj grupa koje su prikazane na ekranu treba ograničiti na prezentaciju do 6 ili 7 grupa.

Ubrzanja sa tastature

Ubrzanja sa tastature omogućavaju izvođenje akcija, pored korištenja miša, korištenjem i tastature. Potrebno ih je obezbijediti za često korištene akcije, čime se olakšava rad korisnicima. Prilikom kreiranja ubrzanja sa tastature potrebno je koristiti do dvije tipke. Ukoliko dvije tipke trebaju biti pritisnute u isto vrijeme, potrebno je koristiti znak (+) koji to naznačava. Opise ubrzanja sa tastature potrebno je poravnati desno. Za mnoge česte izbore menija utvrđena su standardna ekvivalentna slova koja se trebaju koristiti.

U aplikaciji Videoteka početni meni sadrži sve opcije koje sistem nudi korisnicima i prikazan je na slici 9.3. Njegovom upotrebotom počinju sve sistemske akcije, a broj izbora ograničen je na 3 menija čije su stavke grupirane u međusobno isključive i različite opcije. Na slici 9.3 prikazane su strukture menija koji su korišteni u aplikaciji Videoteka zajedno sa njihovim ekvivalentima sa tastature.

RAZVOJ PROGRAMSKIH RJEŠENJA



Slika 9.3: Korištenje ubrzanja sa tastature u aplikaciji Videoteka

Radio dugmad koriste se za selektiranje jedne opcije od malog izbora međusobno isključivih opcija (2-8). Korisnici vrlo lako mogu izabrati željenu opciju ukoliko se koriste ove kontrole. Preporučuje se da radio dugmad imaju selektiranu početnu vrijednost.

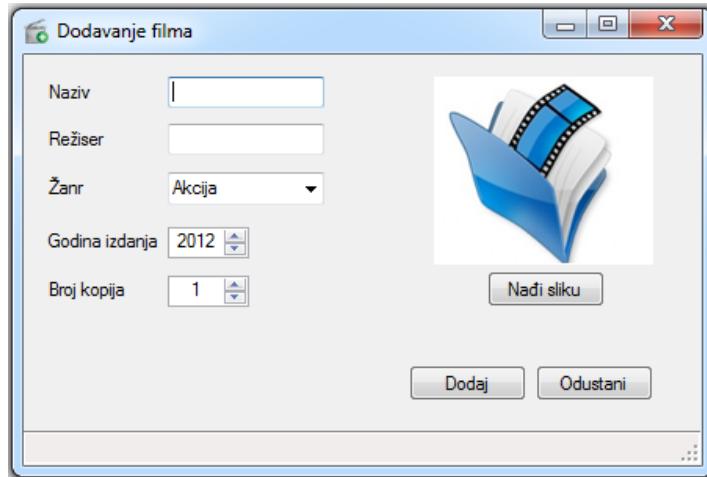
Check box kontrole koriste se za postavljanje jedne ili više opcija na uključeno i isključeno. Koriste se kod opcija koje nisu međusobno isključive, što znači da može biti odabранo više od jedne opcije. Često se koriste za postavljanje atributa i vrijednosti koje utječu na druge kontrole. Česta greška prilikom korištenja ove kontrole je njena upotreba za međusobno isključive opcije.

Tekstualna polja uglavnom sadrže tekst koji je unesen ili izmijenjen koristeći tastaturu. Najčešće se koristi kad je opseg unosa neograničen ili težak za kategorizaciju i varijabilne je dužine. Ova kontrola je obično alternativa kada nije moguće kreiranje liste selekcije.

Tekstualna polja se pogrešno koriste ukoliko se radi o strukturiranom ili segmentiranom ulazu koji ima svoj format, kao što je npr. unos broja telefona. U ove svrhe potrebno je koristiti strukturirana posebno prilagođena tekstualna polja. Na slici 9.1 prikazano je strukturirano polje za unos telefona.

U aplikaciji Videoteka se za unos godine izdanja i broja kopija filmova koristi kontrola koja ograničava unos na numeričke vrijednosti i olakšava korisniku unos podatka.

Za podatke koji su kategorizirani, kao što je žanr filma, koristi se lista u kojoj su prikazani mogući izbori žanrova. Ekran za unos novog filma prikazan je na slici 9.4.



Slika 9.4: Dodavanje novog filma u aplikaciji Videoteka

Tabovi (engl. tab) su prozori koji sadrže kartice koje kreiraju stranice ili odvojene sekcije. Koriste se za prikaz informacija koje se mogu logički organizirati u stranice ili sekcije unutar istog prozora. Najkorisniji su za prikaz izbora koji mogu biti primjenjeni na jedan objekat. Trebaju biti opisani kratkim opisnim nazivom koji identificira njihov sadržaj. Ne treba pretjerivati sa brojem tabova.

Kontrola za grupiranje je okvir koji okružuje grupu kontrola. Može sadržavati naslov (*group box*) u gornjem lijevom uglu i koristi se da vizuelno poveže elemente koji su funkcionalno povezani (2 ili više elementa). Ukoliko se koristi naslov, potrebno je da on bude kratak, sastavljen od jedne do dvije riječi. Česta greška prilikom upotrebe ove kontrole je njen korištenje za samo jedno podešavanje. Ponekad se ova kontrola pogrešno koristi i za uređivanje ivica prozora. Također, česta greška je korištenje kontrole za grupiranje unutar druge kontrole za grupiranje, čime se stvara pretrpan ekran.

Specijalno dizajnirane kontrole za unos podataka

Specijalne kontrole dizajnirane za unos specifičnih podataka koje omogućavaju segmentiranje podataka poboljšavaju strukturu unosa. Naprimjer, umjesto da se koristi tekst polje dizajneri mogu sami dizajnirati specifično polje za prikaz (unos) datuma, e-mail adrese, kreditne kartice, telefonskog broja. Specijalno dizajniranim kontrolama moguće je dobiti efektivnu vizuelnu strukturu.

RAZVOJ PROGRAMSKIH RJEŠENJA

9.3.2. Pravilno raspoređivanje kontrola

Pored ispravnog korištenja kontrola potrebno je i pravilno ih raspoređivati. Prilikom grupiranja kontrola, kao što su npr. radio dugmad, bitno je da povezana radio dugmad budu bliže jedna drugima nego drugim kontrolama. Ovo proizlazi iz principa udaljenosti, koji je prethodno objašnjen, po kojem kontrole koje su međusobno bliže opažaju se kao grupa. Dizajneri često prave grešku postavljajući ove kontrole previše daleko. Moguće ih je grupirati korištenjem praznog prostora, specijalnih kontrola ili korištenjem separatora (npr. horizontalna linija između dvije različite grupe kontrola).

Također, prilikom postavljanja labela uz kontrole potrebno je voditi računa o tome da tekst labele (natpis) uz kontrolu nije previše udaljen od kontrole koja se koriste za unos podataka. Također, tekst labele ne bi trebao biti bliži drugoj kontroli nego onoj na koju se odnosi.

Naprimjer, na slici 9.5 je prikazano unošenje uvjeta pretrage članova u aplikaciji Videoteka. Natpisi su najbliži kontrolama na koje se odnose, što ih čini vizuelno grupiranim.

The screenshot shows a search interface with the following elements: 'Pretraživanje po' (Search by) dropdown set to 'Prezime', an input field labeled 'Unesi' (Enter), and a 'Pretraži' (Search) button. The labels are positioned very close to the controls they describe, illustrating good labeling practices.

Slika 9.5: Tekst labele uz kontrolu u aplikaciji Videoteka

Ukoliko se koriste dugačak tekst uz kontrolu, to može predstavljati problem. Takvi tekstovi mogu se razdvojiti u više labela, tako da prilikom poravnjavanja ne budu previše udaljeni od kontrole na koje se odnose. Također, duge natpise je moguće staviti iznad kontrole. Ovo je primijenjeno u aplikaciji Videoteka gdje su dugi natpisi prilikom iznajmljivanja stavljeni iznad kontrole za unos (slika 9.6).



Slika 9.6: Korištenje dugih natpisa u aplikaciji Videoteka

9.4. Odabir boja i ikona

Poznavanje dobre prakse pri izboru boja je vrlo bitno u procesu dizajna korisničkog interfejsa. Ukoliko su korištene pravilno, boje mogu naglasiti logičku organizaciju informacija i učiniti prikaz sistema zanimljivijim i privlačnijim korisniku. Ipak, ukoliko se ne koriste pravilno, mogu odvlačiti pažnju korisniku i umarati ga.

Iako nije eksperimentalno potvrđeno, iskustvo pokazuje da bi se trebale koristiti maksimalno četiri boje istovremeno na tekstualnim ekranima. Čak se preporučuje i pristup korištenja dvije ili tri boje. Općenito se preporučuje korištenje boja koje se lako razlikuju. Boje koje su najlakše za prepoznavanje su: crvena, zelena, žuta, plava, crna i bijela.

Preporučuje se izbjegavanje korištenja parova boja koje daltonisti ne mogu razlikovati, kao što je npr. par tamnocrvene i crne boje.

9.4.1. Biranje boja

Prilikom izbora kombinacije boja prvog plana (*front*) i pozadine (*background*) najbolje je prvo izabrati boju pozadine. Nakon toga preporučuje se biranje odgovarajuće boje prvog plana. Prilikom izbora pozadine preporučuju se jednobojne pozadine i smatra se da su bolji izbor od pozadine sa uzorcima i teksturama koje otežavaju čitanje teksta.

Prilikom izbora boje pozadine i prednjeg plana potrebno je birati boje visokog kontrasta. Kombinacija boja koja se najviše preporučuje je korištenje crne boje teksta na svjetlo sivoj ili bijeloj pozadini.

Indikacija statusa

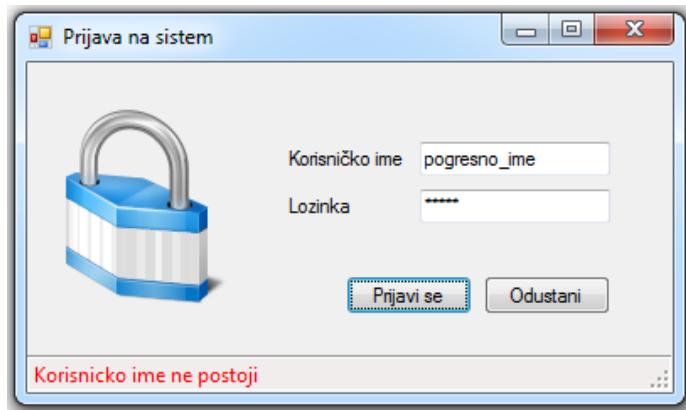
Boje se također mogu koristiti za indikaciju statusa u sistemu. Pravilno korištenje je prikazano u tabeli 9.1.

Status	Boje
Ispravan, normalan ili uredu	Zelena, bijela, plava
Oprez	Žuta ili zlatna
Vanredno stanje ili neispravnost	Crvena

Tabela 9.1. Korištenje boja za indikaciju statusa

RAZVOJ PROGRAMSKIH RJEŠENJA

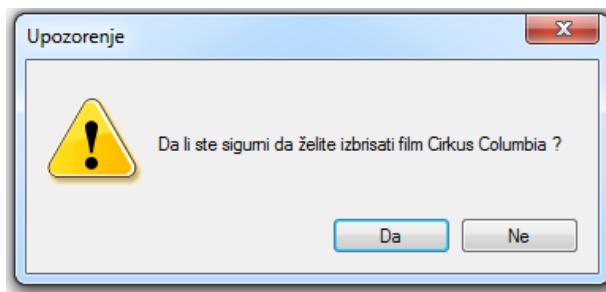
U aplikaciji Videoteka, boje su korištene za prikaz statusa prilikom logiranja na sistem. Ukoliko se korisnik pokuša logirati sa pogrešnim korisničkim imenom, poruka o neispravnosti će biti napisana crvenom bojom, što je i prikazano na slici 9.7.



Slika 9.7: Korištenje crvene boje za indikaciju neispravnog statusa u aplikaciji Videoteka

Ukoliko se korisnik pokuša logirati sa ispravnim korisničkim imenom i lozinkom, poruka o ispravnom statusu i uspješnoj prijavi na sistem će biti napisana zelenom bojom na statusnoj liniji. U sljedećem poglavlju će biti detalji vezani za implementaciju poruka na statusnoj liniji.

Prilikom brisanja podataka o filmu, za prikaza poruke upozorenja korištena je žuta boja ikone upozorenja (slika 9.8).



Slika 9.8: Korištenje žute boje za indikaciju upozorenja u aplikaciji Videoteka

9.4.2 Smjernice za korištenje ikona

Grafika ekrana, ukoliko se koristi pravilno, može predstavljati snažnu komunikacijsku tehniku, kao i tehniku za privlačenje pažnje.

Ikone su slike koje se koriste da prikažu objekte i akcije kojima korisnici mogu manipulirati. Dizajn ikona je važan proces. Smislene i ikone koje su luke za prepoznavanje mogu olakšati proces učenja, a loš dizajn ikona može voditi ka greškama i zbumjenosti korisnika.

Fowler i Stanwick (1995. godine) dali su generalne smjernice o karakteristikama uspješne ikone. Uspješna ikona je ona koja:

- Izgleda drugačije od drugih ikona
- Očito je šta predstavlja
- Prepoznatljiva je
- Izgleda dobro crno-bijela isto kao i u boji

Kada je to moguće, potrebno je koristiti ikone koje već postoje. Mnoge standardne ikone su već razvijene za grafičke sisteme.

Ipak, ukoliko se kreiraju ikone, potrebno je da one budu vizuelno različite i jasno prikazuju objekte. Prilikom dizajna ikona potrebno je zadržati jednostavnost, što olakšava prepoznavanje ikona na ekranu.

Neke od ikone koje se koriste u aplikaciji Videoteka su sljedeće:

-  Ikona za prikaz člana videoteke,
-  Ikona za dodavanje člana videoteke,
-  Ikona za pretragu,
-  Ikona za izmjenu podataka,
-  Ikona za dodavanje filma,
-  Ikona za iznajmljivanje filma,
-  Ikona za prikaz pomoći,
-  Ikona za informacije o sistemu.

9.5. Prikaz informacija za čitanje

Neki od parametara koji dovode do otežanog čitanja su oblik teksta, oblik fonta, veličina teksta, količina teksta, rječnik, način vizuelne hijerarhije informacija koje se predstavljaju.

Veličina fonta

Ako je tekst napisan velikim slovima težak je za čitanje. Također, tekst je težak za čitanje ako su fontovi premali za vizuelni sistem čitaoca. Razvojni inženjeri ponekad koriste fontove sa malom veličinom jer žele da prikažu puno teksta na malom području displeja. Međutim, korisnici ne mogu čitati taj tekst ili ga čitaju teško.

Tekst na ne odgovarajućoj pozadini

Vizuelna pretrpanost (buka) oko i ispod teksta može poremetiti prepoznavanje karaktera i riječi. Vizuelna 'buka' je često rezultat smještanja teksta na pozadini sa paternom ili prikaza teksta u bojama koji su u nedovoljnem kontrastu u odnosu na pozadinu. Postoje situacije u kojima se namjerno želi učiniti tekst teškim za čitanje. Naprimjer, uobičajna mjera sigurnosti na Webu je da se pitaju korisnici da identificiraju iskrivljene riječi.

Forma na pozadini sa paternom

Pozadina ne treba da bude previše uočljiva da ometa čitanje teksta.

Količina i centriranje teksta

Jedan od aspekata brzog čitanja je pomjeranje očiju. U brzom (automatskom) čitanju, ljudske oči se treniraju da idu nazad na istu horizontalnu poziciju i dolje jednu liniju. Ako je tekst centriran ili desno poravnat svaka linija teksta počinje na različitoj horizontalnoj poziciji što otežava čitanje. Mnogi korisnički interfejsi prikazuju previše teksta i traže od korisnika da čita više nego što je potrebno.

Rječnik

Još jedan razlog koji otežava čitanje je korištenje nefamilijarnog rječnika-riječi koje korisnik ne zna dovoljno dobro, upotreba tehničkih termina, žargona, kao i dvosmislena formulacija rečenica.

Vizuelna hijerarhija pomaže ljudima fokusiranje na relevantne informacije

Jedan od najvažnijih ciljeva u strukturiranju prezentacije informacija je da se omogući vizuelna hijerarhija što se može postići sa:

- Podjelom informacija u odvojene sekcije, i podjelom velikih sekcija u podsekcije.
- Označavanjem svake sekcije i podsekcije istaknuto na takav način da jasno identificira sadržaj.
- Prezentiranjem sekcija i podsekcija hijerarhijski, i naglašavanjem viših sekcija u odnosu na niže.

Vizuelna hijerarhija dozvoljava korisnicima, kada skeniraju informacije, da odvoje relevantno od nerelevantnog u ovisnosti od njihovih ciljeva, i da fokusiraju svoju pažnju na relevantne informacije. Vizuelna hijerarhija je isto važna na interaktivnim kontrolnim panelima i formama.

9.5. Odziv sistema i pomoć korisniku

Korisniku kompjuterskog sistema treba obezbijediti i interaktivnost, koja se ogleda u obavještavanju sistema o uspješnosti obavljanja nekih akcija, pomoći za obavljanje akcija, upravljanju greškama.

9.5.1. Odgovor sistema na korisničke akcije

Potrebno je da sistem obezbijedi ispravnu reakciju na akcije korisnika. Efektivan sistem ove reakcije omogućava u prihvatljivom vremenskom periodu, a korisniku odmah daje do znanja da je njegova akcija prihvaćena. Ukoliko to nije slučaj, sistem izaziva nezadovoljstvo kod korisnika. Mnoge studije su se bavile pitanjem idealnog vremena odgovora sistema. Ovo vrijeme odgovora ovisi od očekivanja korisnika, zadacima koji se izvršavaju, kao i situacijama. Generalno vrijeme odgovora sistema je previše dugo za korisnika ukoliko primijeti da akcija traje dugo.

Bavljenje vremenskim kašnjenjima sistema uključuje obavještavanje korisnika da sistem radi i da će postupak biti obavljen u okviru predvidljivog vremenskog intervala.

Ukoliko se radi o vremenskim kašnjenjima do 10 sekundi, potrebno je prikazati signal zauzetosti do završetka operacije (npr. pokazivač u obliku pješčanog sata). Za duže vrijeme čekanja, potrebno je prikazati traku napretka ili procijenjeno vrijeme čekanja.

RAZVOJ PROGRAMSKIH RJEŠENJA

9.5.2. Pomoć korisniku

Svaki dobro dizajniran sistem treba da pruži korisniku pomoć za njegovo korištenje.

Kontekstualna pomoć

Kontekstualna pomoć pruža informacije koje se odnose na zadatak koji se izvršava. Vrste kontekstualne pomoći su: komandna dugmad za pomoć, statusne poruke, specifični opisi (*tooltips*), dugme "Šta je ovo?" ("What's this?" buttons).

U tabeli 9.2. prikazani su načini korištenja prethodno nabrojanih vrsta kontekstualne pomoći.

Kontekstualna pomoć	Način korištenja
Komandno dugme za pomoć	Svrha ove kontrole je da ponudi pomoć ili informacije o namjeni ili sadržaju prozora koji je prikidan. Ovu vrstu pomoći potrebno je predstaviti u sekundarnom prozoru ili dijalog <i>boxu</i> .
Statusna poruka	Statusna poruka je poruka povezana sa predmetom na ekranu na koji je korisnik fokusiran i njena svrha je da ponudi dodatne informacije o objektu.
Specifični opisi (tooltips)	<i>Tooltips</i> je mali prozor koji se pojavljuje u okolini kontrole kada se pokazivač zadrži na kontroli određeni period vremena. Može biti korištena za opis kontrole ukoliko ona nije opisana labelom ili za dodatni opis ili statusnu informaciju o kontroli.
"Šta je ovo?"	Ova komanda može biti locirana na meniju primarnog prozora, može biti predstavljena kao dugme na traci alata ili komanda na <i>pop-up</i> meniju. Koristi se da obezbijedi informacije o bilo kojem objektu na ekranu.

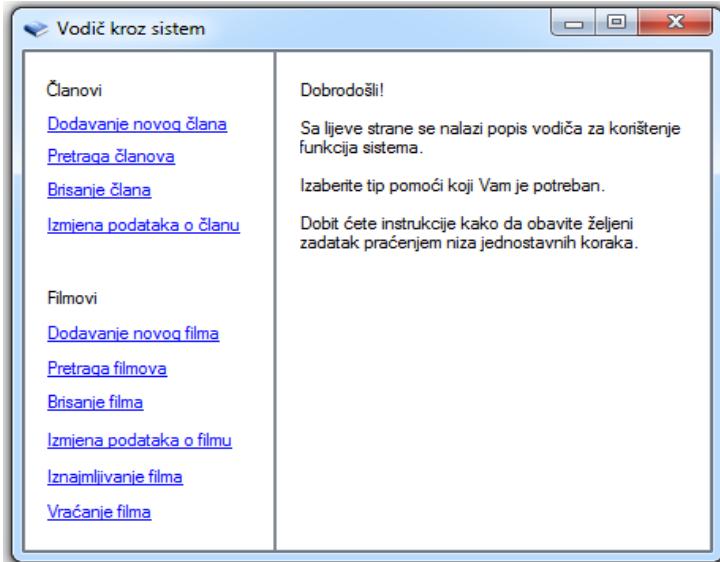
Tabela 9.2: Vrste kontekstualne pomoći

Pomoć orientirana prema zadatku

Cilj ove vrste pomoći je da opiše proceduralne korake potrebne za uspješno završavanje korisnikovog zadatka. Ovoj vrsti pomoći se pristupa kroz preglednik tema za pomoć. Ovaj prozor bi trebao zauzimati minimalno prostora na ekranu, ali ipak biti dovoljno velik da prezentuje sve informacije. Koristi se za opis izvršavanja zadataka. Potrebno je da izvršavanje zadataka bude opisano na jednostavan i precizan način.

RAZVOJ PROGRAMSKIH RJEŠENJA

Na slici 9.9 prikazan je ekran vodiča kroz sistem Videoteka, koji korisnicima nudi pomoć prilikom izvršavanja funkcionalnosti.



Slika 9.9: Vodič kroz sistem u aplikaciji Videoteka

9.5.3. Poruke

Veoma je bitno da poruke koje se daju korisnicima budu uočljive. Sve na ekranu što nije unutar 1-2 centimetra od klik lokacije je u viziji okoline, gdje je rezolucija niska, što objašnjava zašto korisnici ne uoče neke poruke o grešci.

Postoji više dobro-poznatih metoda za osiguranje vidljivosti poruka:

-Staviti ih gdje korisnici gledaju. Ljudski fokus je predikativan prilikom interakcije sa GUI-om, i ta mjesta dizajneri mogu koristiti za smještanje poruka. Npr. kada korisnik klikne na dugme ili link, obično gleda direktno na njega nekoliko trenutaka poslije.

- Označiti poruku: Poruka treba da bude istaknuta i da jasno indicira da je nešto pogrešno.
- Koristiti simbole za greške (za informativne poruke, poruke upozorenja, fatalne poruke...).

-Koristiti crvenu boju za greške: Po konvenciji u interaktivnim kompjuterskim sistemima crvena boja označava alert, opasnost, problem, grešku. Korištenje crvene za bilo koju drugu informaciju može dovesti do pogrešne interpretacije.

Ako je npr. crvena boja dominantna za domenu problema za koju se razvija tada koristiti drugu boju za obilježavanje grešaka, markirati poruke sa simbolom ili koristiti strožije metode za

RAZVOJ PROGRAMSKIH RJEŠENJA

greške. O vrstama grešaka, preporukama za njihovo obilježavanje, implementaciji razmatra se i u okviru sljedećeg poglavlja.

-Kada interfejs proizvede zvuk, to je znak korisniku da se nešto desilo što zahtjeva akciju. Ljudske oči refleksivno skeniraju ekran da nađu šta je uzrokovalo zvuk. To može omogućiti korisniku da uoči poruke o greškama koje su postavljene i na ne standardnim mjestima. Međutim, ako se ista aplikacija koristi od strane više korisnika u nekom zajedničkom prostoru poruke sa zvukom mogu uznemiravati i usporavati rad više korisnika. Zbog toga poruke sa zvukom treba ostaviti samo za korištenje u veoma specijalnim situacijama.

-Ljudska percepcija dobro detektira pokretne objekte, zbog toga dizajneri korisničkog interfejsa koriste *flash* poruke ili poruke (objekte) koje se pomjeraju. Međutim, iskusni korisnici smatraju da *flash* i pomjerajući objekti su dosadni, utiču na radnu aktivnost, a mnogi ih ignoriraju. Zbog toga, ako se koriste poruke koje se pomjeraju ili blinkaju, to treba da bude najduže pola sekunde sa jasnim sadržajem.

9.6. Testiranje upotrebljivosti sistema

Upotrebljivost (*usability*) sistema je jedna od najbitnijih karakteristika softvera. Upotrebljivost softvera je kvalitativni atribut koji procjenjuje koliko je korisnički interfejs jednostavan i lak za korištenje.

Upotrebljiv sistem je onaj koji je:

- | | |
|-------------------------------|--|
| Efektivan: | Sistem omogućava izvršavanje zadataka korisnika potpuno i tačno. |
| Efikasan: | Sistem omogućava izvršavanje zadataka brzo i tačno. |
| Privlačan: | Sistem čiji stil čini produkt ugodnim i zadovoljavajućim za korištenje. |
| Tolerantan: | Sistem sprječava greške i pomaže pri oporavku od grešaka ukoliko se one ipak dese. |
| Jednostavan za učenje: | Sistem podržava i potiče korisnike na učenje i razumijevanje njegovih mogućnosti. |

Primjenom gore opisanih principa može se postići upotrebljiv sistem. Testiranje da li je zaista sistem upotrebljiv je složen proces i način njegove izvedbe je izvan vidokruga ovog rukopisa.

RAZVOJ PROGRAMSKIH RJEŠENJA

Rezime

Kroz ovo poglavlje su objašnjene ljudske psihološke karakteristike koje imaju primjenu u dizajnu korisničkog interfejsa: percepcija, pamćenje i učenje. Dalje u okviru poglavlja su date smjernice za pravilno korištenje kontrola i njihovo raspoređivanje, te smjernice za korištenje boja, ikona i fontova. Također, date su smjernice za pravilan odgovor sistema na korisničke akcije, pružanje pomoći korisniku, kao i smjernice za način upravljanja greškama. Na samom kraju rada dat je uvod u testiranje upotrebljivosti sistema.

Upravljanje greškama i validacija podataka je od izuzetne važnosti za svako programsko rješenje i to se razmatra u okviru sljedećeg poglavlja.

Pitanja i zadaci za samostalni rad:

1. Navedite značajnija dijela vezana za dizajn korisničkog interfejsa.
2. Navedite psihološke aspekte bitne za dizajn korisničkog interfejsa.
3. Navedite Gestalt principe vizuelne percepcije i objasnite njihovu povezanost sa dizajnom korisničkog interfejsa.
4. Obrazložiti zašto je strukturirani i hijerarhijski način prikaza informacija bolji od nestrukturiranog načina.
5. Navedite značaj i primjer neke specifične kontrole za unos podataka.
6. Koje osobine teksta su bitne da bi tekst bio čitljiv?
7. Navedite smjernice za korištenje boja.
8. Navedite smjernice za korištenje ikona.
9. Navedite preporuke oko pisanja i načina prikazivanja poruka.
10. Navedite preporuke za dizajn pretraga i prikazivanje rezultata.
11. Dizajnirati kontrolu za unos e-mail adrese za domenu etf.unsa.ba.
12. Dizajnirati posebnu vizuelno interesantnu kontrolu za prikaz trenutnog stanja o vremenu (temperatura, vlažnost,...).

Poglavlje 10: Validacija podataka i upravljanje izuzecima

U okviru ovog poglavlja opisana je važnost i mehanizmi validacije podataka i upravljanja izuzecima kroz sljedeće tematske jedinice:

1. Validacija podataka
2. Upravljanje izuzecima

10.1. Validacija podataka

U prethodnim poglavljima objašnjeno je kako se kreiraju Windows Forms aplikacije sa različitim kontrolama za unos i prikaz podataka. Implementirani su i mnogi događaji vezani za kontrole. U okviru koda programskog rješenja često su potrebne i dodatne provjere odnosno validacije ulaznih podataka da bi se osigurala tačnost podataka i smanjio broj ulaznih grešaka. Predmet ovog poglavlja je validacija ulaznih podataka unesenih od strane korisnika.

10.1.1. Validacija ulaznih podataka sa jedne kontrole

Korisnik najčešće unosi podatke preko `TextBox` kontrole čime mu je omogućen proizvoljan unos podataka. Mnoge ulazne greške mogu se otkriti sljedećim provjerama:

- Provjerom tipa podatka: Potrebno je provjeriti tip ulaznog podatka i osigurati da su ulazni karakteri konzistentni sa željenim tipom podatka.
- Provjerom ranga vrijednosti: Potrebno je verifikovati da su ulazni podaci u odgovarajućem rangu vrijednosti. Često postoje unaprijed definirani rangovi za neke vrijednosti.
- Provjerom dužine: Često je unos string karaktera koji mora sadržavati tačno određen broj karaktera. ZIP kod, identifikacijski broj, broj telefona su primjeri ulaznih stringova sa unaprijed poznatom dužinom.

Provjera tipa podataka

Najčešća ulazna greška se dešava uslijed unosa podatka neodgovarajućeg tipa. Npr., program za obradu neke kupovine, može zahtijevati unos broja artikala za kupovinu. Korisnik može pogriješiti i ukucati umjesto broja 10 string 10. Dobar programer anticipira takve greške dodavanjem validacijskog koda u program. Validacija korektnog formata odnosno tipa ulazne vrijednosti može se postići sa `TryParse()` metodom prikazanom sa kodom 10.1.

```
bool flag;
int kolicina;
flag = int.TryParse(kolicinaTextBox1.Text, out kolicina);
if (flag == false)
{
    MessageBox.Show("Dozvoljen je unos samo cifara","Ulagna greška");
    kolicinaTextBox1.Focus();
    return;
}
```

Kod 10.1: Provjera tipa podataka sa TryParse metodom

Provjere ranga vrijednosti i dužine ulaznog stringa mogu se jednostavno implementirati primjenom osnovnih programskih iskaza.

Ograničavanje korisnikovog ulaza

Mnogi pogrešni unosi se mogu izbjegći ako se umjesto teksta polje kontrole koriste kontrole koje ograničavaju unos kao npr. liste izbora, radio dugmad, *check* i *combo boxovi*, kontrole za kontrolirani unos numeričkih podataka. Prezentiranjem limitirane liste opcija izbora umjesto kucanja ulaznog podatka u tekst polje kontrolu može se spriječiti veliki broj nekorektnih unosa. Tačno je da korisnik može odabrati pogrešnu opciju, ali broj ulaznih grešaka uzrokovani na taj način mnogo je manji u odnosu na greške koje se dešavaju proizvoljnim unosom. Naprimjer, za izbora spola bolje je koristiti radio dugme sa izborima (ženski, muški) nego dozvoliti korisniku da unese izbor. Radio dugmad su dobar izbor kada je broj ponuđenih opcija relativno mali i kada je potrebno izabrati samo jednu opciju. Drugi način da se ograniči korisnikov unos su *check boxovi* koji daju slobodu korisniku da potvrdi odnosno izabere više opcija. Za izbore podržane sa radio dugmadima i *check boxovima* preporučuje se korištenje *group boxova*. Korištenje *group box* kontrole pomaže korisniku da usmjeri svoju pažnju na izbor čime se smanjuje broj grešaka. Liste opcija su također korisne za limitiranje unosa i koriste se kada je broj opcija veći. *Combo boxovi* su slični *list boxovima* samo što korisnici mogu pored izbora vrijednosti iz liste i unijeti ulazni podatak. Ukoliko se dozvoli unos podataka tada treba provesti dodatne validacije. Ulagni podatak je često datum. Unos datuma od strane korisnika je podložan greškama jer postoji mnogo različitih formata datuma. Isti problem je i sa unosom vremena. Unos ovih podataka može se standardizirati korištenjem *DateTimePicker* objekta koji omogućava izbor različitih metoda i osobina (*Format* osobina) za formatiranje i ekstrakciju podataka.

Za ekstrakciju podataka prezentiranih sa *DateTimePicker* objektom, u *Short* formatu može se koristiti kod 10.2.

```
string datum = dateTimePicker1.Value.ToShortDateString()
string vrijeme = dateTimePicker1.Value.ToShortTimeString();
```

Kod 10.2: Preuzimanje datuma sa `DateTimePicker` kontrole

`NumericUpDown` kontrola omogućava restrikciju unosa u okviru ranga vrijednosti, čime su već podržane dodatne validacije koje utiču na smanjenje korisničkih grešaka.

10.1.2. Događaji validiranja

Validiranje podatka unesenog na jednu kontrolu može se uraditi kada se desi neki događaj nad kontrolom. Validacija podataka se uglavnom implementira u događajima koji se aktiviraju kada se mijenja fokus neke kontrole ili izvrši neka promjena unosa. Naprimjer, validacija tekst polja se može izvršiti kada se desi događaj `TextChanged`.

Događaji vezani za promjenu fokusa izazvani korištenjem tipki sa tastature, pozivom `Select` ili `SelectNextControl` metoda dešavaju se sljedećim redoslijedom: `Enter`, `GotFocus`, `Leave`, `Validating`, `Validated`, `LostFocus`. Kada se vrši promjena fokusa korištenjem miša ili pozivom `Focus` metode, događaji se dešavaju sljedećim redoslijedom: `Enter`, `GotFocus`, `LostFocus`, `Leave`, `Validating`, `Validated`.

U popisu događaja su i dva namjenska događaja za validaciju: `Validating` i `Validated`. `Validating` događaj se dešava pri pokušaju prebacivanja sa kontrole koja je u fokusu na neku drugu kontrolu. Ovaj događaj se ne može prekinuti. Da bi se događaj izvršio potrebno je da osobina kontrole koja se odnosi na validaciju (`CausesValidation`) bude `true`. `Validated` događaj se dešava nakon što je kontrola uspješno validirana. Izvršavanje ovog događaja se može prekinuti.

Slijedi primjer implementacije (kod 10.3) događaja `Validating` za tekst polje koja se odnosi na pojednostavljenu provjeru matičnog broja.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
using System.ComponentModel;
using System.Windows.Forms;
namespace ValidacijaEMailAdrese
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

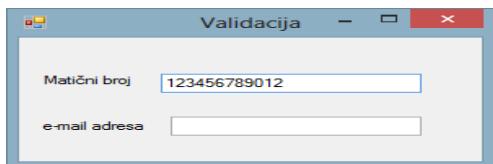
        private void textBox1_Validating(object sender, CancelEventArgs e)
        {
            if (!ValidMaticniBroj(textBox1.Text))
            {
                // Prekida se događaj i fokus se vraća na kontrolu da bi korisnik mogao ponovo izvršiti unos
                e.Cancel = true;
            }
        }
        public bool ValidMaticniBroj(string maticniBroj)
        {
            // provjera da li je matični broj 13 karaktera
            if (maticniBroj.Length != 13)
            {
                MessageBox.Show("Potrebno je unijeti tačno 13 karaktera");
                return false;
            }

            // implementirati dodatnu logiku provjere broja

            return true;
        }
    }
}
```

Kod 10.3: Validating događaj za TextBox kontrolu sa MessageBox kontrolom

Scenarij izvršavanja ovog programskog rješenja:



Nakon unosa 12 cifara za matični broj



Nakon prelaska na polje e-mail adresa

Validacija je implementirana tako da sve dok unos matičnog broja nije uspješno validiran ne može se zatvoriti forma.

RAZVOJ PROGRAMSKIH RJEŠENJA

10.1.3. ErrorProvider kontrola

Izvještavanje o greškama preko `MessageBoxa` može smanjiti korisnikovu efikasnost. Korisnik mora svaku poruku potvrditi. Može se desiti i da korisnik ne zapamti tekst greške. `MessageBox` je namijenjen i za razne obavijesti, upozorenja korisniku ne samo za obavijesti o greškama. Bolja tehnika za prikazivanje poruka o validacijskim greškama je `ErrorProvider` kontrola koja obezbjeđuje interfejs za indikaciju za koju kontrolu na formi postoji povezana greška.

Nalazi se u kategoriji komponenti. Prilikom povezivanja sa formom `ErrorProvider` kontrola se nalazi ispod forme.



Specifične osobine ove kontrole koje se odnose na vizuelne efekte date su u tabeli 10.1.

Osobina	Opis
BlinkRate	Koristi se za podatak o učestalosti blicanja.
BlinkStyle	Odnosi se na informaciju da li će ikona blinkati u slučaju greške. Opcije su: AlwaysBlink, NeverBlink, BlinkIfDifferentError.
Icon	Odnosi se na simbol ikone koja se prikazuje uz kontrolu kada se uspostavi greška.

Tabela 10.1: Osobine `ErrorProvider` kontrole

Kod 10.4 predstavlja modificirani primjer `Validating` događaja `textBox1` kontrole za provjeru matičnog broja. Umjesto `MessageBox.Show` metode koristi se `errorProvider` kontrola, koja je na formu postavljena dizajnerom i nisu se mijenjale njene inicijalne osobine.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
using System.ComponentModel;
using System.Windows.Forms;

namespace ValidacijaEMailAdrese
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void textBox1_Validating(object sender, CancelEventArgs e)
        {
            string porukagreske;

            if (!ValidMaticniBroj(textBox1.Text, out porukagreske))
            {

                // Prekida se događaj i fokus se vraća na kontrolu da bi korisnik mogao ponovo
                izvršiti unos
                e.Cancel = true;

            }
            this.errorProvider1.SetError(textBox1, porukagreske);
        }

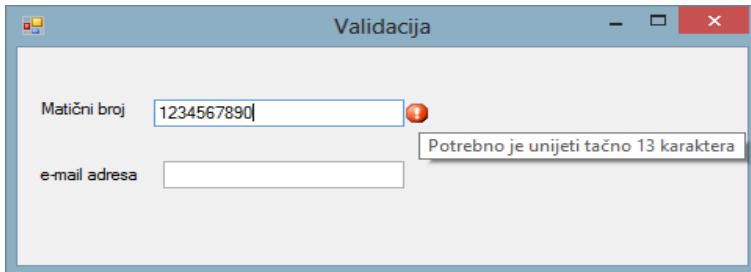
        public bool ValidMaticniBroj(string maticniBroj, out string pgreske)
        {
            // provjera da li je matični broj 13 karaktera
            if (maticniBroj.Length != 13)
            {
                pgreske = "Potrebno je unijeti tačno 13 karaktera";
                return false;
            }
            // implementirati dodatnu logiku provjere broja
            pgreske = "";
            return true;
        }
    }
}
```

Kod 10.4: Validating događaj za TextBox kontrolu sa ErrorProvider kontrolu

Scenarij izvršavanja ovog programskog rješenja:

Nakon unosa pogrešnog matičnog broja i pokušaja prelaska na drugu kontrolu (e-mail adresu) aktivira se Validating događaj i pojavljuje se ikona vezana uz ErrorProvider. Prelaskom mišem preko ikone može se pročitati poruka o grešci.

RAZVOJ PROGRAMSKIH RJEŠENJA



ErrorProvider kontrola se može i upotrebom klase `ErrorProvider` dodati na formu i povezati sa odgovarajućom kontrolom, što je ilustrirano sa kodom 10.5.

```
using System;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        private System.Windows.Forms.ErrorProvider imeErrorProvider;

        public Form1()
        {

            InitializeComponent();

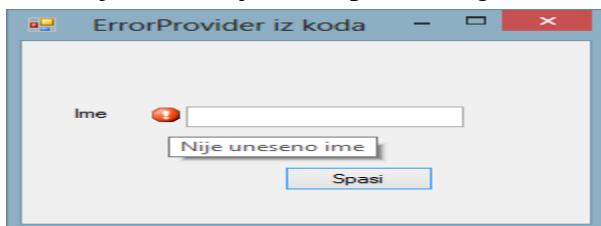
            imeErrorProvider = new System.Windows.Forms.ErrorProvider();
            imeErrorProvider.SetIconAlignment(this.imeTextBox1, ErrorIconAlignment.TopLeft);
            imeErrorProvider.SetIconPadding(imeTextBox1, 2);
            imeErrorProvider.BlinkRate = 1000;
            imeErrorProvider.BlinkStyle = System.Windows.Forms.ErrorBlinkStyle.AlwaysBlink;

        }

        private void imeTextBox1_Validated(object sender, EventArgs e)
        {
            if (imeTextBox1.Text.Length > 0)
            {
                // Briše grešku u ErrorProvideru
                imeErrorProvider.SetError(this.imeTextBox1, String.Empty);
            }
            else
            {
                // Postavlja grešku ako ime nije uneseno
                imeErrorProvider.SetError(this.imeTextBox1, "Nije uneseno ime");
            }
        }
    }
}
```

Kod 10.5: Dinamičko postavljanje `ErrorProvider` forme na kontrolu

Scenarij izvršavanja iznad prikazanog koda:

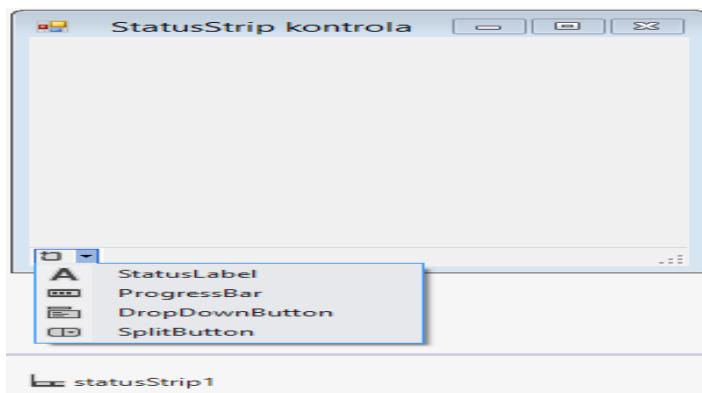


Nakon klika na dugme Spasi pojavljuje se ikona greške postavljena ispred kontrole sa ErrorIconAlignment.TopLeft. Ikona blinka sa učestalosti od 1000 milisekundi (osobine ErrorBlinkStyle.AlwaysBlink, BlinkRate).

10.1.4. StatusStrip kontrola

Poruke o grešci mogu se prikazivati i na statusnoj liniji (statusnom baru) koja se obično postavlja na dnu forme. Statusna linija kreira se sa StatusStrip komponentom. Pogodnost izvještavanja o greškama preko statusne linije je što korisnik može vidjeti poruku, ali ne mora je potvrđivati (klikati).

Prilikom prebacivanja StatusStrip kontrole na formu kontrola se pojavljuje ispod forme, a status bar se dodaje na samu formu (slika ispod).



Komponente StatusStrip kontrole koje se mogu dodati na statusnu liniju su: StatusLabel, ProgressBar, DropDownButton, SplitButton. Za izvještavanja o greškama, i za razna obavještavanja korisnika o ishodu programskih akcija koristi se StatusLabel komponenta.

RAZVOJ PROGRAMSKIH RJEŠENJA

Kod 10.6 ilustrira način kreiranja `StatusStrip` kontrole i njenu upotrebu za izvještavanje o grešci prilikom validacije teksta polja (`imeTextBox`). Validira se dužina polja, koja mora biti veća od 0.

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        private StatusStrip statusStrip1;
        private ToolStripStatusLabel toolStripStatusLabel1;

        public Form1()
        {

            InitializeComponent();
            // Instanciranje StatusStrip kontrole i label komponente
            statusStrip1 = new System.Windows.Forms.StatusStrip();
            toolStripStatusLabel1 = new System.Windows.Forms.ToolStripStatusLabel();

            // Setovanje različitih StatusStrip osobina
            statusStrip1.Dock = System.Windows.Forms.DockStyle.Bottom;
            statusStrip1.GripStyle = System.Windows.Forms.ToolStripGripStyle.Visible;
            statusStrip1.Items.AddRange(new System.Windows.Forms.ToolStripItem[] {
                toolStripStatusLabel1});
            statusStrip1.LayoutStyle =
                System.Windows.Forms.ToolStripLayoutStyle.HorizontalStackWithOverflow;
            statusStrip1.Location = new System.Drawing.Point(0, 0);
            statusStrip1.Name = "statusStrip1";
            statusStrip1.Size = new System.Drawing.Size(280, 20);
            statusStrip1.TabIndex = 0;
            statusStrip1.Text = "statusStrip1";
            statusStrip1.ForeColor = Color.Red;
            //
            // toolStripStatusLabel1
            //
            toolStripStatusLabel1.Name = "toolStripStatusLabel1";
            toolStripStatusLabel1.Size = new System.Drawing.Size(109, 17);

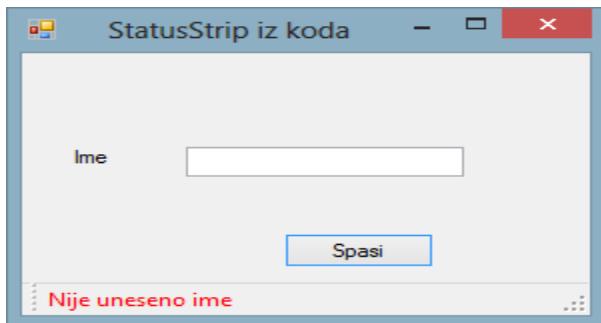
            Controls.Add(statusStrip1);
        }

        private void imeTextBox1_Validate(object sender, EventArgs e)
        {
            if (imeTextBox1.Text.Length > 0)
            {
                // Briše, ako postoji, poruku na statusnoj liniji
                toolStripStatusLabel1.Text = " ";
            }
            else
            {
                // Setuje grešku za slučaj kada ime nije uneseno
                toolStripStatusLabel1.Text = "Nije uneseno ime";
            }
        }
    }
}
```

Kod 10.6: Obavijest o grešci na `StatusStrip` kontroli

U sekciji koda koja se odnosi na postavljanje osobina StatusStrip kontrole nalaze se opcije: DockStyle.Bottom, GripStyle, ForeColor, ToolStripLayoutStyle. U dokumentaciji se mogu naći sve osobine, metode i događaji vezani za ovu kontrolu. Mnoge osobine su iste kao i za meni kontrolu jer obadvije kontrole pripadaju grupi Menus&Toolbar.

Scenarij izvršavanja: Ukoliko se ne unese ime a klikne se na dugme Spasi, Validating događaj se aktivira, ustanovljava se greška i na statusnoj liniji se ispisuje poruka.



10.1.5. Unakrsna validacija

Koncept validacije ulaznih podataka treba pažljivo planirati. Naročito pažljivo treba planirati i implementirati validaciju koja uključuje provjeru ispravnosti međusobno ovisnih podataka sa dvije ili više kontrole. Validacija podataka sa više kontrola, poznata je i kao unakrsna validacija (*cross-checking*). Prilikom unakrsne validacije provjerava se konzistentnost (dosljednost) među podacima. Npr., ako je uneseno da je osoba mlađa od 18 godina tada se ne može označiti da posjeduje vozačku dozvolu, ili ako je izabrano godišnje doba ljeto ne može se iz liste opcija izabrati mjesec decembar.

Validacija podataka može se uraditi na nivou kontrole i na nivou forme. Validacija na nivou forme uglavnom se provodi prilikom spašavanja podataka, da bi se osiguralo spašavanje tačnih podataka. Prilikom planiranja šta treba validirati na nivou kontrole a šta na nivou forme važna je i činjenica da se događaji validacije koji se aktiviraju kada se promijeni fokus kontrole mogu prekinuti ili ne izvršiti ako se izabere meni kontrola.

Na taj način može se desiti da se spase nekonzistentni podaci ukoliko je na meniju opcija Spasi a sve validacije na nivou kontrole.

10.2.Upravljanje izuzecima (*exception handling*)

Bez obzira na implementirane validacije u programu uvjek postoji mogućnost pojave grešaka prilikom izvršavanja programa. Zbog toga programski kod treba da sadrži i dodatne mehanizme otkrivanja grešaka.

10.2.1.Upravljanje izuzecima

Izuzetak (*exception*) je greška koja se dešava za vrijeme izvršavanja programa. Izuzeci nastaju uslijed povrede sistemskih ili aplikacijskih ograničenja ili uslijed pojave specifičnih situacija koje se ne očekuju tokom normalnog izvršavanja programske operacije. Naprimjer, izuzeci se mogu desiti kada program pokušava da dijeli sa nulom ili da piše u fajl koji je samo za čitanje. Kada se dese takvi izuzeci potrebno je da programsko rješenje omogući korisniku ponavljanje operacije koja je izazvala izuzetak ili izbor neke druge opcije programa. U ovoj sekciji se izlažu klase i mehanizmi upravljanja izuzecima (*exception handling*) u okviru C#-a i .NET *frameworka*.

10.2.2. Klase za upravljanje izuzecima

.NET Framework ima opsežni broj klasa za upravljanje izuzecima koji se podižu (*throw*) prilikom izvršavanja individualnih instrukcija za vrijeme izvršavanja programa. Sve klase za upravljanje izuzecima su izvedene iz `SystemException` klase, koja je naslijedena od `Exception` klase. U tabeli 10.2 data je odabrana lista standardnih izuzetaka koji su podržani istoimnim klasama, sa opisanim situacijama kada se pojavljuju.

RAZVOJ PROGRAMSKIH RJEŠENJA

Izuzetak	Razlog nastanka izuzetka
IndexOutOfRangeException	Nastaje pri pokušaju pristupa elementu niza sa indeksom izvan validnih granica.
NullReferenceException	Nastaje pri pokušaju dereferenciranja null reference.
InvalidCastException	Nastaje pri pogrešnoj primjeni cast operadora ili pri pogrešnoj eksplisitnoj konverziji.
OutOfMemoryException	Nastaje kada nema dovoljno memorije da se nastavi izvršavanje programa.
FormatException	Nastaje uslijed nezadovoljavajućeg formata argumenata operacije ili metode.
IOException	Nastaje kada se desi I/O greška.
StackOverflowException	Nastaje kada se desi prekoračenje memorije na steku (npr. uslijed ugnježdenih poziva metoda).
ArgumentException Bazna klasa za: ArgumentNullException, ArgumentOutOfRangeException	Izuzetak nastaje kada nije validan jedan od argumenata metode.
ArgumentNullException	Nastaje pri pozivu metode koja ne dozvoljava null argument, a u pozivu metode je specificiran.
ArgumentOutOfRangeException	Izuzetak nastaje kada vrijednost argumenta metode je izvan dozvoljenog definiranog ranga.
ArithmeticeException Bazna klasa za: DivideByZeroException, NotFiniteNumberException, OverflowException	Izuzeci ovog tipa nastaju pri izvršavanju aritmetičkih operacija i operacija konverzije.
DivideByZeroException	Izuzetak nastaje pri pokušaju dijeljenja cjelobrojnih ili decimalnih vrijednosti sa nulom.
NotFiniteNumberException	Nastaje kada je <i>floating-point</i> vrijednost beskonačna ili nije broj.
OverflowException	Nastaje kada aritmetička operacija ili operacija konverzije rezultira prekoračenjem.
ApplicationException	Izuzeci za ne-fatalne aplikacijske greške.

Tabela 10.2: Odabrana lista standardnih izuzetaka

Osnovna `Exception` klasa definira i osobine koje sadrže detaljniji opis izuzetka.

Korisne osobine klase su date u tabeli 10.3.

Osobina	Opis
Message	Sadrži poruku koja opisuje nastali izuzetak.
StackTrace	Sadrži detalje o steku i vrijeme kada se izuzetak desio.
TargetSite	Sadrži ime metode u kojoj se desio izuzetak.
Source	Odnosi se na naziv aplikacije ili objekta koji uzrokuje izuzetak.
HelpLink	Sadrži link na fajl pomoći u vezi izuzetka.
InnerException	Ako je izuzetak uzrokovan drugim izuzetkom ova osobina sadrži referencu na prethodni izuzetak.

Tabela 10.3: Korisne osobine `Exception` klase

10.2.3. Try/catch/finally struktura za upravljanje izuzecima

Tradicionalni načina upravljanja greškama uglavnom miješa iskaze programske logike i iskaze namijenjene upravljanju greškama. Objektno orijentirana solucija za upravljanje greškama (*error handling*) usvaja koncept fizičke separacije osnovnih programskih iskaza i iskaza za upravljanje greškama. .NET podržava ovaj način upravljanja greškama kroz `try`, `catch`, i `finally` blokove.

```

try
{
    // Programska logika
} //sekcija catch blokova, broj catch blokova je proizvoljan
catch ([klasa_izuzetka1] [identifikator])
{
    // Kod za upravljanje greškama tipa klase klasa_izuzetka1
    // identifikator se koristi za dobijanje detaljnijih informacija o izuzecima
}
catch ([klasa_izuzetka2] [identifikator])
{
    //Kod za upravljanje greškama tipa klase klasa_izuzetka 2
}
catch
{
    //Blok za upravljanje svim izuzecima
    //izvršava se ako nijedan catch u sekciji catch blokova nije obradio izuzetak
}

finally
{
    // Kod koji je potrebno izvršiti bez obzira da li je izvršen kod u okviru nekog catch bloka
    // Blok je opcionalan, jedino je obavezan ako u sekciji catch blokova ne postoji nijedan catch
}

```

Try/catch/finally struktura za upravljanje izuzecima

RAZVOJ PROGRAMSKIH RJEŠENJA

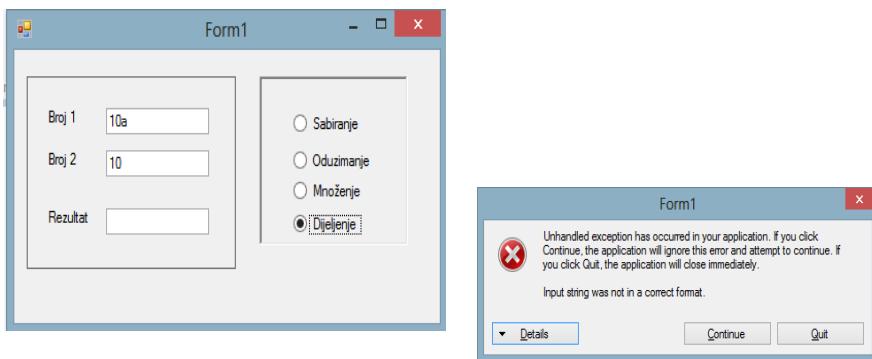
Prvi korak u procesu upravljanja izuzecima je detekcija programskih iskaza (koda) koji mogu uzrokovati izuzetke, kao i detekcija izuzetaka koji se mogu desiti kada se izvršavaju ti iskazi. Kod kojim se mogu uzrokovati izuzeci stavlja se unutar `try` bloka. Sekcija `catch` klauzula koristi se za upravljanje izuzecima. Obično sadrži jednu ili više `catch` klauzula, pri čemu je svaka klauzula vezana za specifičnu klasu izuzetka. Na kraju osnovne strukture za upravljanje greškama je opcionalni `finally` blok koji sadrži kod koji se izvršava bez obzira da li je izazvan i obrađen izuzetak. Pretežno sadrži kod za razna čišćenja (*cleanup code*). Moguće je da `catch` sekcija ne sadrži nijednu klauzulu, u tom slučaju `finally` blok je obavezan.

Prilikom izvršavanja programa ukoliko se desi izuzetak zaustavlja se normalno izvršavanje programa i traži se `catch` blok bazirano na tipu izuzetka koji može obraditi izazvani izuzetak. Ako odgovarajući `catch` blok nije nađen u metodi u kojoj je izuzetak izazvan, traži se blok za upravljanje izuzetkom u metodi koja je pozvala tu metodu. Proces se nastavlja sve dok se ne nađe izuzetak ili dok se ne dostigne krajnja tačka programa.

U sekciji `catch` blokova preporučljivo je na kraju te sekcijske pisati generalnu `catch` klauzulu bez navedene klase ili sa navedenom klasiom `System.Exception` (`catch (System.Exception e)`) ili `catch`). Na taj način je omogućena detekcija svih izuzetaka.

Ukoliko za Windows aplikaciju koja treba da omogući osnovne matematičke operacije nije implementirano upravljanje greškama/izuzecima mogu se pojaviti razni izuzeci prilikom njenog izvršavanja.

U tom slučaju, ukoliko korisnik unese za ulazne brojeve podatke kao na slici ispod i izabere operaciju dijeljenja javlja se izuzetak opisan na posebnom prozoru.



Izuzetak koji se pojavio je veoma zbumujući za korisnika. Evidentno je da u događaju povezanim sa klikom na radio dugme Dijeljenje nije implementirano upravljanje greškama. Osim ilustriranog izuzetka koji je nastao zbog neočekivanog formata broja1, prilikom izvršavanja metode za upravljanje događajem može se desiti i izuzetak uzrokovan dijeljenjem sa nulom, ili izuzetak uslijed prekoračenja dozvoljene vrijednosti za cijelobrune brojeve. Za sve izuzetke potrebno je dati i koristan opis poruke prilagođen korisniku. U nekim slučajevima

RAZVOJ PROGRAMSKIH RJEŠENJA

potrebno je pitati korisnika da ponovo unese traženu vrijednost ili vratiti fokus na kontrolu koja je uzrokovala izuzetak.

Slijedi metoda, kod 10.7, za upravljanje događajem `Click` radio dugmeta Dijeljenje sa implementiranim mehanizmom upravljanja greškama.

```
private void dijeljenjeradioButton_CheckedChanged(object sender, EventArgs e)
{
    try
    {
        (5) int broj1 = Convert.ToInt32(broj1TextBox.Text);
        (6) int broj2 = Convert.ToInt32(broj2TextBox.Text);

        (7)     int rezultat = broj1 / broj2;
                rezultatTextBox.Text = (rezultat).ToString();
    }
    catch (FormatException )
    {
        MessageBox.Show("Neodgovarajući format ulaznih vrijednosti","Izuzetak");
    }
    catch (DivideByZeroException)
    {
        MessageBox.Show("Dijeljenje sa nulom","Izuzetak");
        broj2TextBox.Focus();
    }
    catch (OverflowException)
    {
        MessageBox.Show("Prekoračenje – Preveliki ulazni brojevi","Izuzetak");
    }
    catch (Exception ex)
    {
        MessageBox.Show("Greška:" +ex.Message+" Izvor:" + ex.Source,"Izuzetak");
    }
    finally
    {
        MessageBox.Show(" Finally blok-trenutno bez koda radi ilustracije");
    }
}
```

Kod 10.7: Metoda za upravljanje izuzecima prilikom dijeljenja dva cjelobrojna broja

Programski kod sada upravlja sa specifičnim izuzecima (dijeljenje sa nulom, neodgovarajući format, prekoračenje) ali može da prepozna i svaki nespecificirani izuzetak ako se desi. Za specifične, prepoznate izuzetke data je i prilagođena poruka za korisnika, dok za ostale izuzetke data je poruka sadržana u osobinama `Exception` klase.

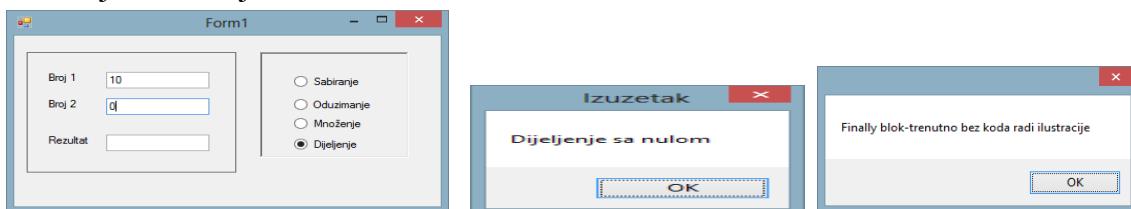
Iskazi na linijama 5 i 6 mogu uzrokovati `FormatException` ili `OverflowException` prilikom smještanja unesenih vrijednosti u odgovarajuće varijable. Ako se desi `FormatException` kontrola programa se prenosi na `catch(FormatException)` blok koji obrađuje ('hvata') ovaj

RAZVOJ PROGRAMSKIH RJEŠENJA

izuzetak. Nakon toga izvršava se `finally` blok. Ako su iskazi na linijama 5 i 6 uzrokovali `OverflowException` kontrola programa se prenosi na `catch(OverflowException)` i nakon toga se izvršava `finally` blok. Iskaz dijeljenja na liniji 7 može prouzrokovati `DivideByZeroException`. Ako jeste tada programska kontrola toka se prebacuje na `catch` blok označen sa klasom `DivideByZeroException` i nakon toga se izvršava `finally` blok.

Ukoliko se desio izuzetak koji nije obuhvaćen sa navedene tri `catch` klauzule izvršava se zadnji navedeni generalni `catch` blok i `finally` blok. Ako nijedan od iskaza u `try` bloku nije prouzrokovao grešku, programski tok u `try` bloku dostiže do kraja i transfer programskog toka se prenosi na `finally` blok.

Scenarij izvršavanja:



Nakon unosa vrijednosti i klika na radio dugme Dijeljenje, obrađuje se izuzetak Dijeljenje sa nulom, poruka iz `finally` bloka

10.2.4. Eksplisitno pozivanje izuzetaka

U kodu se može i eksplisitno pozvati (izazvati, prijaviti) izuzetak korištenjem `throw` iskaza. Sintaksa za `throw` iskaz je: `throw objekatTipaExceptionKlase;`

Izuzetak se može izazvati iskazom iz `try` bloka ili iz bilo koje direktno ili indirektno pozvane metode iz `try` bloka. Tačka sa koje se `throw` izvršava naziva se tačka poziva (`throw point`). Kada se izuzetak pozove, kontrola se ne može vratiti na tačku poziva.

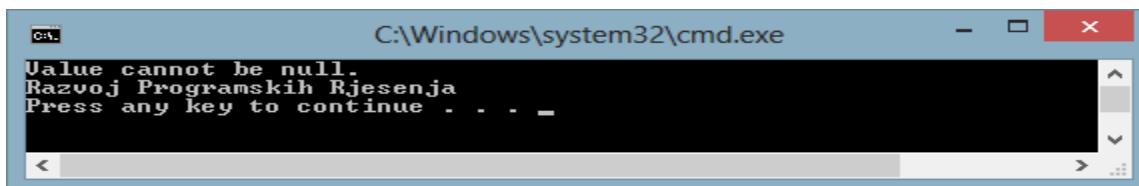
Kod 10.8 definira metodu `PrimjerArgExc` koja kao parametar prima string i štampa ga. Unutar `try` bloka, prvo se provjerava da li je argument `null`. Ako jeste, kreira se instanca tipa `ArgumentNullException` i izaziva se izuzetak tog tipa. Izuzetak se obrađuje u `catch` bloku (`catch (ArgumentNullException e)`), u kojem se i ispisuje poruka o izuzetku. Unutar maina metoda `PrimjerArgExc` poziva se dva puta: jednom sa validnim parametrom i jednom sa `null` parametrom.

```
namespace ConsoleApplicationExc
{
    class MyClass
    {
        public static void PrimjerArgExc(string arg)
        {
            try
            {
                if (arg == null)
                {
                    ArgumentNullException argExc = new ArgumentNullException();
                    throw argExc; // izaziva se izuzetak
                }
                Console.WriteLine(arg);
            }
            catch (ArgumentNullException e) // otkriva se i obrađuje izuzetak
            {
                Console.WriteLine(e.Message); //ispisuje poruku Value cannot be null
            }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            string s=null;
            MyClass.PrimjerArgExc(s); // poziv metode sa null argumentom
            MyClass.PrimjerArgExc("Razvoj Programskih Rjesenja");
        }
    }
}
```

Kod 10.8: Eksplicitni poziv (throw) izuzetka ArgumentNullException

U okviru metode `PrimjerArgExc`, kada se prvi put pozove metoda ustanovljava se da je argument `null` i eksplicitno se u okviru metode `PrimjerArgExc` izaziva izuzetak `ArgumentNullException`. Izazvani izuzetak se obrađuje u `catch` bloku unutar iste metode.

Izvršavanje programa:



RAZVOJ PROGRAMSKIH RJEŠENJA

Primjer: Za aplikaciju kalkulatora postavljeno je dodatno ograničenje da vrijednost broja1 i broja2 mora biti u rangu od 0 do 5000. Ako vrijednosti nisu u navedenim granicama izaziva se izuzetak. Može se koristiti postojeća klasa `IndexOutOfRangeException` koja će se pozivati sa `throw` iskazom za obradu nastalog izuzetka. Pri eksplisitnom pozivu (`throw`) izuzetka predaje se prilagođena poruka za opis izuzetka.

Slijedi dio koda (kod 10.9) metode za upravljanje događajem `click` radio dugmeta Dijeljenje koji ilustrira eksplisitni poziv izuzetka (`throw`).

```
try
{
    int broj1 = Convert.ToInt32(broj1TextBox.Text);

    if (broj1 < 0 || broj1 > 5000)
        throw new IndexOutOfRangeException("Broj 1 je " + broj1);

    int broj2 = Convert.ToInt32(broj2TextBox.Text);

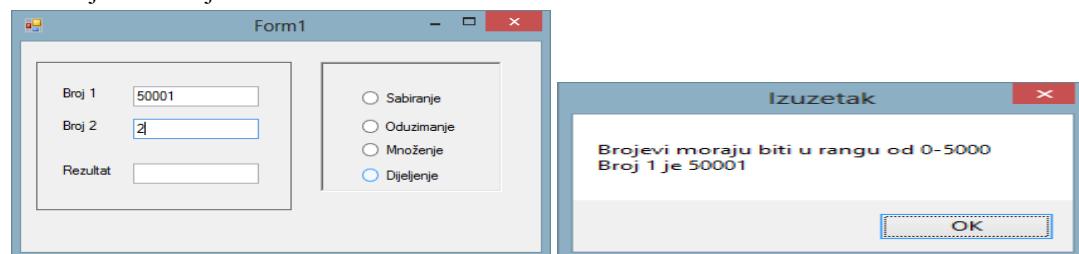
    if (broj2 < 0 || broj2 > 5000)
        throw new IndexOutOfRangeException("Broj 2 je " + broj1);

    int rezultat = broj1 / broj2;
    rezultatTextBox.Text = (rezultat).ToString();
}

catch (IndexOutOfRangeException ex)
{
    MessageBox.Show("Brojevi moraju biti u rangu od 0-5000 \n" +
ex.Message, "Izuzetak");
}
// slijede prikazani catch blokovi za ostale moguće izuzetke
```

Kod 10.9: Eksplisitni poziv izuzetka `IndexOutOfRangeException` sa prilagođenom porukom

Scenarij izvršavanja:



Nakon unosa brojeva, broj1 izvan dozvoljenog ranga, i klik na radio dugme Dijeljenje izaziva se i obrađuje izuzetak `IndexOutOfRangeException`. Drugi prozor prikazuje prilagođenu poruku o izuzetku.

Izazivanje izuzetka bez objekta

Throw iskaz može se koristiti i bez objekta izuzetka, unutar catch bloka. Ta forma ponovo izaziva (*rethrows*) isti izuzetak. U tom slučaju nastavlja se potraga za metodom za obradu izuzetka u vanjskim blokovima. Ukoliko se radi o ugnježdenim try-catch blokovima u okviru iste metode prvo se klasa za obradu izuzetka traži u cath blokovima povezanim sa prvim vanjskim try blokom. Proces se nastavlja u ovisnosti od broja nivoa ugnježdenja. Ukoliko nije pronađen odgovarajući catch blok u metodi u kojoj je nastao izuzetak potraga se nastavlja unutar bloka koji je pozvao metodu. Proces traje sve dok se ne pronađe catch blok ili dok se ne dosegne krajnja tačka programa.

Postoje dvije opcije za upravljanje izuzecima koji se dešavaju unutar metode. Prva opcija je obrada izuzetka u metodi u kojoj je nastao izuzetak bez obavijesti metode koja je pozvala tu metodu. Druga opcija je da se o nastalom izuzetku obavijesti i metoda koja je pozvala metodu u kojoj je nastao izuzetak. Ako se izabere druga opcija izuzetak se u metodi u kojoj je nastao mora ponovo izazvati u catch klauzuli. Slijedi kod 10.10 kojim se podržava druga opcija.

```
using System;
namespace Exception3
{
    class MyClass
    {
        public static void PrimjerArgExc(string arg)
        {

            try
            {
                if (arg == null)
                {
                    ArgumentNullException argExc = new ArgumentNullException("arg");
                    throw argExc;
                }
            }

            catch (ArgumentNullException e)
            {
                Console.WriteLine("Unutrašnji catch u metodi {0} ", e.Message);
                throw;      //***
            }
        }
    }
}

// nastavak na sljedećoj stranici
```

RAZVOJ PROGRAMSKIH RJEŠENJA

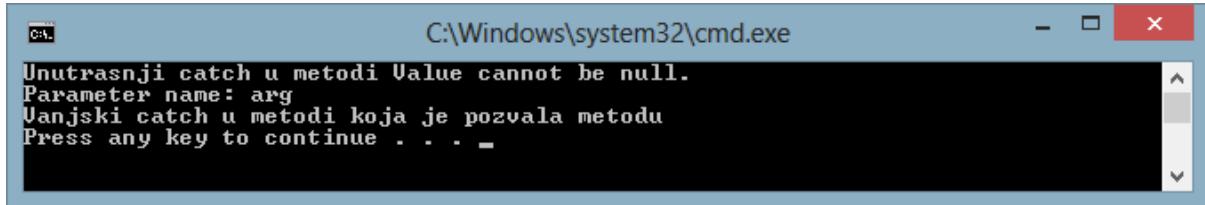
```
class Program
{
    static void Main(string[] args)
    {
        string s=null;

        // poziv metode je u try-catch bloku
        try
        {
            MyClass.PrimjerArgExc(s);
        }

        catch
        {
            Console.WriteLine("Vanjski catch u metodi koja je pozvala metodu");
        }
    }
}
```

Kod 10.10: Ponovno izazivanje izuzetka u catch bloku metode u kojoj je izazvan

Izvršavanje progama:

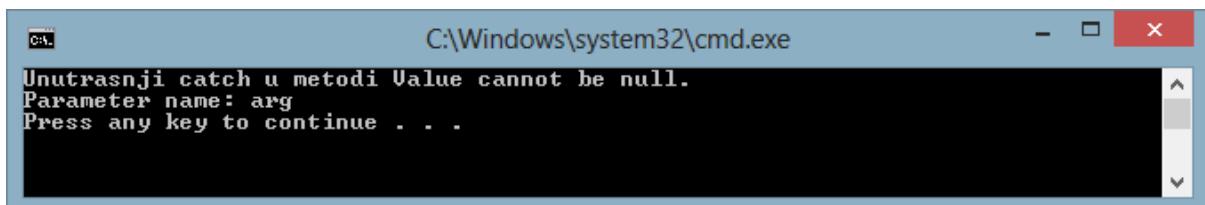


```
C:\Windows\system32\cmd.exe
Unutrasnji catch u metodi Value cannot be null.
Parameter name: arg
Vanjski catch u metodi koja je pozvala metodu
Press any key to continue . . .
```

Metoda `PrimjerArgExc` u kojoj je izazvan izuzetak je obradila izuzetak, izazvala isti izuzetak, koji je proslijedila u metodu `main` koja je pozvala ovu metodu. S obzirom da je poziv metode `PrimjerArgExc` u `main` metodi bio u okviru `try/catch` bloka izuzetak je obrađen i u metodi `main`.

Ukoliko bi se linija koda označena sa (`throw; //***`) obilježila kao komentar tada bi se procesiranje izuzetka obavilo u metodi `PrimjerArgExc`. Metoda koja je pozvala `PrimjerArgExc` ne bi ni imala informaciju da se desio izuzetak.

Izvršavanje programa bez `throw` iskaza:



```
C:\Windows\system32\cmd.exe
Unutrasnji catch u metodi Value cannot be null.
Parameter name: arg
Press any key to continue . . .
```

Korisno je imati poziv metode u okviru `try` bloka sa generalnom `catch` klauzulom, jer u slučaju da se unutar metode koja je pozvana dese i drugi izuzeci za koje nisu specifikirane klase izuzetaka isti bi se obradili u metodi koja je pozvala tu metodu.

10.3. Kreiranje klase za obradu izuzetka

Moguće je dizajnirati i definirati vlastitu klasu za obradu određenih specifičnih izuzetaka. Klasu je potrebno naslijediti od `Exception` klase. Sve klase naslijedene iz `Exception` klase imaju 3 osnovna konstruktora: konstruktor bez parametra, konstruktor sa parametrom koji se odnosi na specificiranu poruku greške, i konstruktor koji inicijalizira novu instancu izuzetka sa specificiranom porukom o grešci i referencom na unutarnji izuzetak koji je uzrokovao izuzetak. Četvrti konstruktor ukoliko postoji odnosi se na kreiranje instance klase sa serijaliziranim podacima.

Primjer: Za potrebe metode `ProsjecnaVrijednost()` potrebno je definirati klasu `BrojacJeNegativanException` za obradu izuzetka koji se dešava u slučaju negativne vrijednosti broja elemenata za koje se računa prosječna vrijednost. Kod 10.11 prikazuje moguće rješenje za postavljeni zadatak.

```
using System;
namespace UserDefinedExc
{
    // definiranje korisničke klase za obradu izuzetka
    public class BrojacJeNegativanException : Exception      // klasa nasljedena od Exception klase
    {
        // klasa posjeduje 3 konstruktora
        public BrojacJeNegativanException()
        {
        }
        public BrojacJeNegativanException(string message) : base(message)
        {
        }
        public BrojacJeNegativanException(string message, Exception inner) : base(message, inner)
        {
        }
    }
    // testiranje: izazivanje izuzetka tipa korisnički definirane klase BrojacJeNegativanException
    class Test
    {
        public static void prosjecnaVrijednost(int suma, int broj)
        {
            float prVrijednost;

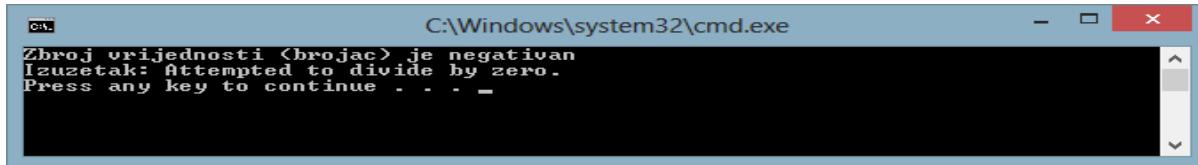
            try
            {
                if (broj < 0)
                {
                    // izazivanje izuzetka tipa klase BrojacJeNegativanException
                    throw (new BrojacJeNegativanException());
                }
                else
                {
                    prVrijednost = suma / broj;
                }
            }
            catch (BrojacJeNegativanException)
            {
                Console.WriteLine("Brojač vrijednosti je negativan");
            }
        }
    }
    // nastavak na sljedećoj stranici
```

RAZVOJ PROGRAMSKIH RJEŠENJA

```
public static void Main()
{
    try
    {
        prosjecnaVrijednost(200, -10);
        prosjecnaVrijednost(200, 0);
    }
    catch (Exception e)
    {
        Console.WriteLine("Izuzetak: {0}", e.Message);
    }
}
```

Kod 10.11: Definicija i test specifične korisničke klase za obradu izuzetka

Izvršavanje gore navedenog koda daje:



Primjeri dati u ovom poglavlju ilustriraju koncept validacije i upravljanja greškama. Forme koje su prikazana nisu povezivane sa klasama. Organizacija poziva metoda validacije u slučaju postojanja modela klasa se mijenja. Metode validacije podataka klase se u tom slučaju najčešće implementiraju u klasama i vrši se njihovo pozivanje sa ulazno-izlaznim operacijama, događajima nastalim uslijed interakcije sa korisnicima.

Rezime:

U okviru ovog poglavlja razmatran je značaj validacije ulaznih podataka. Objasnjen je način interakcije sa korisnikom u slučaju nevalidnih unosa podataka. Jedan od načina interakcije je korištenje poruka, a drugi korištenjem posebnih klasa za identifikaciju grešaka (`ErrorProvider`) i ispis tih grešaka na statusnu liniju. Sama validacija podataka nije dovoljno osiguranje za sprječavanje grešaka, pa je u programima potrebno izvršiti i dodatno upravljanje greškama preko posebnih programskih blokova (`try/catch/finally`) i upotrebom specijaliziranih klasa za upravljanje greškama.

RAZVOJ PROGRAMSKIH RJEŠENJA

Pitanja i zadaci za samostalni rad:

1. Šta je validacija podataka?
2. Zašto su potrebne validacije podataka?
3. Zašto je unos preko `TextBox` kontrole podložan greškama?
4. Kojim provjerama podataka se mogu spriječiti mnoge greške pri unosu podataka?
5. Kojom metodom se može provjeriti tip ulaznog podatka?
6. Koje GUI kontrole se koriste da bi se smanjio broj ulaznih grešaka?
7. Kako se mogu spriječiti greške prilikom unosa datuma?
8. Navedite događaje validacije vezane za validaciju podataka pojedine kontrole.
9. Objasnite događaje `Validating` i `Validated`.
10. Objasnite namjenu `ErrorProvider` kontrole.
11. Zašto se preporučuje korištenje `ErrorProvider` umjesto `MessageBoxa` za izvještavanje o greškama?
12. Navedite neke osobine koje se odnose na vizuelne efekte izvještavanja o greškama `ErrorProvider` kontrole.
13. Objasnite namjenu `StatusStrip` kontrole.
14. Koje su komponente `StatusStrip` kontrole koje se mogu dodati na statusnu liniju?
15. Objasnite pojam unakrsne validacije.
16. Koja je namjena validacije podataka na nivou forme.
17. Šta je izuzetak?
18. Zašto nastaju izuzeci?
19. Obrazložiti važnost upravljanja izuzecima u okviru programskog rješenja?
20. Koja je osnovna klasa za upravljanje izuzecima u okviru .NET-a?
21. Navedite i objasnite osnovne izuzetke koji se često dešavaju.
22. Navedite i objasnite namjenu i osobine `Exception` klase.
23. Objasnite `try/catch/finally` strukturu za upravljanje greškama.
24. Koja je uloga `try` bloka u `try/catch/finally` strukturi za upravljanje greškama?
25. Koja je uloga `catch` bloka u `try/catch/finally` strukturi za upravljanje greškama?
26. Koliko `catch` blokova može sadržavati `try/catch/finally` struktura?
27. Koja je namjena generalnog bloka, kako se piše i gdje se smješta u `try/catch/finally` strukturi?
28. Koja je uloga `finally` bloka u `try/catch/finally` strukturi za upravljanje greškama?
29. Da li je `finally` blok obavezan u `try/catch/finally` strukturi?
30. Kako se vrši eksplisitno izazivanje izuzetka?
31. Koja je namjena izazivanja izuzetaka bez objekta i kako se provodi?
32. Objasnite mehanizam potrage za izuzetkom ukoliko postoji više ugnježdenih `try/catch` struktura kojima se obrađuju izuzeci.
33. Navedite i objasnite dvije opcije za upravljanje izuzecima koje se dešavaju unutar metode.

RAZVOJ PROGRAMSKIH RJEŠENJA

34. Objasnite način kreiranja vlastite klase za obradu izuzetka.
35. Potrebno je razviti programsko rješenje koje će omogućiti evidenciju predstava po scenariju koji je opisan ispod.

Na jednom tabu na formi se evidentiraju predstave, i to predstave koje izvodi matično pozorište i gostujuće predstave. Za sve predstave unose se sljedeći osnovni podaci: naziv predstave, dužina trajanja predstave (validacija dužine trajanja predstave opisana pod c), broj glumaca u predstavi (mora biti veći od 5), datum kada je predstava prvi put izvedena (kontrolirati da je datum u rasponu od 2009-2013 godine). Za gostujuće predstave dodatno se upisuje naziv gostujućeg pozorišta, i rezidentno mjesto pozorišta (kontrolisati da je to lista gradova: Sarajevo, Mostar, Tuzla, Zenica).

Treba omogućiti i prodaju karata za predstave preko istog ili drugog taba. Za svaku predstavu može se prodati maksimalno 100 karata, pri čemu se preko jedne transakcije može prodati maksimalno 20 karata. Cijena karte je $1\text{KM} + 0.1\text{KM}$ za svaku minutu trajanja predstave. Za svaku transakciju prodaje treba evidentirati šifru kome je karta/karte prodane (validacija šifre opisana pod b), broj prodanih karata, ukupnu naplatu i naziv predstave za koju su karte prodane.

Kreirati GUI sa odgovarajućim kontrolama i validacijama (koje su navedene u tekstu scenarija) za unos predstava i prodaju karata. Uspostaviti i dobro dizajniran model klasi.

- b) Obavezno koristiti `ErrorProvider` i `StatusStrip` za sljedeću validaciju:
 - Šifra osobe kojoj se karta prodaje mora obavezno biti sastavljena od 8 karaktera, pri čemu prva dva karaktera moraju biti cifre, a najviše tri karaktera mogu biti malo slovo.
- c) Za validaciju trajanja predstave kreirati vlastitu klasu za upravljanjem izuzecima prilikom unosa trajanja predstave. Trajanje predstave mora biti u rangu 60-125 minuta.

Poglavlje 11: Rad sa datotekama i direktorijima

U okviru ovog poglavlja obrađuju se način manipulacije sa podacima smještenim u raznim vrstama datoteka primjenom C# programskog jezika kroz sljedeće tematske jedinice:

- 1.Uvod u datoteke i pregled klase za rad sa datotekama
2. I/O izuzeci
3. Klase za rad sa datotekama i direktorijima
4. Rad sa generičkim klasama za ulaz i izlaz
- 5.Serijalizacija i deserijalizacija objekata

11.1.Uvod u datoteke i pregled klasa za rad sa datotekama

Mnoga programska rješenja zahtijevaju razne stalne pohrane podataka. Jedan od načina persistenntnog pohranjivanja podataka je smještanje podataka u datoteke.

Datoteka (*file*) je uređena i imenovana kolekcija bajta smještena na disku ili nekom drugom stalnom mediju za pohranu podataka. Suprotno, *stream* (eng. stream, uslijed neadekvatnog prevoda u tekstu se koristi originalna engleska riječ označena italic) je apstraktna reprezentacija sekvenca bajta koja se koristi za čitanje i pisanje. Ta sekvenca bajta može biti smještena na raznim medijima. To može biti datoteka na disku, memorijska lokacija, mrežni kanal ili neki drugi objekat koji podržava čitanje, pisanje ili obadvoje. Nivo apstrakcije nad fizičkim podacima omogućava developerima da pišu generičke rutine jer ne moraju da brinu o specifičnim detaljima vezanim za transfer podataka. Zbog toga se isti kod može koristiti bez obzira da li aplikacija čita iz ulazne datoteke, mrežnog ulaznog *streama* ili neke druge vrste *streama*. Postoje dva *stream* tipa: izlazni i ulazni. Izlazni *stream* se koristi za pisanje podataka na neku eksternu destinaciju, koja naprimjer može biti fizička datoteka na disku, mrežna lokacija, printer. Ulazni *stream* se koristi za čitanje i smještanje podataka u memoriju kojoj program može pristupiti. Ulazni *stream* podataka može biti tastatura, datoteka na disku ili neki drugi uređaj koji podržava ulazne operacije. U ovom poglavlju fokus je na datotečnom sistemu podataka.

RAZVOJ PROGRAMSKIH RJEŠENJA

Pregled klasa za rad sa datotekama i direktorijima

`System.IO` namespace sadrži veliki broj klasa za rad sa datotekama (*file*) i direktorijima. Prikaz osnovnih klasa u imenovanom prostoru `System.IO` je u tabeli 11.1.

Koristan je i `System.IO.Compression` namespace, koji omogućava čitanje iz i pisanje u kompresovane datoteke. Sadrži više klasa, dvije izuzetno korisne su: `DeflateStream` i `GzipStream` koje omogućavaju automatsku kompresiju podataka prilikom pisanja podataka u datoteku i dekompresuju podataka prilikom čitanja iz datoteke. Kompresija se obavlja sa Deflate i GZIP algoritmom.

Klasa	Opis-namjena klase
<code>File</code>	Klase sa mnogim metodama prvenstveno za kreiranje, otvaranje, pomjeranje, kopiranje, brisanje datoteka. Koriste se za povezivanje sa <code>FileStream</code> klasom.
<code>FileInfo</code>	
<code>Directory</code> <code> DirectoryInfo</code>	Uključuju mnoge metode za kreiranje, pomjeranje, kopiranje, brisanje direktorija.
<code>FileSystemInfo</code>	Služi kao bazna klasa za <code>FileInfo</code> i <code> DirectoryInfo</code> , čime je omogućeno da se radi sa direktorijima i datotekama korištenjem polimorfizma.
<code>Path</code>	Klasa se koristi za manipulaciju sa nazivima puteva.
<code>StreamReader</code>	Čita podatke iz <i>streama</i> . Implementira <code>TextReader</code> .
<code>StreamWriter</code>	Piše podatke u <i>stream</i> . Implementira <code>TextWriter</code> .
<code>BinaryReader</code>	Čita osnovne tipove podataka kao binarne vrijednosti.
<code>BinaryWriter</code>	Binarno zapisuje osnovne tipove podataka.
<code>FileSystemWatcher</code>	Klasa se koristi za monitoring datoteka i direktorija.

Tabela 11.1: Klase za rad sa datotekama i direktorijima u `System.IO`

11.2. I/O izuzeci

Prilikom rada sa datotekama i direktorijima moguća je pojava velikog broja izuzetaka uslijed povrede prava pristupa, grešaka na disku, mreži, nemogućnosti pronalaska specificiranih datoteka i slično. Zbog toga je korisno koristiti klase za obradu izuzetaka. U tabeli 11.2 dat je pregled osnovnih izuzetaka. Klase za obradu izuzetaka uglavnom su naslijedene od

RAZVOJ PROGRAMSKIH RJEŠENJA

`SystemException` klase. U ovisnosti od metode koja se izvršava mogu se javiti izuzeci opisani u tabeli 11.2 ili neki dodatni karakteristični izuzeci vezani za tu metodu.

Izuzetak (<i>Exception</i>)	Opis izuzetka/klase
<code>ArgumentException</code>	Izuzetak nastaje kada jedan od argumenata metode nije validan. Za datoteke često se navedu pogrešni parametri prilikom kreiranja, otvaranja, čitanja. Npr. pogrešan put, pogrešan pristup datoteci.
<code>UnauthorizedAccessException</code>	Nastaje uslijed pokušaja pristupa datoteci kojoj je pristup zabranjen od strane operativnog sistema.
<code>FormatException</code>	Izuzetak nastaje kada ulazna datoteka ili <i>stream</i> nije u formatu koji odgovara specifikaciji.
<code>InvalidDataException</code>	Izuzetak nastaje kada je <i>stream</i> neodgovarajućeg formata.
IOException -osnovna klasa za izuzetke: <code>DirectoryNotFoundException</code> <code>DriveNotFoundException</code> <code>EndOfStreamException</code> <code>FileNotFoundException</code>	Koristi se za obradu izuzetaka pri izvršavanju I/O operacija.
<code>DirectoryNotFoundException</code>	Izuzetak nastaje kada se ne može naći specificirani direktorij.
<code>DriveNotFoundException</code>	Izuzetak nastaje pri pokušaju pristupa drajvu koji ne postoji.
<code>EndOfStreamException</code>	Izuzetak nastaje pri pokušaju čitanja nakon kraja <i>streama</i> .
<code>FileNotFoundException</code>	Izuzetak nastaje pri pokušaju pristupa datoteci koja ne postoji na disku.

Tabela 11.2: Izuzeci pri radu sa datotekama i direktorijima

11.3. Klase za rad sa datotekama i direktorijima

U ovoj sekciji se detaljnije opisuju `File` i `Directory` klase. Primjerima se ilustruje njihova primjena.

11.3.1. `File` klasa

Osnovna klasa za rad sa datotekama u .NET okruženju je `File` klasa. Ova klasa ima veliki broj metoda. Izdvajaju se metode za kreiranje, kopiranje, brisanje, pomjeranje i otvaranje datoteka. Pregled dijela metoda ove klase dat je u tabeli 11.3. Parametar većine metoda je apsolutni ili relativni put do datoteke.

Metoda	Opis metode
<code>Create(string nazivDat, [Int32 veličinaBafera], [FileOptions opcijaDat], [FileSecurity zaštitaDat])</code>	Kreira novu datoteku sa specificiranim nazivom. Ako već postoji datoteka sa istim nazivom uništava se njen sadržaj. Metoda vraća objekat tipa <code>FileStream</code> . Opcionalni su parametri koji se odnose na veličinu bafera, opcije datoteke i zaštitu datoteke.
<code>Delete(string nazivDat)</code>	Briše specificiranu datoteku.
<code>Open(string nazivDat, [FileMode modDat], [FileAccess pristupDat], [FileShare dijeljenjeDat])</code>	Otvara specificiranu datoteku. Metoda vraća objekat tipa <code>FileStream</code> . Opcionalni parametri se odnose na način (mod) otvaranja datoteke, mod pristupa i način dijeljenja datoteke. Opcije za <code>FileMode</code> , <code>FileAccess</code> , <code>FileShare</code> su navedene u 11.4.1.
<code>Move(string izvorisniNazivDat, string odredisniNazivDat)</code>	Pomjera specificiranu datoteku na novu lokaciju. Može se specificirati različito ime za datoteku na novoj lokaciji.
<code>Exists(nazivDat)</code>	Određuje da li postoji specificirana datoteka.
<code>Replace(string nazivDatKojaSeZamjenjuje, string nazivDatKojomSeZamjenjuje, String nazivBekapDat)</code>	Zamjenjuje sadržaj specificirane datoteke prvim parametrom, sa datotekom specificiranom drugim parametrom. Kreira bekap (<i>backup</i>) zamijenjene datoteke.
<code>CreateText(string nazivDat)</code>	Kreira ili otvara datoteku za pisanje u UTF-8 tekstualnom formatu. Vraća objekat tipa <code>StreamWriter</code> .
<code>OpenRead(string nazivDat)</code>	Otvara postojeću datoteku za čitanje. Vraća objekat tipa <code>FileStream</code> .
<code>OpenText(string put/nazivDat)</code>	Otvara za čitanje postojeću UTF-8 tekstualnu datoteku. Vraća objekat tipa <code>StreamReader</code> .
<code>OpenWrite(string nazivDat)</code>	Otvara za pisanje postojeću ili kreira novu datoteku. Vraća objekat tipa <code>FileStream</code> .

Tabela 11.3: Metode klase `File` za kreiranje, otvaranje, brisanje, pomjeranje, zamjenu

RAZVOJ PROGRAMSKIH RJEŠENJA

File klasa sadrži i metode za čitanje, pisanje, dodavanje. Pregled dijela tih metoda je u tabeli 11.4.

Metoda	Opis metode
ReadAllBytes(string nazivDat)	Otvara binarnu datoteku, čita sadržaj datoteke u niz i poslije zatvara datoteku.
ReadAllLines(string nazivDat, [Encoding kodniStandard]) ReadAllText(string nazivDat, [Encoding kodniStandard])	Otvara datoteku, čita sve linije (cjelokupni tekst) datoteke sa navedenim kodnim standardom i poslije zatvara datoteku.
WriteAllBytes(string nazivDat, byte[] brojBajta)	Kreira novu datoteku, piše specificirani niz bajta u datoteku i poslije zatvara datoteku.
WriteAllLines(string nazivDat, string[] nizStringova, Encoding kodniStandard) WriteAllText(string, string, Encoding kodniStandard)	Kreira novu datoteku, piše specificirani niz stringova (ili tekst) u datoteku korištenjem navedenog kodnog standarda i poslije zatvara datoteku.

Tabela 11.4: Metode klase `File` za čitanje i pisanje

Pored tipičnih operacija za koje se koristi `File` klasa datih u tabelama iznad, `File` klasa može se koristiti za postavljanje i dobijanje atributa datoteke ili vremenskih informacija vezanih za kreiranje, pristup i pisanje u datoteku (`GetAccess()`, `GetAttributes()`, `GetCreationTime()`, `GetLastAccessTime()`, `GetLastWriteTime()`, `SetAttributes()`, `SetLastAccessTime()`, ...).

Mnoge `File` metode (npr. `Open`, `Create`) vraćaju objekte tipa `FileStream`, `StreamWriter`, `StreamReader`. Objekti tih tipova koristi se za daljnju manipulaciju sa datotekama (objašnjeno u sekiji 11.4).

11.3.2. `Directory` klasa za rad sa direktorijima

`Directory` klasa koristi se za izvršavanje raznih operacija nad direktorijem. Ova klasa podržava kreiranje, pomjeranje, brisanje direktorija i poddirektorija. Tabela 11.5 sadrži opis dijela metoda ove klase. Većina `Directory` metoda zahtijeva naziv direktorija i put do direktorija kojim se manipulira.

`Directory` klasa pruža i metode za dobijanje i postavljanje vremenskih informacija za kreiranje, pristup i pisanje u direktorij (`GetLastAccessTime()`, `GetCreationTime()`, `GetLastWriteTime()`, `SetLastAccessTime()`, `SetCreationTime()`, ...).

Metoda	Opis metode
CreateDirectory (string nazivDirektorija, [DirectorySecurity zastitaDirektorija])	Kreira specificirani direktorij. Opcionalni parametar se odnosi na postavku Windows zaštite.
Delete (string nazivDirektorija, [Boolean podDirektoriji])	Briše specificirani direktorij (i poddirektorije, ako je sa drugim opcionalnim parametrom naznačeno).
1.EnumerateDirectories (string nazivDirektorija, [string paternPretrazivanja], [SearchOption opcijePretrazivanja]) 2.EnumerateFiles (string nazivFajlova, [opcionlani parametri kao za metodu 1]) 3.EnumerateFileSystemEntries (string nazivFajlovaIDirektorija, string, [opcionlni parametri kao za metodu 1])	Vraća kolekciju naziva 1.direktorija, 2.datoteka 3.datoteka i direktorija. Drugi opcionalni parametar se odnosi na opcije pretraživanja a trećim se specificira da li se pretražuju poddirektoriji.
Exists (string nazivDirektorija)	Određuje da li naznačeni direktorij postoji na disku.
Move (string izvorisniNazivDirektorija, string odredisniNazivDirektorija)	Pomjera izvorišni direktorij i njegov sadržaj na novu lokaciju.

Tabela 11.5: Osnovne metode za rad sa direktorijima

Pored navedenih metoda u tabeli 11.5 izdvajaju se i `get` metode koje omogućavaju dobijanje informacija o tekućem direktoriju (`GetCurrentDirectory()`), o poddirektorijima (`GetDirectories()`), korjenu i logičkom drajvu direktorija (`GetDirectoryRoot()`, `GetLogicalDrivers()`), datotekama na disku `GetFiles()`, `GetFileSystem()`).

Pored `File` i `Directory` klase za rad sa datotekama i direktorijima, postoje još i klase `FileInfo` i `DirectoryInfo`. Ove klase imaju većinu metoda kao i klase `File` i `Directory`. U ovom materijalu neće se detaljnije izlagati osobine i metode ovih klasa.

11.3.3. Primjer upotrebe klasa za rad sa datotekama i direktorijima

Sljedeći primjer koda 11.1 provjerava da li `PocDir` direktorij postoji. Ako `PocDir` postoji, direktorij se pomjera u drugi direktorij (`TestDir`). Nakon toga, se u `TestDir` direktoriju kreiraju dvije dodatne (test.txt, test1.txt) datoteke i očitava se ukupni broj datoteka. Na kraju se formira kolekcija naziva datoteka i njihova imena se ispisuju na ekran.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
using System.IO;
using System.Collections.Generic;

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

            // Specificiranje naziva direktorija.
            String pocetniDirektorij = "c:\\\\PocDir";
            String odredisniDirektorij = "c:\\\\TestDir";
            try
            {
                // Određuje da li početni direktorij postoji korištenjem metode Exists
                if (!Directory.Exists(pocetniDirektorij))
                {
                    throw new DirectoryNotFoundException("Ne postoji pocetni direktorij PocDir");
                    // ako direktorij ne postoji izuzetak
                }

                // Određuje da li odredišni direktorij postoji korištenjem metode Exists
                if ( Directory.Exists(odredisniDirektorij) )
                {
                    // Brisanje odredišnog direktorija ako već postoji, da se osigura da ne postoji
                    Directory.Delete(odredisniDirektorij, true );
                }

                // Pomjera direktorij PocDir u TestDir
                Directory.Move(pocetniDirektorij,odredisniDirektorij);

                // Kreira dodatnu datoteku test.txt u odredišnom direktoriju
                File.CreateText( String.Concat(odredisniDirektorij, "\\\\test.txt" ) );

                // Kreira datoteku test1.txt i u nju upisuje tekst
                File.WriteAllText( String.Concat(odredisniDirektorij, "\\\\test1.txt" ),"Rad sa datotekama");

                // Broji datoteke u odredišnom direktoriju
                Console.WriteLine( "Broj datoteke u direktoriju {0} je {1}", odredisniDirektorij,
                Directory.GetFiles(odredisniDirektorij).Length );

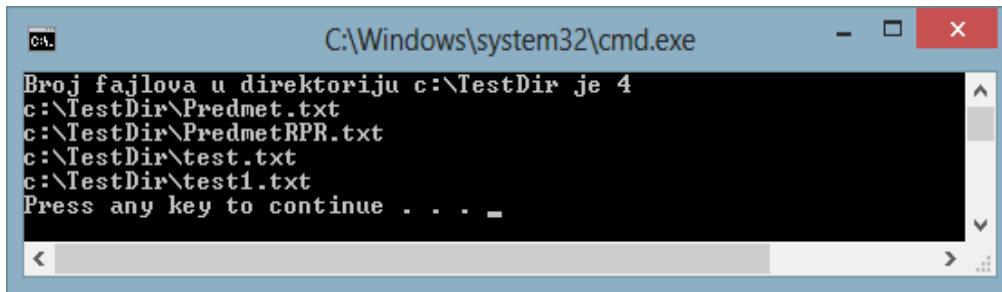
                // formira listu datoteka u direktoriju - sa Directory.EnumerateFiles(odredisniDirektorij)
                List<string> datoteke = new List<string>(Directory.EnumerateFiles(odredisniDirektorij));

                // ispisuje imena datoteka
                foreach (var dat in datoteke)
                {
                    Console.WriteLine("{0}", dat);
                }
            }
            catch (DirectoryNotFoundException e)
            {
                Console.WriteLine( "Izuzetak {0}", e );
            }
            catch ( IOException e )
            {
                Console.WriteLine( "Izuzetak {0}", e );
            }
        }
    }
}
```

Kod 11.1: Primjena `File` i `Directory` klase

Scenarij izvršavanja:

Na disku je postojao direktorij `PocDir` sa dvije datoteke `Predmet.txt` i `PredmetRPR.txt`.



The screenshot shows a Windows Command Prompt window titled "C:\Windows\system32\cmd.exe". The window displays the following text:
Broj fajlova u direktoriju c:\TestDir je 4
c:\TestDir\Predmet.txt
c:\TestDir\PredmetRPR.txt
c:\TestDir\test.txt
c:\TestDir\test1.txt
Press any key to continue

Sadržaj direktorija TestDir

11.4. Rad sa generičkim klasama za ulaz i izlaz

Mnoge metode `File` klase vraćaju objekte koji predstavljaju neki *stream*. Naprimjer, za vrijeme izvršavanja programa kada se datoteka kreira i otvori sa `File.Create()` metodom, kreira se objekt `FileStream` klase i *stream* se povezuje sa datotekom. Već je naglašeno da je *stream* apstrakcija sekvence bajta, koja se odnosi na datoteku, ulazno/izlazni uređaj. U okviru .NET okruženja apstraktna bazna klasa `Stream` i na osnovu nje izvedene klase pružaju generički pogled na različite tipove ulaza i izlaza, i izoliraju programera od specifičnih detalja operativnog sistema i uređaja.

Streamovi podržavaju tri osnovne operacije:

- Čitanje: Čitanje podataka različitih formata iz *streama* u određene strukture podataka. Za *streamove* koji podržavaju čitanje postoje u ovisnosti od tipa *streama* razne `Read` metode za čitanje koje omogućavaju čitanje teksta, stringova, binarnih podataka.
- Pisanje: Pisanje podataka iz programske strukture podataka u *stream*. Za *streamove* koji podržavaju pisanje postoje u ovisnosti od tipa *streama* razne `Write` metode za pisanje koje omogućavaju upis teksta, stringova, binarnih podataka.
- Pomjeranje (*seeking*): Odnosi se na pointer koji označava tekuću poziciju *streama*. Za *streamove* koji podržavaju pomjeranje, koristi se `Seek` metoda i `Position` osobina za očitavanje i modifikaciju tekuće pozicije *streama*.

U okviru ovog teksta objašnjava se `FileStream` klasa.

RAZVOJ PROGRAMSKIH RJEŠENJA

11.4.1. `FileStream` objekat

`FileStream` objekat predstavlja *stream* koji pokazuje na datoteku na disku. `FileStream` klasa se koristi za čitanje iz, pisanje u, otvaranje, zatvaranje datoteke. `FileStream` klasa povezuje datoteku i program.

Postoji nekoliko preklopljenih konstruktora za kreiranje objekata `FileStream` klase koji dozvoljavaju da se kombinira sljedeće: naziv datoteke, način (mod) koji određuje kako će datoteka biti kreirana i/ili otvorena (`FileMode` lista vrijednosti), zahtijevani tip pristupa (`FileAccess` lista vrijednosti), način dijeljenja datoteke (`FileSharing` lista vrijednosti).

Mogući načini kreiranja i otvaranja datoteke, sadržani u `FileMode` listi vrijednosti su: `Append` (ako specificirana datoteka postoji, otvara se, suprotno kreira se nova datoteka, dodavanje sadržaja se vrši na kraj datoteke), `Create` (ako specificirana datoteka već postoji otvara se i uništava se njen sadržaj, inače kreira se nova datoteka), `CreateNew` (ako specificirana datoteka već postoji javlja se izuzetak, inače kreira se nova datoteka), `Open` (otvara postojeću datoteku), `OpenOrCreate` (otvara postojeću datoteku ili kreira novu), `Truncate` (otvara postojeću datoteku, javlja se izuzetak ako ne postoji datoteka, ako postoji uništava se njen sadržaj).

Mogući pristupi datoteci opisani sa `FileAccess` listom vrijednosti su: `Read` (datoteka je namijenjena za čitanje), `ReadWrite` (datoteka je namijenjena za čitanje i pisanje, podrazumijevana vrijednost), `Write` (datoteka je namijenjena za pisanje).

Dijeljenje datoteke sa drugim procesima je specificirano `FileShare` listom vrijednosti: `None` (Bez dijeljenja datoteke), `Deleting` (Dovoljeno sekvencijalno brisanje datoteke), `Inheritable` (Dijete proces nasljeđuje prava pristupa), `Read` (Dijeljeni pristup čitanju), `ReadWrite` (Dijeljeni pristup za čitanje/pisanje), `Write` (Dijeljeni pristup za pisanje).

Sljedećim iskazom se kreira datoteka sa nazivom `datoteka2.txt` za čitanje i pisanje sa dozvoljenim dijeljenim čitanjem:

```
FileStream fs1 = new FileStream("datoteka2.txt", FileMode.Create,  
                                FileAccess.ReadWrite, FileShare.Read);
```

`File` i `FileInfo` klase imaju metode `OpenRead()` i `OpenWrite()` koje omogućavaju jednostavnije kreiranje `FileStream` objekata. Prva otvara datoteku za čitanje, druga za pisanje. Ove metode ne zahtijevaju specifikaciju svih parametara koje zahtjeva `FileStream` konstruktor. Datoteka za čitanje se može otvoriti sa:

```
FileStream fs2 = File.OpenRead("datoteka2.txt");
```

Pomjeranje (`seek`)

Za svaki `FileStream` objekat povezan je interni datotečni pointer (*file pointer*) koji pokazuje na lokaciju unutar datoteke sa koje će početi sljedeće čitanje ili pisanje. U većini slučajeva, kada se datoteka otvori, pointer pokazuje na početak datoteke, ali ovaj pointer se može pomijerati. Pointer se automatski repozicionira pri upotrebi `FileStream` metoda za čitanje i pisanje. Moguće je da se pointer pomjera i direktno sa `Seek` metodom, čime je omogućen slučajni (*random*) pristup datoteci i sposobnost direktnog skoka na specifičnu lokaciju datoteke. To može uštediti mnogo vremena kada se radi sa velikim datotekama.

Repozicioniranje pointera se radi korištenjem `Seek` metode. Metoda ima dva parametra. Prvi parametar je broj bajta za koji je potrebno izvršiti pomjeranje pointera. Taj broj može biti pozitivan sa značenjem da se pomjeranje vrši prema kraju datoteke ili negativan sa značenjem da se pomjeranje vrši prema početku datoteke. Drugi parametar je pozicija od koje je potrebno izvršiti pomjeranje za dati broj bajta. Moguće pozicije su tri člana `SeekOrigin` liste vrijednosti: `Current` (pomjeranje se vrši od tekuće pozicije pointera), `Begin` (pomjeranje se vrši od početka datoteke), `End` (pomjeranje se vrši od kraja datoteke).

Npr sa iskazom: `fp.Seek(8, SeekOriginBegin);` vrši se pomjeranje pointera `fp` za 8 bajta od početka datoteke, odnosno od prvog bajta u datoteci.

Sa iskazom: `fp.Seek(2, SeekOriginCurrent);` vrši se pomjeranje pointera `fp` za 2 bajta od tekuće pozicije prema kraju datoteke.

Sa iskazom: `fp.Seek(-5, SeekOrigin.End);` vrši se pomjeranje pointera `fp` za 5 bajta od kraja datoteke prema početku.

`StreamReader` i `StreamWriter` klase koje su opisane u 11.4.2 pristupaju datoteci sekvencijalno i ne dozvoljavaju da se na ovaj način manipulira sa pointerom.

Čitanje i pisanje podataka

`FileStream` objekti podržavaju sinhrone i asinhrone operacije čitanja i pisanja. Asinhrone metode se koriste za izvršavanje resursno zahtjevnih operacija nad datotekama bez blokiranja glavne niti (poglavlje 13). Osnovne metode za čitanje i pisanje pomoću `FileStream` klase su date u tabeli 11.6.

RAZVOJ PROGRAMSKIH RJEŠENJA

Metoda	Opis metode
Read (byte[] nizBajta, int offset, int count)	Ova metoda čita podatke iz datoteke i poslije ih piše u niz bajta (prvi parametar). Sa drugim parametrom se naznačava pozicija u nizu bajta od koje se smještaju pročitani podaci. Obično je inicijalna vrijednost 0 da bi se počelo smještanje podataka na početak niza. Treći parametar je broj koji specificira koliko bajta se čita iz datoteke.
ReadByte ()	Čita bajt iz datoteke i pomjera poziciju datotečnog pointera za jednu poziciju (jedan bajt).
Write (byte[] nizBajta, int offset, int count)	Formira niz bajta koji se upisuje u datoteku. Značenje drugog i trećeg parametra kao kod Read metode.
WriteByte ()	Piše bajt na tekuću poziciju u <i>streamu</i> .

Tabela 11.6: Metode za čitanje iz i pisanje u objekte tipa `FileStream`

Iako, `FileStream` klasa nudi metode za čitanje iz i pisanje u datoteku češće za tu namjenu se koriste `StreamReader` i `StreamWriter` klase. To je zbog toga što `FileStream` klasa radi nad bajtimi i nizovima bajta dok `Stream` klase rade nad karakternim podacima. Jednostavnije je raditi nad karakternim podacima, ali neke operacije kao što je slučajni pristup datoteci može se izvršiti samo sa `FileStream` objektom.

Pored iznad navedenih metoda za `FileStream` klasu, za *streamove* su bitne metode koje zatvaraju *streamove* i oslobađaju resurse zauzete tokom rada sa *streamom*. Te metode su prikazane u tabeli 11.7.

Naziv	Opis
<code>Close()</code>	Zatvara tekući <i>stream</i> i oslobađa sve resurse vezane za <i>stream</i> .
<code>Dispose()</code>	Oslobađa sve resurse korištene sa <i>streamom</i> .
<code>Flush()</code>	Čisti bafere korištene od <i>streama</i> i uzrokuje da se svi baferovani podaci zapisuju u datoteku.

Tabela 11.7: Značenje `Close`, `Dispose`, `Flush` metoda

Ukoliko se otvaranje datoteke i formiranje objekta `FileStream` klase vrši u okviru `using` direktive, tada se automatski izvršava zatvaranje datoteke i oslobađanje resursa. Upotreba `using` metode i rad sa `Read`, `Write`, `Seek` metodama `FileStream` klase data je u primjeru ispod.

RAZVOJ PROGRAMSKIH RJEŠENJA

Primjer: Potrebno je pročitati sadržaj datoteke c:\testDir\test1.txt, kreirati datoteku c:\testDir\novadat.txt i u nju upisati pročitani sadržaj iz prve datoteke. Nakon toga potrebno je pročitati zadnjih 10 karaktera datoteke novadat.txt i ispisati ih na ekran. Kod 11.2 realizira postavljeni zadatak.

```
using System;
using System.IO ;
using System.Text;
namespace Datoteka3
{
    class Program
    {
        static void Main(string[] args)
        {
            string pocetnaDat = @"c:\testDir\test1.txt"; //specificira naziv datoteke iz koje će se čitati
            string novaDat = @"c:\testDir\novadat.txt"; // specificira naziv datoteke koja će se kreirati

            try
            {
                // otvara datoteku c:\testDir\test1.txt za čitanje, povezuje se datoteka i stream
                using (FileStream fsPocetnaDat = new FileStream(pocetnaDat, FileMode.Open, FileAccess.Read))
                {
                    byte[] bafer = new byte[fsPocetnaDat.Length]; // kreira niz bajta

                    // specificira maksimalan broj bajta za čitanje, u ovom slučaju cijelokupna veličina
                    // može se specificirati i manji broj pa sucesivno čitati određen broj karaktera iz datoteke
                    // pri tome treba voditi računa i o drugom parametru Read metode
                    int maxBrojBajtaZaCitanje = (int)fsPocetnaDat.Length;

                    // čita iz fajla maksimalno specificiran broj karaktera
                    int brojProcitanihBajta = fsPocetnaDat.Read(bafer, 0, maxBrojBajtaZaCitanje);

                    // Piše niz bajta u drugi FileStream.
                    using (FileStream fsNovi = new FileStream(novaDat,
                        FileMode.Create, FileAccess.Write))
                    {
                        fsNovi.Write(bafer, 0, brojProcitanihBajta);
                    }
                    // Čita zadnjih 10 karaktera iz datoteke novadat.txt i prikazuje ih na ekran
                    using (FileStream fsCit = new FileStream(novaDat,
                        FileMode.Open, FileAccess.Read))
                    {
                        char[] charPodaci = new char[10];

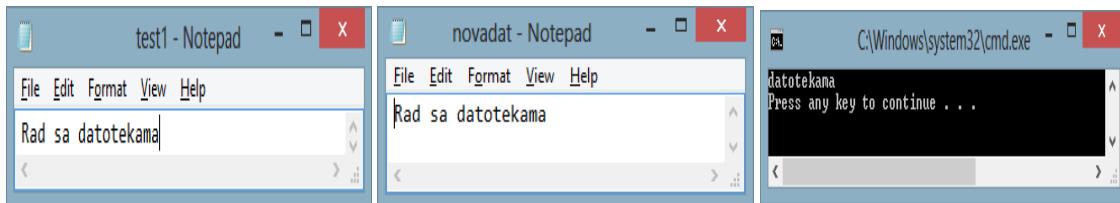
                        // pomjera se pointer na deseti karakter od kraja datoteke
                        fsCit.Seek(-10, SeekOrigin.End);
                        // čita zadnjih 10 karaktera kao bajte u bafer
                        fsCit.Read(bafer, 0, 10);

                        // Dekodira pročitane bajte u karaktere i smješta ih u niz karaktera - charPodaci
                        Decoder d = Encoding.UTF8.GetDecoder();
                        d.GetChars(bafer, 0, 10, charPodaci, 0);

                        // Ispisuje zadnjih 10 karaktera iz datoteke na ekran
                        Console.WriteLine(charPodaci);
                    }
                }
            }
            catch (FileNotFoundException ioEx)
            {
                Console.WriteLine(ioEx.Message);
            }
        }
    }
}
```

Kod 11.2: FileStream klasa: kreiranje, čitanje, pisanje, pomjeranje i ispis dekodiranih bajta na ekran

Mogući scenarij izvršavanja:



Postojeća datoteka test1.txt, kreirana datoteka novadat.txt, prikaz 10 zadnjih karaktera

11.4.2. StreamReader i StreamWriter klase

StreamReader i **StreamWriter** objekti omogućavaju jednostavnije čitanje i pisanje u *stream* objekte. Sa klasama **StreamReader** i **StreamWriter** omogućeno je čitanje i pisanje karaktera i stringova u datoteku. **FileStream** klasa koja je uvedena u sekciji 11.4.1. čita i piše nizove bajta. Ukoliko je dovoljan sekvencijalni pristup datoteci tada se preporučuje upotreba klase **StreamReader** i **StreamWriter** za čitanje iz i pisanje u datoteku.

StreamWriter klasa

StreamWriter klasa ima dosta korisnih metoda za pisanje karaktera i stringova u datoteku. Postoji više načina da se kreira **StreamWriter** objekat. Ako već postoji **FileStream** objekat, taj objekat se može iskoristiti da se kreira **StreamWriter**.

```
FileStream logDat = new FileStream("Log.txt", FileMode.CreateNew);  
StreamWriter swLog=new StreamWriter(logDat)
```

StreamWriter objekat može se kreirati direktno iz datoteke, pri čemu konstruktor prima naziv datoteke i Boolean vrijednost (true ili false). Sa true se označava da ako postoji specificirana datoteka ista se otvara, a novi sadržaj se dodaje na kraj datoteke, sa false se naznačava da se kreira nova datoteka.

```
StreamWriter swLog=new StreamWriter("Log.txt",true) ;
```

U tabeli 11.8 su prikazane neke metode **StreamWriter** klase za pisanje u *stream* ili tekstualni string. Podržano je upisivanje tekstualne reprezentacije vrijednosti svih osnovnih tipova podataka.

RAZVOJ PROGRAMSKIH RJEŠENJA

Metoda	Opis
Write(Boolean)	Piše tekstualnu reprezentaciju Boolean vrijednosti.
Write(Char), Write(Char[])	Piše karakter(e).
Write(Decimal)	Piše tekstualnu reprezentaciju decimalne vrijednosti.
Write(Single), Write(Double)	Piše tekstualnu reprezentaciju <i>floating-point</i> vrijednosti.
Write(Int32), Write(Int64) Write(UInt32), Write(UInt64)	Piše tekstualnu reprezentaciju cijelobrojne označene ili neoznačene vrijednosti.
Write(String)	Piše string.
Write(Object)	Piše tekstualnu reprezentaciju objekta pozivanjem <code>ToString</code> metode nad objektom.
WriteLine()	Piše karakter nove linije.
WriteLine(Boolean), WriteLine(Decimal), WriteLine(Single), ...	Piše tekstualnu reprezentaciju vrijednosti koju prima kao parametar i karakter nove linije.

Tabela 11.8: Metode `StreamWriter` klase

Primjer: Potrebno je kreirati dvije datoteke test1.txt i test2.txt i upisati u njih sadržaj. Kod 11.3. ilustrira pisanje u datoteke korištenjem `StreamWriter` klase.

```

using System;
using System.IO ;
namespace PisanjeStreamWriter
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
// I: Varijanta kreiranja objekta StreamWriter klase i pisanja u datoteku test1.txt
                //1.kreira FileStream
                FileStream fs = new FileStream(@"c:\testDir\test1.txt", FileMode.Create);

                // 2.kreira objekat swriter1 klase StreamWriter
                StreamWriter swriter1 = new StreamWriter(fs);

                // 3.piše tekst-dvije linije u swriter1
                swriter1.WriteLine("Razvoj programskih rješenja ");
                // upisuje se tekst i tekstualna reprezentacija broja 100
                swriter1.WriteLine("- broj studenata {0} ", 100);
                swriter1.WriteLine("Objektno orijentisana analiza i dizajn");

                // 4. čisti bafer i zatvara fajl
                swriter1.Flush();
                swriter1.Close();

// druga varijanta na sljedećoj stranici
            }
        }
    }
}

```

RAZVOJ PROGRAMSKIH RJEŠENJA

```
// II: Varijanta kreiranja objekta StreamWriter klase (swriter2) u using bloku i upis u datoteku test2.txt
    using (StreamWriter swriter2 = new StreamWriter(@"c:\testDir\test2.txt")
    {
        swriter2.WriteLine("Data Mining - ");
        swriter2.WriteLine("prosjek {0}", 9.8);
    }
}
catch (IOException pe)
{
    Console.WriteLine(pe.ToString());
}
```

Kod 11.3: Primjena StreamWriter klase

StreamReader klasa

Ulagni *streamovi* se koriste za čitanje podataka sa eksternog uređaja. Često, to je datoteka na disku ali izvor može biti sve što šalje podatke, kao što je mrežna aplikacija ili čak konzola. StreamReader klasa je jedna od klase koja se može koristiti za čitanje podataka iz datoteka. StreamReader klasa je generička klasa kao i StreamWriter, i može se koristiti sa bilo kojim *streamom*. U tabeli 11.9 date su značajnije metode ove klase.

Metoda	Opis
StreamReader(Stream)	Inicijalizira novu instancu StreamReader klase za specificirani <i>stream</i> .
Peek()	Vraća cijelobrojnu reprezentaciju sljedećeg karaktera za čitanje ili -1 ako nema karaktera za čitanje. Može se koristiti za provjeru da li je kraj datoteke.
Read()	Čita sljedeći karakter iz ulaznog <i>streama</i> . Vraća cijelobrojnu reprezentaciju pročitanog karaktera koja se može pretvoriti u karakter korištenjem Convert klase ili -1 ako string nije pročitan.
Read(char[] bufer , Int32 pozicija, Int32 brojKaraktera)	Čita maksimalno specificirani broj karaktera iz tekućeg <i>streama</i> u bafer, počevši od navedenog indeksa.
ReadLine()	Čita liniju karaktera iz tekućeg <i>streama</i> , vraća pročitane podatke kao string.
ReadToEnd()	Čita sve karaktere od tekuće pozicije do kraja <i>streama</i> .

Tabela 11.9: Metode StreamReader klase

Kodom 11.4 se čitaju datoteke *c:\testDir\test1.txt* i *c:\testDir\test2.txt* u koje je tekst upisan sa kodom 11.3. koji je ilustrirao primjenu StreamWriter klase. U kodu 11.4 se demonstrira primjena StreamReader klase.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
using System.IO;
namespace CitanjeStreamReader
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                // I: Čitanje iz datoteke c:\testDir\test1.txt
                //1.kreiraje stream instance za datoteku c:\testDir\test1.txt
                FileStream fs = new FileStream(@"c:\testDir\test1.txt", FileMode.Open);

                //2.kreiranje StreamReader objekta (sreader1)
                StreamReader sreader1 = new StreamReader(fs);

                Console.WriteLine("Pročitani tekst iz datoteke test1.txt");
                string linija;

                //3. Čitanje iz streama korištenjem ReadLine metode
                while ((linija = sreader1.ReadLine()) != null)
                {
                    Console.WriteLine(linija);
                }
                sreader1.Close(); // zatvaranje čitača/readera

                // II: I: Čitanje iz datoteke c:\testDir\test2.txt
                Console.WriteLine("\nPročitani tekst iz datoteke test2.txt");

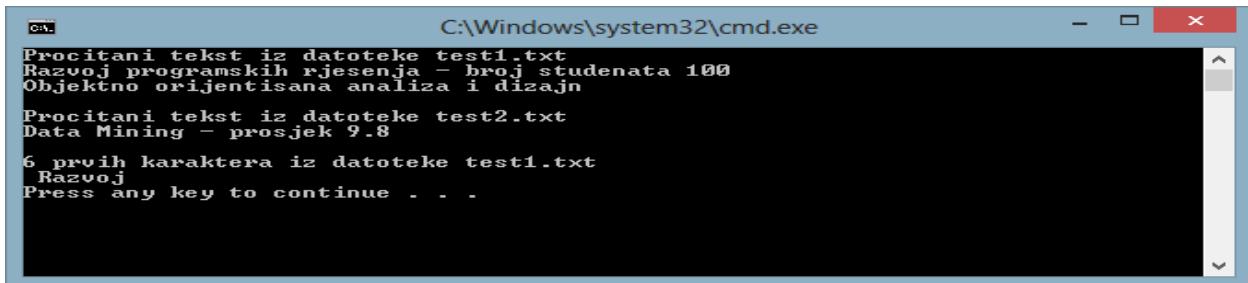
                //1. u okviru using bloka kreiranje StreamReader objekta za datoteku c:\testDir\test2.txt
                using (StreamReader sreader2 = File.OpenText(@"c:\testDir\test2.txt"))
                {
                    //2. Čitanje iz streama korištenjem ReadLine metode
                    while ((linija = sreader2.ReadLine()) != null)
                    {
                        Console.WriteLine(linija);
                    }
                }

                // III: čitanje prvih 6 karaktera iz c:\testDir\test1.txt
                Console.WriteLine("\n6 prvih karaktera iz datoteke test1.txt");

                //1. u okviru using bloka kreiranje StreamReader objekta za datoteku c:\testDir\test2.txt
                using (StreamReader sreader3 = File.OpenText(@"c:\testDir\test1.txt"))
                {
                    char[] buffer = new char[10];
                    //2. Čitanje iz streama prvih 6 karaktera korištenjem Read metode
                    sreader3.Read(buffer, 1, 6);
                    Console.WriteLine(buffer);
                }
            }
            catch (FileNotFoundException)
            {
                Console.WriteLine("Izuzetak: Ne postoji datoteka");
            }
            catch (EndOfStreamException)
            {
                Console.WriteLine("Izuzetak: Čitanje nakon kraja streama");
            }
            catch (IOException pe)
            {
                Console.WriteLine("IO Izuzetak: {0} ", pe.ToString());
            }
        }
    }
}
```

Kod 11.4: Čitanje iz datoteke/streama pomoću StreamReader klase

Scenarij izvršavanja:



Pročitani sadržaj test1 i test2 datoteka

U iznad prikazanom programskom kodu navedeni su i neki izuzeci. Lista specifičnih izuzetaka koja se može desiti nije u potpunosti obuhvaćena.

11.4.3. Binarno pisanje i čitanje

Namjena binarnog ulaznog i izlaznog *streama* je čitanje i pisanje binarnih vrijednosti. `BinaryReader` klasa sadrži metode koje podržavaju binarno čitanje svih osnovnih tipova podataka. Naprimjer, `ReadBoolean` metoda čita sljedeći bajt kao `Boolean` vrijednost, `ReadChar` čita sljedeći bajt kao karakternu vrijednost i pomjeraju tekuću poziciju u *streamu* za jedan bajt. Sa `Read()` metodama može se čitati specificirani broj bajta iz *streama* i smještati u niz bajta od naznačene pozicije. Tu su i metode `ReadDecimal()`, `ReadDouble()`, `ReadInt32()`, `ReadSingle()`, `ReadString()` i još mnoge metode za svaki osnovni tip podataka.

`BinaryWriter` klasa nudi metode za binarno pisanje osnovnih tipova podataka. Neke od tih metoda su: `Write(Int64)` koja upisuje osmo-bajtnu označenu cjelobrojnu vrijednost u *stream* i pomjera tekuću *stream* poziciju za osam bajta, `Write(Char)` koja piše karakter u *stream* i pomjera tekuću *stream* poziciju u skladu sa upisanim karakterom i kodnim standardom. I za sve ostale tipove podataka postoje specijalizirane metode za pisanje.

Pored metoda za čitanje i pisanje `BinaryWriter` ima i metodu `Seek` čime je omogućeno direktno pomjeranje u *streamu* na određenu poziciju. Parametri `Seek` metode su već objašnjeni u 11.4.1. Kada se kreira instanca `BinaryReader` i `BinaryWriter` klase obavezan parametar koji se predaje konstruktoru je naziv *streama*.

Primjer koda 11.5 ilustrira primjenu `BinaryReader` i `BinaryWriter` klase. U datoteku `c:\testDir\brojevi.dat` potrebno je upisati 5 nasumičnih binarnih vrijednosti. Nakon toga se te vrijednosti čitaju iz datoteke i prikazuju na standardni izlaz.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
using System.IO;
namespace DatotekaBinary
{
    class Program
    {
        static void Main(string[] args)
        {
            string put = @"c:\testDir\brojevi.dat";
            try
            {
                // poziv metode za binarno pisanje, parametri put/naziv datoteke i broj vrijednosti za upis
                pisiBinaryWriter(put,5);

                // poziv metode za binarno čitanje, parametri put/naziv datoteke i broj vrijednosti za čitanje
                citajBinaryReader(put,5);
            }
            catch (EndOfStreamException)
            {
                Console.WriteLine("EndOfStreamException ");
            }
            catch (IOException ep)
            {
                Console.WriteLine("Exception ",ep.Message);
            }
        }
    }
}
```

```
// Metoda za binarno pisanje, parametri put/naziv datoteke i broj vrijednosti za upis
public static void pisiBinaryWriter(string dat, int brojEl)
{
    FileStream fileStream = File.OpenWrite(dat); // kreira se instanca streama za pisanje
    Random random = new Random();

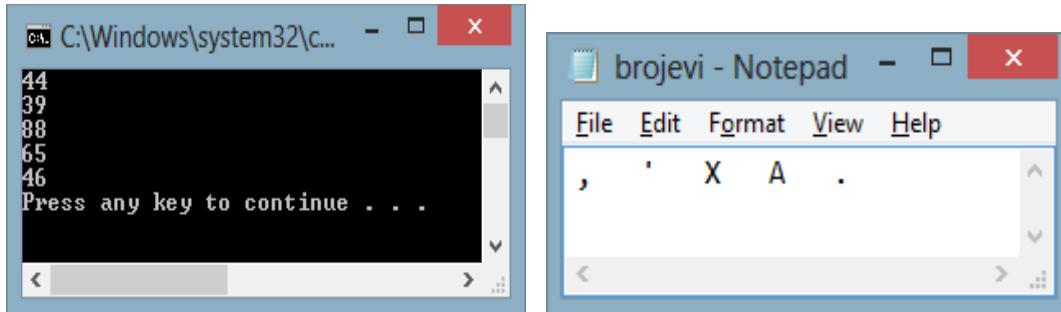
    // kreira i povezuje objekat (pisac) BinaryWriter klase sa streamom
    // piše u pisac i zatvara stream
    using (BinaryWriter pisac = new BinaryWriter(fileStream))
    {
        for (int i = 0; i < brojEl; i++)
        {
            pisac.Write(random.Next(0, 100));
        }
    }
    fileStream.Close(); // zatvara stream
}

// Metoda za binarno čitanje , parametri put/naziv datoteke i broj vrijednosti za čitanje
public static void citajBinaryReader(string dat, int brojEl)
{
    FileStream fileStream = File.OpenRead(dat); // kreira se instanca streama za čitanje

    // kreira i povezuje instancu citac BinaryReader klase sa streamom, čita i zatvara BinaryReader
    using (BinaryReader citac = new BinaryReader(fileStream))
    {
        for (int i = 0; i < brojEl; i++)
        {
            int broj = citac.ReadInt32();
            Console.WriteLine(broj);
        }
    }
    fileStream.Close();
}
}
```

Kod 11.5: Pisanje slučajnih vrijednosti u datoteku korištenjem `BinaryWriter` klase i čitanje iz datoteke korištenjem `BinaryReader` klase

Scenarij izvršavanja:



Prikaz upisanih brojeva u datoteku pročitanih sa `BinaryReader`, sadržaj datoteke `brojevi.dat`

11.5. Serijalizacija i deserijalizacija objekata

Logički dizajn objektno orientiranih programske rješenja zasnovan je na klasama. Često je potrebno u okviru programskog rješenja upisati vrijednost instance klase odnosno stanje objekta u datoteku. Potrebno je i pročitati stanje objekta. Metode za pisanje i čitanje koje su do sada uvedene upisivale su, odnosno čitale pojedinačne podatke u/iz datoteke. Spašavanje stanja objekta u datoteku naziva se binarna serijalizacija. Rekonstrukcija stanja objekta čitanjem informacija smještenih na disku u objekat klase naziva se binarna deserijalizacija.

Potrebno je eksplicitno naznačiti da se objekat serijalizira. Eksplicitna serijalizacija se postiže tako da se iznad klase na osnovu koje se instancira objekat postavlja `[Serializable]` atribut. To znači da se podaci objekta, odnosno stanje objekta serijalizira.

Npr. Ukoliko će biti potrebno serijalizirati objekte klase `Proizvod` tada se klasa označava sa `[Serializable]` atributom.

```
[Serializable]
public class Proizvod
{
    public int sifra;
    public string naziv;
    public double cijena;
    string napomena;

    ....
}
```

Ukoliko nije potrebno da se neki član klase serijalizira tada se iznad tog atributa postavlja atribut `[NonSerialized]`. Npr. ako za klasu `Proizvod` nije potrebno serijalizirati polje napomene tada se izvršava sljedeće obilježavanje:

RAZVOJ PROGRAMSKIH RJEŠENJA

```
[Serializable]
public class Proizvod
{
    public int sifra;
    public string naziv;
    public double cijena ;

    [NonSerialized]
    string napomena ;

    ....
}
```

Podrška za serijalizaciju objekata u okviru .NET okruženja nalazi se u `System.Runtime.Serialization` i `System.Runtime.Serialization.Formatters`. Klasa `BinaryFormatter` u `System.Runtime.Serialization.Formatters.Binary` implementira serijalizaciju u binarne podatke i deserijalizaciju iz binarnih zapisa u objekat. Metode za serijalizaciju i deserijalizaciju obezbjeđuje `IFormatter` interfejs.

Serijalizacija se obavlja sa metodom: `void Serialize(stream, objekt)` ;

Naprimjer, ukoliko je potrebno serijalizirati objekt klase `Proizvod` u datoteku `Proizvod.bin` sa kojom je povezan `datStream`, to se postiže sljedećim linijama koda:

```
IFormatter serializer = new BinaryFormatter();
serializer.Serialize(datStream, proizvod1);
```

Deserijalizacija se obavlja sa metodom: `objekt Deserialize(stream)`

```
IFormatter serializer = new BinaryFormatter();

Proizvod spaseniProizvod = (Proizvod)serializer.Deserialize(datStream);

ili

Proizvod spaseniProizvod = serializer.Deserialize(datStream) as Proizvod;
```

Slijedi programski kod 11.6 kojim se serijalizira jedan objekat klase `Proizvod` u datoteku `Proizvod.bin`. Nakon toga se deserijalizira objekat klase `Proizvod`. Stanje objekta se ispisuje na standardni izlaz.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace SerijalizacijaObjekta
{
    class Program
    {
        [Serializable]
        public class Proizvod
        {
            public int sifra;
            public string naziv;
            public double cijena;

            string napomena;

            public Proizvod(int psifra, string pnaziv, double pcijena, string pnapomena)
            {
                sifra = psifra;
                naziv = pnaziv;
                cijena = pcijena;
                napomena = pnapomena;
            }
            public override string ToString()
            {
                return string.Format("{0}, {1}, {2:F2}KM, {3}", sifra, naziv, cijena, napomena);
            }
        }

        static void Main(string[] args)
        {
            try
            {
                Proizvod proizvod1 = new Proizvod(1, "Coca Cola", 1.20, "Akcija");

                // Serijalizacija
                IFormatter serializer = new BinaryFormatter();
                FileStream dat = new FileStream(@"c:\TestDir\Proizvod.bin", FileMode.Create, FileAccess.Write);

                serializer.Serialize(dat, proizvod1);
                dat.Close();

                // Deserijalizacija
                FileStream cdat = new FileStream(@"c:\TestDir\Proizvod.bin", FileMode.Open, FileAccess.Read);
                Proizvod spaseniProizvod = (Proizvod)serializer.Deserialize(cdat);
                cdat.Close();

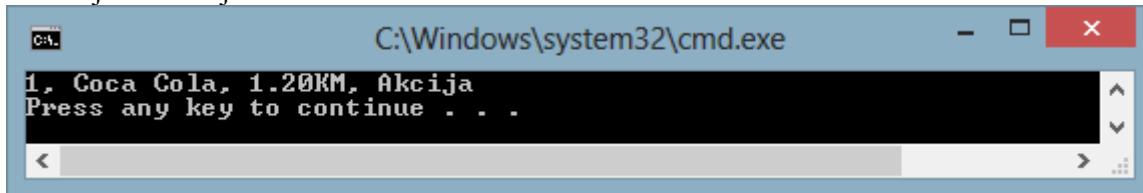
                Console.WriteLine(spaseniProizvod);

            }
            catch (SerializationException pe)
            {
                Console.WriteLine("Izuzetak pri serijalizaciji: {0}", pe.Message);
            }
            catch (IOException pe)
            {
                Console.WriteLine("IO izuzetak: {0}", pe.Message);
            }
        }
    }
}
```

Kod 11.6: Serijalizacija i deserijalizacija objekta

RAZVOJ PROGRAMSKIH RJEŠENJA

Scenarij izvršavanja:



Prilikom serijalizacije i deserijalizacije mogu se javiti specifični izuzeci koji se obrađuju sa `SerializationException` klasom.

Pored serijalizacije pojedinačnih objekata moguće je binarno serijalizirati i deserijalizirati i razne kolekcije podataka, kao što su npr. liste. Serijalizacija i deserijalizacije liste proizvoda se ilustrira kodom 11.7.

```
List<Proizvod> proizvodi = new List<Proizvod>() ;
proizvodi.Add(new Proizvod (1,"Coca Cola", 1.20, "Akcija")) ;
proizvodi.Add(new Proizvod (2,"Fanta",1.50, "Specijalna edicija")) ;

// Serijalizacija

IFormatter serializer = new BinaryFormatter() ;

FileStream dat = new FileStream(@"c:\TestDir\Proizvodi.bin", FileMode.Create, FileAccess.Write) ;
serializer.Serialize(dat, proizvodi) ;
dat.Close() ;

// Deserijalizacija

FileStream cdat =new FileStream(@"c:\TestDir\Proizvodi.bin", FileMode.Open, FileAccess.Read) ;
List<Proizvod> spaseniProizvodi = serializer.Deserialize(cdat) as List<Proizvod>;
cdat.Close();

foreach (Proizvod proizvod in spaseniProizvodi)
    Console.WriteLine(proizvod) ;
```

Kod 11.7: Dio koda koji ilustrira serijalizaciju i deserijalizaciju kolekcije proizvoda

Rezime:

U okviru programskih rješenja potrebno je vršiti trajno zapisivanje raznih vrsta podataka. Često to mogu biti razni šifarnici, konfiguracijski podaci, tekstualni podaci. Jedan od načina je zapisivanje podataka u datoteke. Mogu se spašavati i liste podataka, podaci objekata klase. Ukoliko se radi o većoj količini podataka tada se umjesto datoteka koriste baze podataka. U okviru poglavlja prikazane su klase sa rad sa binarnim i tekstualnim datoteka. Prikazan je i način binarne serijalizacije objekata u datoteke i način deserijalizacije objekata u klase.

Pitanja i zadaci za samostalan rad

1. Šta je datoteka?
2. Šta predstavlja *stream*?
3. Koja je veza datoteke i *streama*?
4. Navedite osnovne klase za rad sa datotekama i direktorijima smještenih u `System.IO`.
5. Navedite osnovne izuzetke koji se vežu za ulaz i izlaz.
6. Za koje operacije sa datotekama `File` klase sadrži metode?
7. Navedite metode `File` klase za kreiranje datoteke?
8. Navedite metode `File` klase za čitanje i pisanje u datoteku?
9. Koji tip objekta vraća većina metoda `File` klase?
10. Navedite osnovne metode `Directory` klase.
11. Navedite i objasnite tri osnovne operacije nad *streamovi*.
12. Objasnite šta predstavlja `FileStream` objekat.
13. Koje parametre sadrže konstruktori za kreiranje objekata `FileStream` tipa?
14. Koji su mogući načini za kreiranje i otvaranje datoteke sadržani u `FileMode` listi?
15. Koji su mogući načini pristupa datoteci sadržani u `FileAccess` listi?
16. Koji su mogući načini dijeljenja datoteke sadržani u `FileMode` listi?
17. Objasnite šta se postiglo sa iskazom:

```
FileStream fs1 = new FileStream("datoteka2.txt", FileMode.Create, FileAccess.ReadWrite,  
FileShare.Read);
```

18. Kojim operacijama se pomjera pointer datoteke?
19. Kojom metodom se vrši direktno pomjeranje i repozicioniranje pointera datoteke?
20. Navedite i objasnite parametre metode `Seek`.
21. Navedite metode za čitanje iz i pisanje u `FileStream` objekte.
22. Osim metoda za čitanje i pisanje koje su još bitne metode za `FileStream` objekte.
23. Šta omogućava `StreamReader` klasa?
24. Navedite neke od metoda `StreamReader` klase?
25. Šta omogućava `StreamWriter` klasa?
26. Navedite neke od metoda `StreamWriter` klase?
27. Za koji pristup čitanja i pisanja se koriste `StreamReader` i `StreamWriter` klase?
28. Koja je namjena binarnog ulaznog i izlaznog *streama*?
29. Koje klase se koriste za rad sa binarnim *streamom*?
30. Navedite neke metode `BinaryReader` i `BinaryWriter` klasa.
31. Da li `BinaryWriter` klasa ima `Seek` metodu?
32. Objasnite šta se postiže binarnom serijalizacijom.
33. Objasnite šta se postiže binarnom deserijalizacijom.
34. U okviru kojeg .NET *namespace* je podrška za binarnu serijalizaciju i deserijalizaciju?
35. Kako se obilježava klasa koju treba serijalizirati.

RAZVOJ PROGRAMSKIH RJEŠENJA

36. Ukoliko se klasa serijalizira, a neki atribut klase se ne želi serijalizirati kako se taj atribut obilježava?
37. Koji interfejs i koje metode se koristi za serijalizaciju i deserijalizaciju?
38. Napisati program kojim se postiže serijalizacija i deserijalizacija neke klase.
39. Kako se vrši binarna serijalizacija i deserijalizacija liste objekata?
40. Za programsko rješenje urađeno za zadatak 35 u okviru poglavlja 11 uraditi sljedeće:
Omogućiti preko menija opciju Upis u Datoteku, da bi se osnovni podaci o svim predstavama upisali u datoteku predstave.dat. Implementirati i upravljanje karakterističnim izuzecima koji se mogu desiti prilikom rada sa datotekama.

Poglavlje 12: Rad sa XML dokumentima

U okviru ovog poglavlja dat je uvod u XML i rad sa XML dokumentima primjenom C# jezika kroz sljedeće tematske jedinice:

- 1.Uvod u XML
2. XML dokument
3. XML i .NET
4. XML serijalizacija i deserijalizacija

12.1.Uvod u XML

XML - Extensible Markup Language omogućava smještanje podataka u jednostavan, tekstualni format. XML nudi mogućnost razmjene podataka između više platformi. Struktura XML dokumenata je zasnovana na tagovima/oznakama (*markup*). Ne postoje unaprijed definirani tagovi i elementi. Tagove uvodi korisnik da bio opisao neki podatak ili objekat. Tagovi opisuju i semantiku dokumenta. Tag može indicirati da je element datum, osoba, predmet. U XML dokumentima, tagovi ne govore ništa o tome kako se dokument prikazuje. XML nema prezentacijsku namjenu. XML specifikacija definira gramatiku za XML dokumente koja propisuje pravila vezana za smještanje tagova, nazine tagova, pozicije tagova. Više o namjeni XML-a i njegovojoj specifikacije se može naći na sajtu www.w3.org.

12.2. XML dokument

XML dokument je set podataka sastavljen od više XML elemenata. XML dokument se na disk zapisuje kao tekstualna datoteka sa ekstenzijom .xml.

12.2.1.XML elementi

XML element sadrži otvarajući tag, podatak unutar elementa i zatvarajući tag. Otvarajući (početni) i zatvarajući (krajnji) tag elementa sadrži naziv elementa u okviru simbola <>, sa razlikom da zatvarajući tag počinje sa </>. Nazivi tagova generalno reflektiraju sadržaj unutar elementa. Ako je naziv elementa predmet tada <predmet> i </predmet> su otvarajući i zatvarajući tag tog elementa. Sve između početnog i krajnjeg taga se naziva sadržaj elementa. Ako je sadržaj elementa predmet „Razvoj programskih rješenja“ to se zapisuje XML notacijom:

```
<predmet> Razvoj programskih rješenja </predmet>
```

RAZVOJ PROGRAMSKIH RJEŠENJA

XML je osjetljiv na slova. <Predmet> nije isto što i <PREDMET> ili <predmet>. XML nazivi elemenata mogu početi sa slovima, idiogramima i donjom crticom. Ne mogu početi sa brojem ili tačkom.

Elementi mogu sadržavati druge elemente. Naprimjer, predmet se može detaljnije opisati da se naznači naziv predmeta, broj studenata i semestar. U tom slučaju XML element `predmet` sadrži 3 elementa. To se može prikazati sa XML dokumentom 12.1.

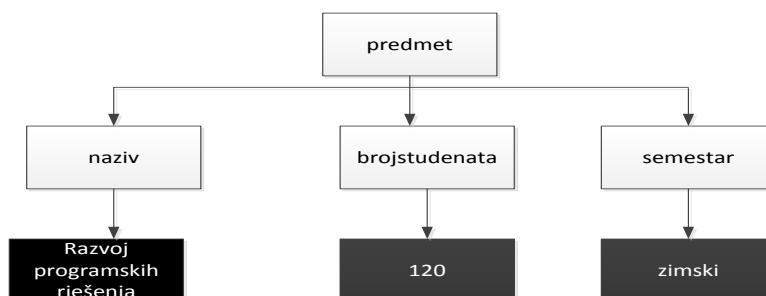
```
<predmet>
    <naziv> Razvoj programskih rješenja </naziv>
    <brojstudenata> 120 </brojstudenata>
    <semestar> zimski </semestar>
</predmet>
```

XML 12.1: XML dokument koji prikazuje jedan element (`predmet`) koji unutar sebe sadrži 3 elementa

`predmet` element se naziva roditelj (eng. parent) element za elemente `naziv`, `brojstudenata`, `semestar`, koji su njegova djeca. Djeca čvorovi u XML-u imaju tačno jednog roditelja.

Korijen (root) element

Svaki XML dokument ima jedan element koji nema roditelja. To je prvi element u dokumentu i element koji sadrži sve druge elemente. U iznad prikazanim primjerima element `predmet` ispunjava tu ulogu. Naziva se korijen (*root*) element dokumenta. Svaki dobro formiran XML element ima tačno jedan korijen element. XML dokument formira strukturu podataka koja se naziva drvo. Slika 12.1 prikazuje hijerarhijsku relaciju među elementima. U sivim pravougaoncima su elementi a u crnim sadržaj elemenata.



Slika 12.1: Hijerarhijska struktura XML dokumenta

RAZVOJ PROGRAMSKIH RJEŠENJA

Struktura XML dokumenta može biti mnogo složenija od prikazanog XML dokumenta na slici 12.1. Elementi i sami mogu biti roditelj elementi. Naprimjer, element naziv bi mogao biti element roditelj sa elementima koji bi opisali puni naziv predmeta, šifru predmeta, skraćeni naziv predmeta.

Da bi se naglasila činjenica da svaki XML dokument mora imati korijen slijedi prikaz ne dozvoljene XML strukture u kojoj postoje dva elementa koja su bez korijena.

```
<predmet> Razvoj programskih rjesenja </predmet>  
<predmet> Objektno orijentirana analiza </predmet>
```

Može se formirati XML dokument koji sadrži više elemenata. Ukoliko je potrebno formirati XML dokument koji sadrži podatke za više predmeta tada je najbolje korijen element nazvati predmeti (XML 12.2). Taj element bi bio roditelj za sve predmet elemente.

```
<predmeti>  
  <predmet>  
    <naziv> Razvoj programskih rješenja </naziv>  
    <brojstudenata> 120 </brojstudenata>  
    <semestar> zimski </semestar>  
  </predmet>  
  
  <predmet>  
    <naziv> Objektno orijentisana analiza </naziv>  
    <brojstudenata> 100 </brojstudenata>  
    <semestar> ljetni </semestar>  
  </predmet>  
  
</predmeti>
```

XML 12.2: XML dokument sa više predmet elemenata

XML elementi se ne mogu preklapati. To znači ako je neki element otvoren unutar nekog taga, unutar njega mora biti i zatvoren. XML dokument 12.3 prikazan ispod nije dobro formiran jer element naziv se mora zatvoriti u okviru elementa predmet.

```
<predmet>  
  <naziv>Razvoj programskih rješenja  
  </predmet>  
  </naziv>
```

XML 12.3: Preklapanje elemenata nije dozvoljeno

Moguće je da XML element sadrži element bez sadržaja, tzv. prazni element. Prazni element se označava sa `<predmet></predmet>` ili `<predmet />`.

Atributi

XML elementi mogu imati attribute. Atributi se navode u otvarajućem tagu u formi `naziv="vrijednost"`. Vrijednosti atributa se smještaju u okviru jednostrukih ili dvostrukih navodnika. Npr. element semestar može imati atribut broj. To se naznačava sa:

```
<semestar broj="III"> zimski </semestar>
```

Ukoliko element ima više atributa, atributi se odvajaju praznim prostorom.

```
<semestar broj="II" godina=="III" > zimski </semestar>
```

Atributi se pretežno koriste za označavanje informacija koje nisu esencijalne za sve korisnike dokumenta. Elementi, tekst odnosno sadržaj unutar elemenata, atributi se tretiraju kao posebni čvorovi (*nodes*) unutar XML dokumenta. Kao posebni čvorovi tretiraju se i komentari, XML deklaracija.

Komentari

Komentari imaju istu namjenu kao i komentari u programskim jezicima. Mogu se naći bilo gdje u okviru XML dokumenta osim u okviru tagova u formatu:

```
<!--XML dokument sadrži podatke za predmete -->
```

12.2.2.XML deklaracija

XML dokumenti mogu sadržavati XML deklaraciju. XML deklaracija počinje sa `xml` nakon čega slijedi `version`, `standalone` i `encoding` pseudo-atributi. Atribut `version` označava verziju XML-a, `encoding` se odnosi na set karaktera koji se koriste za prikazivanje sadržaja dokumenta, `standalone` označava da li postoji schema uz dokument. Ako XML dokumenti imaju XML deklaraciju, tada deklaracija mora biti prva linija u dokumentu. Primjer deklaracije sa naznačenom XML verzijom i kodnim tekstualnim standardom:

```
<?xml version="1.0" encoding="utf-8"?>
```

Dobro-formiran XML dokument

Svaki XML dokument mora biti dobro-formiran. Neka sintaksna pravila su već spomenuta. Sumarno, da bi XML dokument bio dobro formiran sljedeća pravila trebaju biti ispunjena:

- Svaki početni (otvarajući) tag mora imati odgovarajući krajnji (zatvarajući) tag.
- Elementi se mogu ugnježdavati ali ne i preklapati.
- Mora biti tačno jedan korijen element.

- Vrijednosti atributa moraju biti pod navodnicima.
- Element ne može imati dva atributa sa istim nazivom.
- Komentari se ne mogu naći unutar tagova.
- < , & , > ne mogu biti karakteri u okviru elementa ili atributa.

Za XML dokumente je važno i da se osiguraju pravila formiranja dokumenta koja se odnose na pojedinu XML primjenu. Da bi se provjerilo da li XML dokument ispunjava pravila propisana za specifičnu primjenu potrebno je specificirati ta pravila. To se postiže sa shemama i definicijama tipa XML dokumenta. Nakon toga je moguće provjeravati da li je dokument validan odnosno u skladu sa propisanim pravilima.

Programi koji čitaju XML dokumente i ispituju njihove pojedinačne elemente nazivaju se XML parseri. Neki parseri provjeravaju samo da li je dokument dobro formiran. Drugi imaju mogućnost provjere da li je dokument validan. Visual Studio ima ugrađen parser koji omogućava obadvije provjere.

12.2.3. Validacija XML dokumenata

XML podržava dva načina definiranja koji elementi i atributi se mogu smjestiti u dokument i u kojem redoslijedu. To su DTD- (Document Type Definition - definicija tipa dokumenta) i shema (*schemas*).

DTD dokumenti formalno specificiraju koji elementi se mogu pojaviti u dokumentu, gdje se mogu pojaviti u dokumentu, koji su mogući atributi, koji je dozvoljeni sadržaj elementa. DTD ne koristi XML baziranu sintaksu i danas se rjeđe koriste.

Sheme za opisivanja pravila koriste XML baziranu sintaksu. Pružaju poboljšani način specificiranja validnosti XML dokumenta. Sheme mogu biti veoma kompleksne. Za pisanje shema potrebno je dublje znanje o XML-u. U ovom materijalu se neće detaljnije opisivati sheme, slijedi samo prikaz jedne jednostavne sheme da bi se stekao utisak o osnovnoj strukturi i namjeni pojedinačnih shema. Mnogi alati imaju mogućnost automatskog formiranja XML sheme za XML dokument. Obično se formira jedan element u XML dokumentu, nakon toga se formira shema po kojoj se provjeravaju svi ostali elementi. XML 12.4 prikazuje XML dokument i shemu za taj dokument formiranu u okviru Visual Studio parsera.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
<predmet>          <?xml version="1.0" encoding="utf-8"?>

    <naziv>RPR</naziv>      <xsschema
    <brojstudenata>120</brojstudenata>  attributeFormDefault="unqualified"elementFormDefault="qualified"
                                            xmlns:xss="http://www.w3.org/2001/XMLSchema">

    <semestar>zimski</semestar>
    </predmet>
        <xss:complexType>

            <xss:sequence>

                <xss:element name="naziv" type="xss:string" />

                <xss:element name="brojstudenata" type="xss:unsignedByte" />

                <xss:element name="semestar" type="xss:string" />

            </xss:sequence>
        </xss:complexType>
    </xss:element>
</xss:schema>
```

XML12.4: XML dokument i validacijska shema

Bez objašnjavanja sintakse na kojoj je shema zasnovana vidljivo da je element predmet kompleksni tip sastavljen od sekvene tri elementa (naziv, brojstudenata, semestar). Ukoliko bi izbrisali naprimjer element naziv ili formirali dodatni element drugačije strukture parser bi prijavio grešku.

12.3.XML i .NET

Osnovni imenovani prostor koji omogućava veliki broj klasa za rad sa XML-om je System.Xml. Neke od tih klasa su:

- XmlTextReader klasa se koristi za brzo čitanje prema naprijed (*forward reading*) XML dokumenta bez validacije. XmlValidatingReader implementira isti princip čitanja kao XmlTextReader klasa ali pruža i mogućnost validiranja XML dokumenta. XmlTextReader i XmlValidatingReader su izvedene iz apstraktne klase XmlReader.
- XmlTextWriter se koristi za pisanje XML-a u datoteku.
- Xml DOM je skupina naprednijih klasa koje implementiraju W3C Document Object Model (DOM).

RAZVOJ PROGRAMSKIH RJEŠENJA

12.3.1. `XmlTextWriter` klasa

Za pisanje podataka samo prema naprijed u XML dokument koristi se `XmlTextWriter` klasa. Klasa nudi mnoge osobine koje se odnose na formatiranje XML elemenata (tabela 12.1), metode za kreiranje *streama*, metode za zatvaranje i oslobađanje resursa korištenih od objekata ove klase, metode za pisanje (tabela 12.1).

Osobina	Opis
Formatting	Određuje kako se izlaz formatira. Opcije su <code>None</code> (bez formatiranja) i <code>Indented</code> (sa uvlačenjem).
IndentChars	Koristi se za označavanje karaktera kojim se obilježava uvlačenje. Podrazumijevana oznaka je prazan karakter.
Indentation	Odnosi se broj <code>IndentChars</code> karaktera koji se pišu za svaki nivo u hijerarhiji XML dokumenta. Podrazumijevana vrijednost je 2.
QuoteChar	Predstavlja karakter koji se koristi za označavanje vrijednosti atributa. Podrazumijevana oznaka su navodnici.
Metoda	Opis
<code>WriteAttributeString</code>	Piše atribut sa specifičnom vrijednosti.
<code>WriteComment</code>	Piše tekst kao XML komentar.
<code>WriteName</code>	Piše naziv elementa.
<code>WriteStartDocument</code>	Piše XML deklaraciju.
<code>WriteStartElement</code>	Piše otvarajući tag elementa.
<code>WriteEndElement</code>	Piše zatvarajući tag elementa.
<code>WriteString</code>	Piše tekst.
<code>WriteWhitespace</code>	Piše string praznih karaktera.

Tabela 12.1: Osobine za formatiranje i metode klase `XmlTextWriter` za pisanje u XML dokumente

Kada se koristi `XmlTextWriter`, za upis u XML datoteku, prvo se kreira *stream* koji predstavlja datoteku u koju će se XML dokument zapisati. Nakon toga se kreira instanca klase `XmlTextWriter`. Sa `XmlTextWriter` klasom ne može se jednostavno upisati cijeli element u datoteku, prvo se upisuje početni tag, poslije sadržaj i na kraju zatvarajući tag.

Slijedi kod 12.1 koji kreira datoteku `c:\TestDir\predmet2.xml` i u nju upisuje jedan element koji opisuje predmet. Struktura elementa predmet je već pominjana. Komentarima u kodu su navedeni i objašnjeni svi bitni koraci.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
private void kreirajButton_Click(object sender, EventArgs e)
{
    //1. Kreiranje streamWritera za datoteku c:\TestDir\predmet2.xml
    StreamWriter streamWriter = new StreamWriter(@"c:\TestDir\predmet2.xml" );

    //2. Kreiranje XmlTextWriter-a sa nazivom pisac
    XmlTextWriter pisac = new XmlTextWriter(streamWriter);

    //3. Postavljanje formatiranja
    pisac.Formatting = Formatting.Indented;

    //4. Pisanje u dokument - element po element

    pisac.WriteStartDocument();           // Deklaracija dokumenta
    pisac.WriteStartElement("predmeti"); // Početni tag korijen elementa

    pisac.WriteStartElement("predmet"); // Početni tag elementa predmet

    pisac.WriteStartElement("naziv");      // Početni tag elementa naziv
    pisac.WriteString("Razvoj Programskih rjesenja"); // Tekst vezan za element naziv
    pisac.WriteEndElement();             // Krajnji tag elementa naziv
    pisac.WriteStartElement("brojstudenata"); // Početni tag elementa brojstudenata
    pisac.WriteString("120");           //Tekst vezan za element brojstudenata
    pisac.WriteEndElement();             //Krajnji tag elementa brojstudenata
    pisac.WriteStartElement("semestar"); //Početni tag elementa semestar
    pisac.WriteString("zimski");        //Tekst vezan za element semestar
    pisac.WriteEndElement();            //Krajnji tag elementa semestar

    pisac.WriteEndElement();           // Krajnji tag elementa predmet

    pisac.WriteEndElement();           // Krajnji tag korijen elementa

    // 5. Zatvaranje streama i datoteke
    pisac.Close();
}
```

Kod 12.1: Kreiranje i pisanje jednog elementa u XML dokument sa XmlTextWriter klasom

Nakon izvršavanja programa, na lokaciji c:\TestDir se nalazi datoteka predmet2.xml sadržaja:

```
<?xml version="1.0" encoding="utf-8"?>
<predmeti>
    <predmet>
        <naziv>Razvoj Programskih rjesenja</naziv>
        <brojstudenata>120</brojstudenata>
        <semestar>zimski</semestar>
    </predmet>
</predmeti>
```

XML 12.4: Datoteka Predmet2.xml formirana korištenjem XmlTextWriter klase

RAZVOJ PROGRAMSKIH RJEŠENJA

12.3.2. `XmlTextReader` klasa

`XmlTextReader` omogućava način čitanja i prepoznavanja XML podataka sa minimalnim korištenjem resursa. Prilikom čitanja XML dokumenta sa ovom klasom nije moguć povratak na neku od prethodnih pozicija bez ponovnog čitanja dokumenta od početka. Važna osobina `XmlTextReader` klase je osobina `NodeType` koja određuje tip čvora. Tip čvora je član `XmlNodeType` liste vrijednosti koja sadrži vrijednosti: `Attribute` (čvor je atribut), `Comment` (čvor je komentar), `Document` (čvor je korijen), `Element` (čvor je početni tag elementa), `Text` (čvor je tekst), `EndElement` (čvor je krajnji tag elementa) i dr..

Najvažnija metoda `XmlTextReader` klase je `Read`, koja dobavlja sljedeći čvor dokumenta. Kada se čvor pribavi može se koristiti `NodeType` osobina da bi se odredio tip čvora. Ako `XmlTextReader` otkrije da XML dokument nije dobro formiran, dešava se izuzetak tipa klase `XmlException`.

Slijedi primjer koji pokazuje kako da se čita XML dokument koristeći `XmlTextReader` klasu. Sadržaj dokumenta se piše u okviru multilinijskog teksta polja.

U okviru programa (kod 12.2) potrebno je:

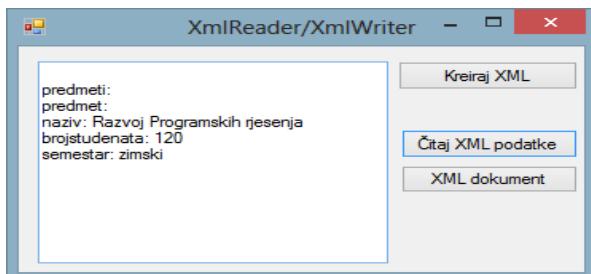
- Kreirati `XmlTextReader` objekat-čitač.
- Čitati čvorove XML dokumenta (metoda `Read`)
- Provjeravati tip čvora i na osnovu tipa čvora (`Element`, `Text`) pisati odgovarajući tekst u multilinijsko teksta polje.

```
private void citacButton_Click(object sender, EventArgs e)
{
    //1. Kreiranje citaca XmlTextReader klase i povezivanje sa datotekom c:\TestDir\predmet.xml
    XmlTextReader citac = new XmlTextReader(@"c:\TestDir\predmet2.xml");

    // 2. Pribavljanje-čitanje čvorova XML dokumenta sa Read metodom
    while (citac.Read())
    {
        //3. provjerava se tip čvora,
        // u ovom slučaju se provjerava da li je element (početni) ili je sadržaj (tekst) uz element
        switch (citac.NodeType)
        {
            case XmlNodeType.Element:
                textBox1.Text += "\r\n" + citac.Name + ": ";
                break;
            case XmlNodeType.Text:
                textBox1.Text += citac.Value;
                break;
        }
    }
    citac.Close(); //4.zatvaranje strelama
}
```

Kod 12.2: Čitanje XML dokumenta

Scenarij izvršavanja:



Nakon klika na **Čitaj XML podatke** nazivi elemenata i njihov sadržaj se prikazuju na formi u multilinijskom tekstualnom polju.

Upravljanje atributima

Ako XML elementi imaju atribute procesiranje takvog elementa može se uraditi sa linijama koda tipa:

```
case XmlNodeType.Element:  
    //...  
    if (citac.AttributeCount > 0)  
  
        while (citac.MoveNextAttribute())  
  
            textBox1.Text += citac.Name + citac.Value;
```

Dio koda za rad sa atributima unutar elemenata

AttributeCount osobina daje informaciju o broju atributa elementa, a metoda MoveToNextAttribute iterativno procesira kolekciju atributa.

12.3.3. Prikaz XML dokumenta u DataGridView kontroli

DataGridView kontrola omogućava tabelarni prikaz sadržaja. Najčešće se povezuje sa tabelom u bazi podataka, datotekom ili XML dokumentom. Implementira je istoimena klasa, koja nudi mnoge osobine, metode i događaje.

Slijedi primjer koda koji prikazuje sadržaj dokumenta c:\TestDir\Predmet2.xml tabelarno u DataGridView kontroli.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
private void gridXmlButton_Click_1(object sender, EventArgs e)
{
    try
    {
        //1. Kreiranje strem-a (citaca) tipa klase XmlTextReader i
        // povezivanje sa c:\TestDir\Predmet2.xml"

        XmlTextReader citac = new XmlTextReader(@"c:\TestDir\Predmet2.xml");

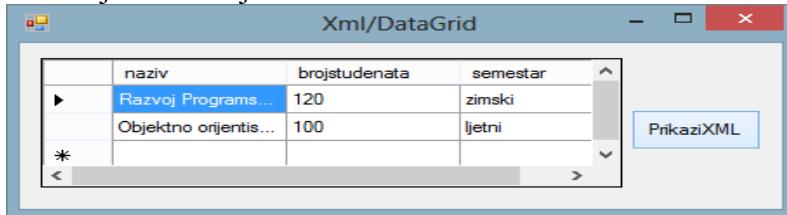
        // 2. Kreiranje seta podataka
        DataSet ds = new DataSet();

        //3. Čitanje sadržaja strem-a u set podataka ds
        ds.ReadXml(citac);

        // 4. Prikazivanje podataka u dataGridView kontroli
        dataGridView1.DataSource = ds.Tables[0].DefaultView;
        citac.Close();
    }
    catch (XmlException ex)
    {
        MessageBox.Show(ex.ToString());
    }
}
```

Kod 12.3: Prikaz dokumenata u DataGridView kontroli

Scenarij izvršavanja:



Nakon klika na PrikaziXML sadržaj dokumenta se prikazuje u DataGridView kontroli

RAZVOJ PROGRAMSKIH RJEŠENJA

12.3.4.XML DOM

XML Document Object Model (XML DOM) je skup klasa koje omogućavaju intuitivan pristup i rad sa XML dokumentima. DOM predstavlja i procesira XML dokument kao strukturu drveta.

XML DOM sadrži više klasa, smještenih u `System.XML`. U tabeli 12.2 je dat pregled dijela XML DOM klasa.

Klasa	Opis/Namjena
<code>XmlNode</code>	Klasa za hijerarhijski rad sa čvorovima u XML dokumentu. Sa jednog čvora moguća je navigacija na bilo koju poziciju u dokumentu. Sve ostale klase u XML DOM-u su naslijedene od ove klase. Nudi metode selekcije <code>SelectSingleNode</code> , <code>SelectNodes</code> koje koriste specijalni upitni <code>Xpath</code> jezik.
<code> XmlDocument</code>	Koristi se za spašavanje (<code>Save</code> metoda) na disk ili učitavanje (<code>Load</code>) sa diska. Sadrži i metode za kreiranje: <code>CreateNode</code> - kreira čvor. Tipovi čvorova su određeni sa <code>XmlNodeTypes</code> listom. Sadrži i specijalizirane metode za kreiranje elemenata: <code>CreateElement</code> , <code>CreateAttribute</code> , <code>CreateTextNode</code> , <code>CreateComment</code> . Sadrži i metode za ubacivanje (<code>insert</code>) čvorova: <code>AppendChild</code> -dodaje dijete čvor za naznačeni čvor, <code>InsertAfter</code> - dodaje čvor iza naznačenog čvora, <code>InsertBefore</code> - dodaje čvor ispred naznačenog čvora.
<code>XmlElement</code>	Klasa za rad sa elementima u XML dokumentu. Sadrži osobine i metode za obradu čvorova i atributa. Osobine: <code>InnerText</code> – vraća spojeni tekst (sadržaj) djece tekućeg čvora. <code>InnerText</code> – isto kao i <code>InnerText</code> ali u rezultat uključuje i tagove. <code>FirstChild</code> – vraća prvo dijete elementa. <code>LastChild</code> – vraća zadnje dijete elementa. <code>ParentNode</code> –vraća roditelj čvor tekućeg čvora. <code>NextSibling</code> - vraća sljedeći čvor koji ima istog roditelja kao i tekući čvor. <code>HasChildNodes</code> –provjerava da li tekući element ima djece.
<code>XmlAttribute</code>	Klasa za rad sa atributima.
<code>XmlText</code>	Klasa za rada sa tekstrom (sadržajem) između početnog i krajnjeg taga.
<code>XmlComment</code>	Klasa za rad sa komentarima.
<code>XmlNodeList</code>	Klasa za rad sa kolekcijom čvorova.

Tabela 12.2: XML DOM klase

RAZVOJ PROGRAMSKIH RJEŠENJA

Slijedi primjer (kod 12.4) koji koristi XML DOM klase za dodavanje elementa u dokument Predmet.xml, čija struktura je već poznata. Prvo se kreira objekat klase XmlDocument. Nakon toga se čita korijen elementa, kreiraju tagovi i sadržaj za novi element i njegovu djecu. Vrši se povezivanje kreiranih elemenata u hijerarhijsku cjelinu. Na kraju se novo kreirani element dodaje iza prvog djeteta korijen element.

```
private void dodajXmlElement_Click(object sender, EventArgs e)
{
    //1. Kreiranje objekta dokument tipa klase XmlDocument i punjenje XML dokumenta u memoriju
    XmlDocument dokument = new XmlDocument();
    dokument.Load(@"C:\TestDir\Predmet.xml");

    //2. Čitanje korijena dokumenta, u ovom slučaju to je tag - predmeti
    XmlElement korijen = dokument.DocumentElement;

    //3. Kreiranje tagova za novi element nazvan novipredmet i njegovu djecu
    XmlElement novipredmet = dokument.CreateElement("predmet");
    XmlElement novinaziv = dokument.CreateElement("naziv");
    XmlElement novibrojstudenata = dokument.CreateElement("brojstudenata");
    XmlElement novisemestar = dokument.CreateElement("semestar");

    //4. Kreiranje sadržaja za djecu elementa novipredmet
    XmlText nazivpredmeta = dokument.CreateTextNode("OOAD");
    XmlText brojstudenata = dokument.CreateTextNode("122");
    XmlText semestar = dokument.CreateTextNode("ljetni");

    //5. Povezivanje elementa novipredmet sa djecom
    novipredmet.AppendChild(novinaziv);
    novipredmet.AppendChild(novibrojstudenata);
    novipredmet.AppendChild(novisemestar);

    //6. Povezivanje elemenata (djece) sa kreiranim sadržajem za te elemente
    novinaziv.AppendChild(nazivpredmeta);
    novibrojstudenata.AppendChild(brojstudenata);
    novisemestar.AppendChild(semestar);

    //7. Ubacivanje novog kreiranog elementa iza prvog djeteta korijena
    korijen.InsertAfter(novipredmet, korijen.FirstChild);

    //8. Spašavanje XML dokumenta sa promjenama (dodan novi element) na disk
    dokument.Save(@"C:\TestDir\Predmet.xml");
}
```

Kod 12.4: Dodavanje elementa u Predmet.xml dokument pomoću kolekcije DOM klasa

Nakon izvršavanja programa sadržaj dokumenta Predmet.xml je:

```
<?xml version="1.0" encoding="utf-8"?>
<predmeti>
    <predmet>
        <naziv>Razvoj Programskih rjesenja</naziv>
        <brojstudenata>120</brojstudenata>
        <semestar>zimski</semestar>
    </predmet>
    <predmet>
        <naziv>OOAD</naziv>
        <brojstudenata>122</brojstudenata>
        <semestar>ljetni</semestar>
    </predmet>
</predmeti>
```

XML 12.5: Novi element dodan iza prvog djeteta čvora predmeti

Prikaz XML dokumenta u `treeView` kontroli

GUI kontrola `treeView` prikazuje formu drveta sastavljenu od hijerarhijski povezanih čvorova. Čvorovi su objekti koji sadrže vrijednosti i koji mogu pokazivati na druge objekte. Roditelj (*parent*) čvor sadrži djecu (*child*) čvorove. Djeca čvorovi mogu biti roditelji drugim čvorovima. U okviru ovog materijala se neće detaljnije objašnjavati osobine i metode klase koja implementira `treeView` kontrolu, već će se ilustrirati primjena ove kontrole za hijerarhijski prikaz XML dokumenta.

Primjer: Potrebno je kreirati Windows Forms aplikaciju sa `treeView` kontrolom i jednim dugmetom (`prikaziXml`). Događaj `Click` za to dugme treba da odgovori sa hijerarhijskim prikazom XML dokumenta u `treeView` kontroli.

Postupak povezivanja XML dokumenta i `treeView` kontrole objašnjen je u ispod prikazanom kodu 12.5.

```
private void prikaziXml_Click(object sender, EventArgs e)
{
    //1. Kreiranje dokument objekta tipa klase XmlDocument i punjenje XML dokumenta u memoriju
    XmlDocument dokument = new XmlDocument();
    dokument.Load(@"c:\testDir\Predmet.xml");

    //2. Čitanje korijena dokumenta, u ovom slučaju to je tag - predmeti
    XmlElement korijen = dokument.DocumentElement;

    treeView1.Nodes.Clear(); // brisanje čvorova, ako postoje, u treeView kontroli

    //3. Dodavanje korijena XML dokumenta u treeView kontrolu
    treeView1.Nodes.Add(new TreeNode(dokument.DocumentElement.Name));

    //4. Kreiranje čvora u treeView kontroli
    TreeNode drvoCvor = treeView1.Nodes[0];

    //5. Dodavanje XML elemenata u hijerarhijsku strukturu treeView kontrole
    // Parametri metode dodajCvor - cvor XMLdokumenta i cvor treeView kontrole
    dodajCvor(korijen,drvoCvor);
}
```

Kod 12.5: Dio koda za prikaz XML dokumenta u hijerarhijskoj strukturi

Unutar koda 12.5 poziva se metoda `dodajCvor` (kod 12.6).

RAZVOJ PROGRAMSKIH RJEŠENJA

```
private void dodajCvor(XmlNode XmlCvor, TreeNode drvoCvor)
{
    XmlNode XmlTekuciCvor ; // trenutni/tekući čvor u XML dokumentu
    TreeNode drvoTekuciCvor ; // trenutni/tekući čvor u treeView kontroli
    XmlNodeList XmlListaCvorova ; //koristi se za formiranje liste čvorova djece za tekući čvor

    if (XmlCvor.HasChildNodes) // provjerava da li XML čvor ima djece
    {

        XmlListaCvorova = XmlCvor.ChildNodes; // formira se lista XML djece za trenutni čvor

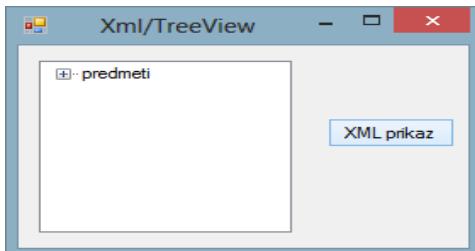
        // dok god čvor ima djece
        for (int i = 0; i <= XmlListaCvorova.Count - 1; i++)
        {
            // tekući čvor je i-djete
            XmlTekuciCvor = XmlCvor.ChildNodes[i];

            drvoCvor.Nodes.Add(new TreeNode(XmlTekuciCvor.Name)); // dodaj XML čvor u drvo
            drvoTekuciCvor = drvoCvor.Nodes[i];

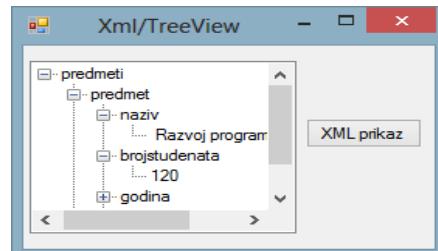
            dodajCvor(XmlTekuciCvor,drvoTekuciCvor); // rekurzivni poziv metode dodajCvor
        }
    }
    else
    {
        drvoCvor.Text = XmlCvor.InnerText.ToString(); // tekst (sadržaj) elementa
    }
}
```

Kod 12.6: Metoda za dodavanje XML čvorova u hijerarhijsku strukturu drveta

Scenarij izvršavanja:



Nakon klika na XML prikaz



Moguće je kretanje po čvorovima

12.4.XML Serijalizacija i deserijalizacija

XML serijalizacija je proces konverzije javnih (`public`) osobina objekta u XML format. Serijalizirane objekte je jednostavnije prenosići i razmjenjivati između različitih platformi. XML deserijalizacija vraća objekat u originalno stanje na osnovu podataka iz XML datoteke.

12.4.1. XML serijalizacija

Prilikom serijalizacije klase u XML format, serijaliziraju se samo javna polja i osobine. Klasa mora imati ne parametarizirani konstruktor. `ReadOnly` članovi se ne serijaliziraju.

`XmlSerializer` klasa locirana u `System.Xml.Serialization` se koristi za serijalizaciju i deserijalizaciju. Za serijalizaciju se koristi `Serialize` metoda `XmlSerializer` klase. Metoda ima dva parametra. Prvi je referenca na XML dokument u koji se zapisuju rezultati serijalizacije, a drugi je objekat koji se serijalizira.

Slijedi primjer koji serijalizira objekat klase `Proizvod` u dokument `Proizvod.xml`. U metodi koja obrađuje serijalizaciju (kod 12.7) prvo se kreira objekat tipa klase `XmlSerializer`. Pri tome se konstruktoru `XmlSerializer` klase predaje tip objekta (`Proizvod` klasa). Nakon toga se kreira `stream` (`Proizvod.xml`) u koji će se upisati stanje objekta zapisano u XML formatu. Serijalizacija se obavlja metodom `Serialize` kojoj se predaje referenca na dokument `c:\TestDir\Proizvod.xml` i objekat `proizvod` tipa klase `Proizvod`.

```
using System;
using System.Xml;
using System.Xml.Serialization;
using System.IO;
namespace SerijalizacijaObjekta
{
    public class Program
    {
        // Tip objekata serijalizacije je klasa Proizvod
        public class Proizvod
        {
            public int sifra;           // public članovi klase
            public string naziv;
            public double cijena;
            public string napomena;

            public Proizvod() {} // zahtjevani konstruktor bez parametara
            public Proizvod(int psifra, string pnaziv, double pcijena, string pnapomena)
            {
                sifra = psifra;
                naziv = pnaziv;
                cijena = pcijena;
                napomena = pnapomena;
            }
            public override string ToString()
            {
                return string.Format("{0}, {1}, {2:F2}KM, {3}", sifra, naziv, cijena, napomena);
            }
        }
    }
}
// nastavak na sljedećoj stranici
```

RAZVOJ PROGRAMSKIH RJEŠENJA

```
static void Main(string[] args)
{
    string XmlDok = @"c:\TestDir\Proizvod.xml";

    Proizvod proizvod1 = new Proizvod(1, "Coca Cola", 1.20, "Akcija");

    /* Poziv metode koja implementira serijalizaciju
       Objekta proizvod1 u c:\TestDir\Proizvod10.xml dokument */
    XmlSerijalizacija(proizvod1, XmlDok);

}

/* Metoda u kojoj se vrši serijalizacija
static public void XmlSerijalizacija(Proizvod proizvod, string XmlDok)
{
    //1. Kreiranje objekta serializer tipa klase XmlSerializer
    // navodi se tip objekta serijalizacije (Proizvod)
    XmlSerializer serializer = new XmlSerializer(typeof(Proizvod));

    //2. Kreiranje streama za pisanja
    StreamWriter pisac = new StreamWriter(XmlDok);

    //3. Poziv metode serializer objekta koja serijalizira objekat u stream
    serializer.Serialize(pisac, proizvod);

    pisac.Close(); //4. Zatvaranje streama/datoteke
}
}
```

Kod 12.7: Serijalizacija objekta u XML dokument

Objekat je serijaliziran u dokument Proizvod.xml:

```
<Proizvod>
<sifra>1</sifra>
<naziv>Coca Cola</naziv>
<cijena>1.2</cijena>
<napomena>Akcija</napomena>
</Proizvod>
```

XML 12.6: XML dokument formiran serijalizacijom proizvoda (objekta) u XML dokument

RAZVOJ PROGRAMSKIH RJEŠENJA

Specijalni atributi za XML serijalizaciju

Postoji više pseudo-atributa koji se koriste prilikom serijalizacije. Oni daju posebne instrukcije pri kreiranju XML dokumenta za članove (polja i attribute) klase. Neki od tih atributa su:

-`XmlAttribute`: Član klase označen sa ovim atributom se neće serijalizirati.

-`XmlElement`: Uz ovaj atribut se specificira naziv elementa u koji se serijalizira član klase. Prilikom serijalizacije XML tagovi dobijaju imena članova klase. Ovaj tag se koristi u slučaju kada se želi drugačiji naziv za XML tag klase.

-`XmlAttribute`: Član klase će se serijalizirati kao XML atribut roditelj čvora. I uz ovaj atribut se može navesti novi naziv za tag.

-`XmlRoot`: Korijen element u XML dokumentu dobija ime klase ako se sa ovim atributom ne navede drugo ime za korijen XML dokumenta. Ovaj atribut se postavlja iznad klase koja se serijalizira.

Ukoliko se klasa iz primjera iznad označi sa specijalnim pseudo-atributima:

```
[XmlAttribute ("DomaciProizvod")]
public class Proizvod
{
    [XmlElement ("Barcode")]
    public int sifra;
    [XmlAttribute ("brand")]
    public string naziv;
    public double cijena;
    [XmlAttribute]
    public string napomena;

    ....
}
```

Klasa označena sa pseudo-atributima za XML serijalizaciju

nakon serijalizacije iznad prikazane klase dođen je XML dokument 12.7.

```
<DomaciProizvod brand="Coca Cola">

<Barcode>1</Barcode>

<cijena>1.2</cijena>

</DomaciProizvod>
```

XML 12.7: XML dokument nakon serijalizacije klase označene pseudoatributima

RAZVOJ PROGRAMSKIH RJEŠENJA

Serijalizacija kolekcije objekata

Moguće je izvršiti i XML serijalizaciju kolekcije objekata. Ispod je metoda kojom se vrši serijalizacija liste objekata. Postupak je isti kao kod serijalizacije objekta, samo što se kao parametar konstruktoru `XmlSerializer` klase predaje lista objekata.

```
static public void XmlSerijalizacija(List<Proizvod> listaProizvoda)
{
    XmlSerializer x = new XmlSerializer(typeof(List<Proizvod>));
    StreamWriter writer = new StreamWriter(@"c:\TestDir\Proizvod.xml");

    x.Serialize(writer, listaProizvoda);
    writer.Close();
}
```

Kod 12.8: XML serijalizacija liste objekata

12.4.2. XML deserijalizacija

XML deserijalizacija se koristi za prenos XML podataka u neki objekt. Proces deserijalizacije počinje kreiranjem objekta tipa `XmlSerializer` klase. Konstruktoru se kao i kod serijalizacije predaje tip objekta koji se serijalizira. Nakon toga se kreira *stream* za čitanje XML dokumenta i poziva se `Deserialize` metoda da vratи stanje objekta. `Deserialize` metoda kao parametar prima referencu na XML objekat a rezultat vraća u objekat koji je predmet serijalizacije.

```
static public void XmlDeserijalizacija(Proizvod proizvodD, string XmlDok)
{
    //1. Kreiranje objekta desererializer tipa klase XmlSerializer
    // navodi se tip objekta deserijalizacije (Proizvod)
    XmlSerializer serializer = new XmlSerializer(typeof(Proizvod));

    //2. Kreiranje streama za pisanja
    XmlTextReader citac = new XmlTextReader(XmlDok);

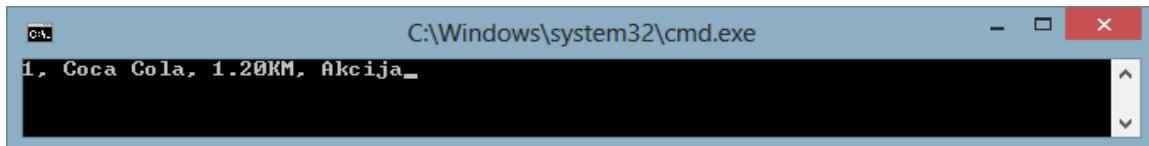
    //3. Poziva se Deserialize metoda da vrati stanje objekta
    proizvodD = (Proizvod)serializer.Deserialize(citac);

    citac.Close(); //4. Zatvaranje streama/datoteke
    Console.WriteLine(proizvodD); // Prikaz stanja objekta
}
```

Kod 12.9: XML deserijalizacija

RAZVOJ PROGRAMSKIH RJEŠENJA

Ukoliko se ova metoda pozvala iz programa prikazanog za XML serijalizaciju nakon metode serijalizacije dobijaju se sljedeće informacije:



```
C:\Windows\system32\cmd.exe
1. Coca Cola, 1.20KM, Akcija
```

Prilikom deserijalizacije se mogu koristiti pseudo-atributi `XmlElement`, `XmlAttribute`, `XmlText`. Ovi atributi pružaju informacije mapiranja XML tagova i članova klase. Njihovo značenje je objašnjeno kod serijalizacije.

Rezime:

U okviru ovog poglavlja opisana je osnovna struktura XML dokumenta. Prikazana su osnovna sintaksna pravila za kreiranje XML dokumenta. Objasnjenje su osnovne klase za rad sa XML-om iz .NET okruženja, i ilustrirani principi rada sa XML dokumentom iz C# koda. Uvedene su i dvije nove GUI kontrole za prikaz podataka u tabelarnom formatu i hijerarhijskom drvetu. Objasnjena je serijalizacija objekta i kolekcije objekata u XML dokument i deserijalizacija XML dokumenta u objekat ili kolekciju objekata.

Pitanja i zadaci za samostalni rad

1. Navedite puni naziv skraćenice XML.
2. Koja je osnovna namjena XML-a?
3. Objasnite šta XML pruža zato što je *markup* jezik?
4. Šta je tag (*markup*)?
5. Da li XML ima unaprijed definiran skup tagova?
6. Šta je XML dokument?
7. Od čega je sastavljen XML dokument?
8. Navedite strukturu osnovnog XML elementa.
9. Koja pravila se moraju poštovati kod imenovanja tagova?
10. Kako se navodi otvarajući a kako zatvarajući tag.
11. Da li XML elementi mogu sadržavati druge elemente?
12. Objasnite poziciju korijen elementa u XML dokumentu i njegove odnose sa drugim elementima.
13. Da li XML dokument može imati više elemenata?
14. Da li je dozvoljeno preklapanje XML elemenata?
15. Kako se navode atributi uz XML dokument?
16. Šta sve predstavlja čvor XML dokumenta?

RAZVOJ PROGRAMSKIH RJEŠENJA

17. Kako se pišu komentari u XML dokumentu?
18. Objasnite atribute deklaracijske linije XML dokumenta?
19. Navedite pravila koja se moraju ispoštovati da bi XML dokument bio dobro formiran.
20. Kako se naziva program koji provjerava da li je XML dokument dobro formiran?
21. Da li je svaki dobro formiran XML dokument i validan XML dokument?
22. Kako se može provjeravati validnost XML dokumenta?
23. Koji je osnovni .NET *namespace* koji omogućava rad sa XML-om?
24. Navedite osnovne klase za rad sa XML-om.
25. Koja je namjena `XmlTextWriter` klase?
26. Navedite neke osobine i metode `XmlTextWriter` klase.
27. Koja je namjena `XmlTextReader` klase?
28. Navedite neke osobine i metode `XmlTextReader` klase.
29. Navedite osobinu `XmlTextReader` klase kojom se određuje tip čvora.
30. Napišite dio koda kojim se procesiraju atributi nekog elementa.
31. Šta predstavlja i omogućava XML Document Object Model (XML DOM)?
32. Koju strukturu XML dokumenta podržava DOM?
33. Navedite osnovne DOM klase.
34. Šta se postiže XML serijalizacijom, a šta XML deserijalizacijom?
35. U kojem .NET *namespace* je podrška za rad sa XML-om?
36. Navedite koji članovi klase se prilikom serijalizacije serijaliziraju u XML format.
37. Navedite koji članovi klase se ne serijaliziraju u XML format.
38. Kojom klasom i metodom se vrši XML serijalizacija?
39. Navedite i objasnite značenje pseudo-atributa koji se koriste prilikom serijalizacije klase u XML format.
40. Kojom klasom i metodom se vrši deserijalizacija?
41. Kako se vrši XML serijalizacija i deserijalizacija liste objekata?
42. Za programsko rješenje urađeno za zadatak 35 u okviru poglavlja 11 uraditi sljedeće:

Podatke o prodanim kartama upisati u XML dokument `prodaja.xml`. Omogućiti i prikaz podataka upisanih u XML na formu (izaberite kontrolu).

POGLAVLJE 13: Višenitnost i asinhrono programiranje

U okviru ovog poglavlja objašnjeni su osnovni koncepti višenitnog programiranja, asinhrono programiranje, podrška C# i .NET okruženja za rad sa višenitnošću kroz sljedeće tematske jedinice:

- 1.Uvod
- 2(Programsko upravljanje nitima
3. Asinhroni mehanizmi
4. GUI i asinhronne operacije

13.1.Uvod

Mnoge današnje aplikacije zahtijevaju da se istovremeno, paralelno obavlja više zadataka. Mehanizmima izloženim u prethodnim poglavljima mogu se programirati samo zadaci koji se izvršavaju jedan za drugim. Kada metoda pozove neku drugu metodu tada metoda koja je izvršila poziv čeka da se izvrši metoda koju je pozvala i tek nakon toga se nastavljaju izvršavati ostali iskazi u toj metodi. U tom slučaju radi se o sinhronom izvršavanju. Postoje razni mehanizmi koji omogućavaju izvršavanje više zadataka istovremeno. Neki od njih su niti, taskovi, asinhrono programiranje. Niti (*threads*) predstavljaju bazične cjeline za izvršavanje koda pod savremenim operativnim sistemima. Nit je programska cjelina koja obavlja neki zadatak. Niti (jedna ili više) pripadaju jednom složenijem procesu. Višenitno programiranje predstavlja pisanje takvih programa koji se sastoje od više kooperativnih procesa i niti koje se izvršavaju simultano ili paralelno, pri tom koristeći zajedničke resurse kompjuterskog sistema.

Mnogi softverski paketi za prikazivanje Web stranica, programi za obradu teksta su višenitni. Primjer takve aplikacije je MS Word. Istovremeno je omogućen i unos teksta, brojanje riječi u dokumentu, gramatičke provjere, automatsko spašavanje dokumenta u zadanim vremenskim intervalima. Implementacija automatske dealokacije memorije u okviru .NET-a je također primjer niti koja radi istovremeno sa glavnim procesom.

Korištenje višenitnog koncepta omogućava interaktivnim aplikacijama da nastave rad, i kada se izvršava neka dugotrajna specifična operacija. U slučaju višeprocesorske arhitekture, niti se mogu istovremeno izvršavati, svaka na različitom procesoru, tako da se više taskova može izvršavati konkurentno, omogućavajući aplikacijama da operiraju efikasnije. Višenitost može također povećati performanse na jedno procesorskim sistemima simulirajući konkurentnost rada niti.

RAZVOJ PROGRAMSKIH RJEŠENJA

Sa asinhronim programiranjem metoda se može izvršavati u pozadini, a da pri tome metoda koja je izvršila poziv nastavi sa izvršavanjem. To se postiže u C#5.0 sa `async/await` ključnim rječima. Sama implementacija je tipično zasnovana na nitima ili taskovima.

U okviru .NET-a, podrška za rad sa nitima je u okviru `System.Threading` imenovanog prostora. Osnovna klasa za rad sa nitima je `Thread`. Metode ove klase se uvode u sljedećoj sekciiji. Nakon toga se uvode i klase iz `System.Threading.Tasks` imenovanog prostora.

13.2. Programsko upravljanje nitima

U okviru ovog dijela se objašnjava kreiranje niti, pauziranje niti, prekidanje izvršavanja niti, pozadinske niti, postavljanje prioriteta niti, predaja parametara nitima i sinhronizacija niti.

13.2.1. Kreiranje niti

Proces kreiranja niti sastoji se od kreiranja objekta `Thread` klase. Objekat `Thread` klase prima kao parametar objekat koji je tipa `ThreadStart` klase. Parametar `ThreadStart` klase je delegat koji pokazuje na metodu koja se treba izvršiti kada se nit startuje.

Sljedećim iskazima je moguće kreirati nit i delegirati joj zadatak koji treba izvršiti:

```
Thread nit1 = new Thread( new ThreadStart ( BrojacNit1));
```

ili:

```
ThreadStart startDelegate = new ThreadStart(BrojacNit1);  
  
Thread nit1 = new Thread(startDelegate);
```

Nit se počinje izvršavati sljedećim iskazom:

```
nit1.Start( );
```

`BrojacNit1` u ovim iskazima predstavlja metodu (zadatak) koju nit treba da izvrši. Isti zadatak se može dodijeliti za više niti.

Slijedi primjer koji ilustrira kreiranje i izvršavanje niti. U okviru glavne (`main`) metode u programu implementirane su dvije metode `BrojacNit1` i `BrojacNit2` koje broje i pišu do šest na standardni izlaz. Metodu `BrojacNit1` će izvršavati `nit1`, a metodu `BrojacNit2` `nit2`. U metodama se piše i tekst da bi se pojasnio proces izvršavanja niti. U kodu 13.1 komentarima su naglašeni bitni koraci vezani za kreiranje i startanje niti.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
using System.Threading;      // podrška za rad sa nitima

namespace KreiranjeStartanjeNiti
{
    class Program
    {
        // Metoda koju će izvršavati nit1
        public static void BrojacNit1 ( )
        {
            for( int i = 0 ; i <= 6 ; i ++ )
            {
                Console.WriteLine( "Prva pozvana nit iz glavne (main) niti " + i.ToString ( ) );
            }
        }

        // Metoda koju će izvršavati nit2
        public static void BrojacNit2 ( )
        {
            for( int i = 0 ; i <= 6 ; i ++ )
            {
                Console.WriteLine( "Druga pozvana niti iz glavne (main) niti " + i.ToString ( ) );
            }
        }

        public static void Main ( string [ ] args )
        {

            //1. Kreiranje niti nit1 i delegiranje zadatka (metoda BrojacNit1) za izvršavanje
            Thread nit1 = new Thread( new ThreadStart ( BrojacNit1));

            //2. Startovanje niti nit1 - Nit se počinje izvršavati
            nit1.Start( );

            //3. Kreiranje niti nit2 i delegiranje zadatka (metoda BrojacNit2) za izvršavanje
            Thread nit2 = new Thread( new ThreadStart ( BrojacNit2));

            // 4. Startovanje niti nit2 - Nit se počinje izvršavati
            nit2.Start( );

            //5. Glavna nit broji i piše do 6
            for( int i = 0 ; i <= 6 ; i++ )
            {
                Console.WriteLine( "Glavna nit " + i.ToString ( ) );
            }
            Console.WriteLine("Zavrsena glavna nit -metoda main");
        }
    }
}
```

Kod 13.1: Kreiranje, dodjeljivanje zadatka za izvršavanje i pokretanje dvije neovisne niti

RAZVOJ PROGRAMSKIH RJEŠENJA

Scenarij izvršavanja:

```
C:\Windows\system32\cmd.exe
Glavna nit 0
Glavna nit 1
Glavna nit 2
Glavna nit 3
Glavna nit 4
Glavna nit 5
Glavna nit 6
Završena glavna nit -metoda main
Prva pozvana nit iz glavne <main> niti 0
Prva pozvana nit iz glavne <main> niti 1
Prva pozvana nit iz glavne <main> niti 2
Prva pozvana nit iz glavne <main> niti 3
Druga pozvana niti iz glavne <main> niti 0
Druga pozvana niti iz glavne <main> niti 1
Druga pozvana niti iz glavne <main> niti 2
Druga pozvana niti iz glavne <main> niti 3
Druga pozvana niti iz glavne <main> niti 4
Druga pozvana niti iz glavne <main> niti 5
Druga pozvana niti iz glavne <main> niti 6
Prva pozvana nit iz glavne <main> niti 2
Prva pozvana nit iz glavne <main> niti 3
Prva pozvana nit iz glavne <main> niti 4
Prva pozvana nit iz glavne <main> niti 5
Prva pozvana nit iz glavne <main> niti 6
Press any key to continue . . .

C:\Windows\system32\cmd.exe
Prva pozvana nit iz glavne <main> niti 0
Prva pozvana nit iz glavne <main> niti 1
Prva pozvana nit iz glavne <main> niti 2
Prva pozvana nit iz glavne <main> niti 3
Druga pozvana niti iz glavne <main> niti 0
Druga pozvana niti iz glavne <main> niti 1
Druga pozvana niti iz glavne <main> niti 2
Druga pozvana niti iz glavne <main> niti 3
Druga pozvana niti iz glavne <main> niti 4
Druga pozvana niti iz glavne <main> niti 5
Druga pozvana niti iz glavne <main> niti 6
Prva pozvana nit iz glavne <main> niti 1
Prva pozvana nit iz glavne <main> niti 2
Druga pozvana niti iz glavne <main> niti 3
Druga pozvana niti iz glavne <main> niti 4
Druga pozvana niti iz glavne <main> niti 5
Druga pozvana niti iz glavne <main> niti 6
Press any key to continue . . .
```

Rezultati nakon prvog izvršavanja programa

Rezultati nakon drugog izvršavanja programa

Analizom dobijenih rezultata za dva izvršavanja programa vidljivo je da su različiti koraci izvršavanja. Iz programa se nije moglo uticati na tok izvršavanja pojedinih niti. Zanimljivo je uočiti da je metoda `main` terminirala prije ostalih niti. I metoda `main` je programska nit, koja se naziva glavna nit. Međutim, cijelokupni proces (program) ne terminira, dok se sve programske niti ne izvrše.

13.2.2. Pauziranje niti

Izvršavanje niti se može zaustaviti, odnosno nit se može pauzirati neki određeni period. Pauziranje niti se može postići sa `Sleep` i `TimeSpan` metodama. `Sleep` metoda kao parametar prima broj milisekundi za koje se pauzira izvršavanje niti. `TimeSpan` struktura ima više preklopnih konstruktora, koji dozvoljavaju postavljanje dana, sati, minuta, sekundi, milisekundi za koje treba zaustaviti proces izvršavanja niti.

Sljedeći kod 13.2 je modifikacija koda 13.1. U okviru koda 13.2 je kreirana klasa `BrojacNit` čija metoda `brojac` se dodjeljuje kao delegat metoda i za `nit1` i `nit2`. U metodi se sa `Thread.CurrentThread` osobinom dobija informacija o tome koja nit izvršava for petlju. Da bi omogućilo da se što ravnopravnije izvršavaju obadvije niti u okviru petlje je urađeno pauziranje tekuće niti za 1 milisekundu. Tretirani su izuzeci koji se mogu javiti prilikom poziva `Thread.Sleep` metode.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
using System.Threading;

namespace PauziranjeNiti
{
    // Klasa koja sadrži metodu koju će niti izvršavati

    public class BrojacNit
    {
        public void brojac()
        {
            // Metoda koju će niti izvršavati

            for (int i = 0; i < 10; i++)
            {
                Thread tekunaNit = Thread.CurrentThread;

                Console.WriteLine(tekunaNit.Name + " Brojim i pišem " + i);

                try
                {
                    Thread.Sleep(1);
                }
                catch (ArgumentException ae)
                {
                    Console.WriteLine(ae.ToString());
                }
                catch (ThreadInterruptedException tie)
                {
                    Console.WriteLine(tie.ToString());
                }
            }
        }
    }

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

            //1. Kreiranje dvije instance klase BrojacNit
            BrojacNit nzad1 = new BrojacNit();
            BrojacNit nzad2 = new BrojacNit();

            //2. Kreiranje niti nit1 i nit2 i delegiranje zadatka (metoda brojac klase BrojacNit) za izvršavanje

            Thread nit1 = new Thread(new ThreadStart(nzad1.brojac));
            Thread nit2 = new Thread(new ThreadStart(nzad2.brojac));

            //3. Postavljanje osobine Name (naziv) za niti
            nit1.Name = "Nit 1";
            nit2.Name = "Nit 2";

            // 4. Startovanje niti - Niti se počinju izvršavati
            nit1.Start();
            nit2.Start();

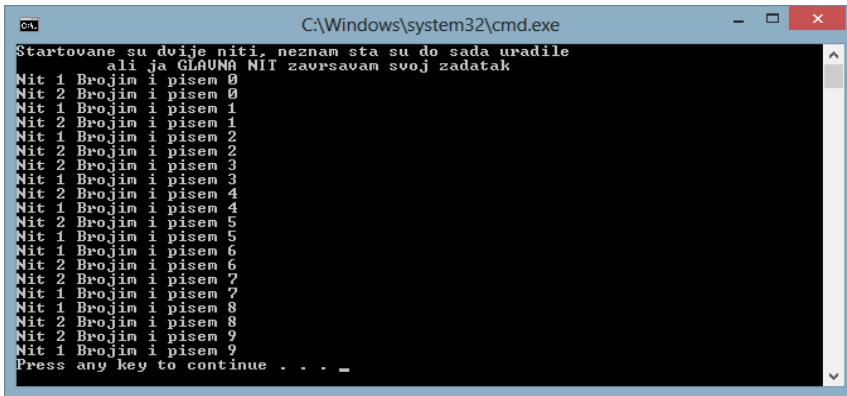
            // Poruka kojom se potvrđuje neovisan rad glavne niti i kreiranih niti
            Console.WriteLine("Startovane su dvije niti, neznam šta su do sada uradile");
            Console.WriteLine("           ali ja GLAVNA NIT završavam svoj zadatak");

        }
    }
}
```

Kod 13.2: Pauziranje niti sa Sleep metodom

RAZVOJ PROGRAMSKIH RJEŠENJA

Scenarij izvršavanja koda 13.2:



```
C:\Windows\system32\cmd.exe
Startovane su dvije niti, neznam sta su do sada uvadile
ali ja GLAVNA NIT zavrsavam svoj zadatak
Nit 1 Brojim i pisen 0
Nit 2 Brojim i pisen 0
Nit 1 Brojim i pisen 1
Nit 2 Brojim i pisen 1
Nit 1 Brojim i pisen 2
Nit 2 Brojim i pisen 2
Nit 1 Brojim i pisen 3
Nit 2 Brojim i pisen 3
Nit 1 Brojim i pisen 4
Nit 2 Brojim i pisen 4
Nit 1 Brojim i pisen 5
Nit 2 Brojim i pisen 5
Nit 1 Brojim i pisen 6
Nit 2 Brojim i pisen 6
Nit 1 Brojim i pisen 7
Nit 2 Brojim i pisen 7
Nit 1 Brojim i pisen 8
Nit 2 Brojim i pisen 8
Nit 1 Brojim i pisen 9
Nit 2 Brojim i pisen 9
Nit 1 Brojim i pisen 9
Press any key to continue . . .
```

Izvršavanje programa se odvijalo tako što je glavna nit (`main`) pokrenula `nit1` i `nit2` da se izvršavaju i odmah nakon toga izvršeno je pisanje poruke iz glavne niti. Zatim su `nit1` i `nit2` odradile svoj zadatak brojanja, i to dosta ravnopravno, što se postiglo pauziranjem jedne niti milisekundu. Pauziranje jedne niti je omogućilo drugoj niti da radi. Mogao se desiti i drugačiji scenarij izvršavanja, naprimjer mogla je `nit1` uraditi iteraciju `for` petlje prije nego što je glavna nit napisala poruku.

13.2.3. Prekidanje izvršavanja niti

Potpuno prekidanje izvršavanja niti se postiže sa `Abort` metodom `Thread` klase. Slijedi prikaz izmjenjene `main` metode iznad prikazanog programa. Izvršeno je pokretanje niti `nit1`, zatim je zaustavljeni izvršavanje glavne niti (`main` metoda) za 10 milisekundi. Nakon toga je prekinuto izvršavanje niti `nit1` sa `Abort` metodom i startanje druge niti. Komentarima u kodu 13.3 su naznačeni koraci vezani za prekid izvršavanja niti.

```
static void Main(string[] args)
{
    BrojacNit nzad1 = new BrojacNit();
    BrojacNit nzad2 = new BrojacNit();

    Thread nit1 = new Thread(new ThreadStart(nzad1.brojac));
    Thread nit2 = new Thread(new ThreadStart(nzad2.brojac));

    nit1.Name = "Nit 1";
    nit2.Name = "Nit 2";

    // Startovana nit1
    nit1.Start();

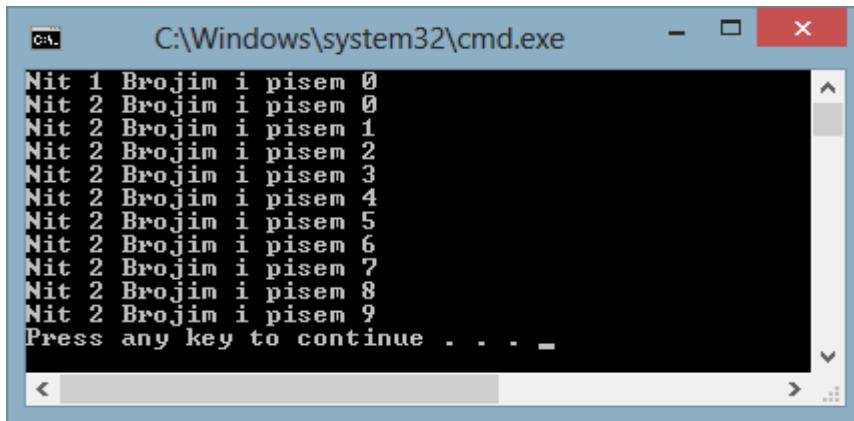
    // Usporavanje izvršavanja glavne niti, da bi nit1 uradila dio zadatka
    Thread.Sleep(10);

    // Prekid izvršavanja niti1
    nit1.Abort();

    // Startovana nit2
    nit2.Start();
}
```

Kod 13.3: Prekid izvršavanja niti sa `Abort` metodom

Scenarij izvršavanja koda 13.3:



```
Nit 1 Brojim i pisem 0
Nit 2 Brojim i pisem 0
Nit 2 Brojim i pisem 1
Nit 2 Brojim i pisem 2
Nit 2 Brojim i pisem 3
Nit 2 Brojim i pisem 4
Nit 2 Brojim i pisem 5
Nit 2 Brojim i pisem 6
Nit 2 Brojim i pisem 7
Nit 2 Brojim i pisem 8
Nit 2 Brojim i pisem 9
Press any key to continue . . . .
```

U ovom slučaju `nit1` je nakon startovanja a prije prekida njenog izvršavanja uspjela da izvrši jednu iteraciju for petlje. Druga nit je izvršila u potpunosti metodu `brojac` tj. sve iteracije for petlje. Mogao se desiti i drugačiji scenarij izvršavanja.

U okviru neke metode koja je pokrenula izvršavanje niti može se čekati da nit terminira korištenjem `Join` metode. Postoje preklopljene varijante ove metode.

Naprimjer, sa iskazima:

```
nid1.Join();
nid2.Join(new TimeSpan(0,0,1));
```

u okviru prethodno prikazane `main` metode postiglo bi se da glavna nit čeka prvu nit da u potpunosti završi svoj zadatak i nakon toga drugu nit jednu sekundu.

13.2.4. Pozadinska nit

Nit se može izvršavati kao pozadinska nit ili kao nit u prednjem planu. Izvršavanje pozadinske niti se završava kada glavna nit terminira, dok nit u prednjem planu se nastavlja izvršavati i nakon toga. Kako će se niti izvršavati se postavlja sa `IsBackground` osobinom. Ako je `IsBackground` opcija postavljena na `true` tada je nit pozadinska nit. Postavljanje osobine `IsBackground` je prikazano kodom 13.4.

```
static void Main(string[] args)
{
    BrojacNit nzad1 = new BrojacNit();
    BrojacNit nzad2 = new BrojacNit();

    Thread nit1 = new Thread(new ThreadStart(nzad1.brojac));
    Thread nit2 = new Thread(new ThreadStart(nzad2.brojac));

    nit1.Name = "Nit 1";
    nit2.Name = "Nit 2";

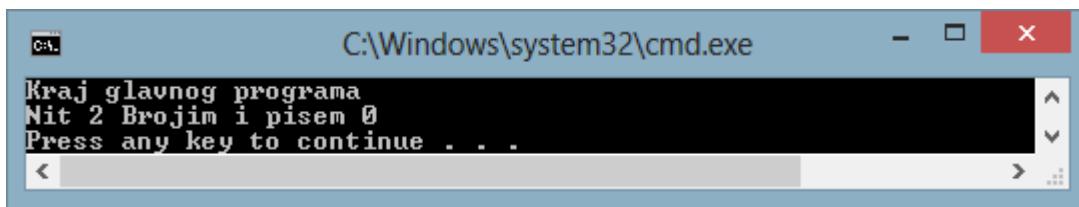
    // Postavljanje niti da se izvršavaju kao pozadinske niti
    nit1.IsBackground = true;
    nit2.IsBackground = true;

    // pokretanje niti u okviry try/catch strukture
    try
    {
        nit1.Start();
        nit2.Start();
    }
    catch (ThreadStateException te)
    {
        Console.WriteLine(te.ToString());
    }

    Console.WriteLine("Kraj glavnog programa");
}
```

Kod 13.4: Setovanje pozadinskih niti sa IsBackground osobinom

Scenarij izvršavanja:



S obzirom da su obadvije niti bile pozadinske niti, prilikom terminiranja glavne niti zaustavilo se njihovo izvršavanje.

13.2.5. Prioritet niti

Nitima se može dodijeliti prioritet izvršavanja. Nit sa većim prioritetom dobija više procesorskog vremena nego nit sa nižim prioritetom. Prioritet se može mijenjati sa `Priority` osobinom `Thread` klase, koja prihvata vrijednost iz `ThreadPriority` nabrojive liste. `ThreadPriority` lista sadrži vrijednosti `Lowest`, `BelowNormal`, `Normal` (podrazumijevana vrijednost), `AboveNormal` i `Highest`. Iskazom `nit1.Priority=ThreadPriority.Highest` se postavlja `Highest` (najveći) prioritet za `nit1`.

13.2.6. Predaja parametara niti

U primjerima prikazanim iznad nitima je dodjeljivan zadatak za izvršavanje koji je metoda bez parametara. Često je potrebno da se niti preda parametar odnosno da joj se dodijeli zadatak koji je metoda sa parametrima. Predaju parametara niti je najlakše ostvariti pomoću lambda izraza. Lambda izraz može specificirati poziv metode na sljedeći način:

```
() => Metoda()
```

Na jednoj strani su prazni parametri, a na drugoj je tijelo lambda izraza koje je poziv neke metode.

U primjeru datim sa kodom 13.5 metoda `broji` koja se delegira niti za izvršavanje ima parametar.

```
using System;
using System.Threading;

namespace NitSaParametrima
{
    // Klasa koja sadrži metodu koju će nit izvršavati
    public class BrojacNit
    {
        public long uBrojac;

        // Metoda koju će nit izvršavati, metoda ima parametar
        public long broji(long brojac)
        {
            uBrojac = brojac;
            Console.WriteLine("Počeo sa {0}", brojac);
            while (true)
            {
                brojac++;
                uBrojac++;
            }
        }
    }
}

// nastavak na sljedećoj stranici
```

RAZVOJ PROGRAMSKIH RJEŠENJA

```
public class Program
{
    public static void Main(string[] args)
    {
        //Kreiranje instance klase BrojacNit
        BrojacNit c1 = new BrojacNit();

        // Kreiranje niti, i dodjeljivanje zadatka, metode sa parametrom, korištenjem lambda izraza
        Thread nit1 = new Thread(new ThreadStart(() => c1.broji(1000)));

        // Postavljanje prioriteta niti
        nit1.Priority = ThreadPriority.Highest;

        nit1.Start(); // Pokretanje niti

        nit1.Abort(); // Prekidanje izvršavanja niti

        Console.WriteLine("Nit je izbrojala do: {0}", c1.uBrojac);
    }
}
```

Kod 13.5: Predaja parametara niti

Sa iskazom:

```
Thread nit1 = new Thread(new ThreadStart(()=>c1.broji(1000)));
```

niti je korištenjem lambda izraza dodijeljena metoda koja ima parametar. U principu `ThreadStart` konstruktor je pozvan bez parametara, ali kod koji se izvršava za prazne parametre je ustvari poziv metode `broji` koja ima parametar.

13.2.7. Sinhronizacija niti

Često, više niti pri izvršavanju koriste i dijele iste podatke. Ako je dijeljenje podataka neophodno, moraju se koristiti tehnike sinhronizacije. Tehnikama sinhronizacije može se jednoj niti dati ekskluzivni pristup manipulacije sa zajedničkim, dijeljenim podacima. Za to vrijeme, druge niti koje žele pristup podacima čekaju. Kada nit sa ekskluzivnim pristupom podacima završi svoj rad sa podacima omogućava se pristup podacima jednoj niti iz reda čekanja.

Postoji više mehanizama koji omogućavaju sinhronizaciju rada niti. Neki od njih su:

- lock iskaz
- Interlocked klasa
- Monitor klasa
- Mutex klasa
- Semaphore klasa

U ovom materijalu se obrađuje `lock` iskaz.

C# obezbjeđuju programsku konstrukciju `lock`, kojom se može sinhronizirati rad nad zajedničkim objektom. Sintaksa `lock` iskaza je:

RAZVOJ PROGRAMSKIH RJEŠENJA

```
lock ( referenciaObjekta )
{
    // Kod koji zahtjeva sinhronizaciju.
    // Naziva se i kritična sekcija.
    // Kritična sekcija je dio koda koji pristupa dijeljenom resursu.
    // Samo jedna nit može ući u ovu sekciju u jednom trenutku.
}
```

Kod 13.6 sadrži `lock` izraz. Potrebno je osigurati sinhronizaciju između dvije novčane transakcije sa istog računa. Ukoliko ne bi bilo sinhronizacije moglo bi se desiti da se omogući transakcija koja je neizvodljiva. Naprimjer, stanje računa je 300 KM, i postoje dvije transakcije koje podižu 200 KM sa računa. Ukoliko i jedna i druga transakcija počnu obavljanje podizanja novca sa računa sa početnim stanjem 300 KM, obadvije će biti omogućene i konačno stanje računa će biti negativno -100. Da se to ne bi desilo prva transakcija treba zaključati zajednički resurs (račun) dok se u potpunosti ne završi.

```
using System;
using System.Threading;
namespace Sinhronizacija
{
    // Klasa koja sadrži metodu koju će niti izvršavati
    class Racun
    {
        // Objekat koji služi za zaključavanje dijeljenih podataka i zaštitu kritične sekcije
        private Object kriticnaSekcija = new Object();

        // Članovi klase
        int stanjeRacuna; // stanje računa je dijeljeni podatak-zahtjeva ekskluzivni pristup
        int iznosTransakcije;

        // Konstruktor klase prima dva parametra: početno stanje računa i iznos transakcije
        public Racun(int pocetnoStanje, int piznosTransakcije)
        {
            stanjeRacuna = pocetnoStanje;
            iznosTransakcije = piznosTransakcije;
        }
        // Metoda kojom se vrši podizanje novca, sadrži lock na objekat
        public void podizanjeNovca()
        {
            lock (kriticnaSekcija) // ekskluzivni pristup za jednu nit kritičnoj sekciji
            {
                if (stanjeRacuna >= iznosTransakcije)
                {
                    Console.WriteLine("Stanje na računu prije podizanja novca : " + stanjeRacuna);
                    Console.WriteLine("Iznos za podizanje : -" + iznosTransakcije);
                    stanjeRacuna = stanjeRacuna - iznosTransakcije;
                    Console.WriteLine("Stanje na računu poslije podizanja novca : " + stanjeRacuna);
                }
                else
                    Console.WriteLine("Nema dovoljno novca na računu za transakciju");
            }
        }
    }
    // nastavak na sljedećoj stranici
}
```

RAZVOJ PROGRAMSKIH RJEŠENJA

```
class Program
{
    static void Main()
    {

        // Kreiranje instance klase Racun, stanje računa je 300, novčana transakcija 200
        Racun transakcija = new Racun(300, 200);

        // kreiranje niti t1 i niti t2
        // isti zadatak se dodijeljuje i za nit1 i nit2
        // sinhronizacija se obavlja u metodi podizanjeNovca klase Racun

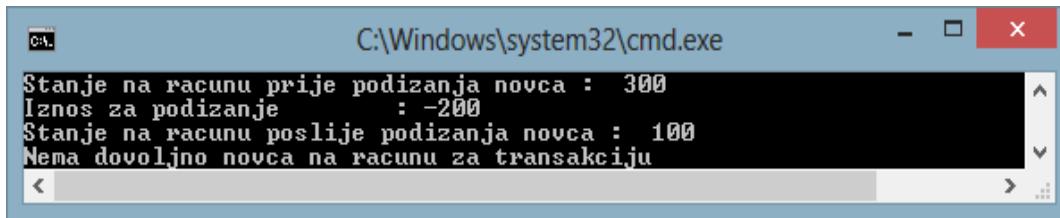
        Thread t1 = new Thread(new ThreadStart(transakcija.podizanjeNovca));
        t1.Start();

        Thread t2 = new Thread(new ThreadStart(transakcija.podizanjeNovca));
        t2.Start();

        Console.WriteLine("Kraj programa");
    }
}
```

Kod 13.6: Sinhronizacija niti sa `lock` iskazom

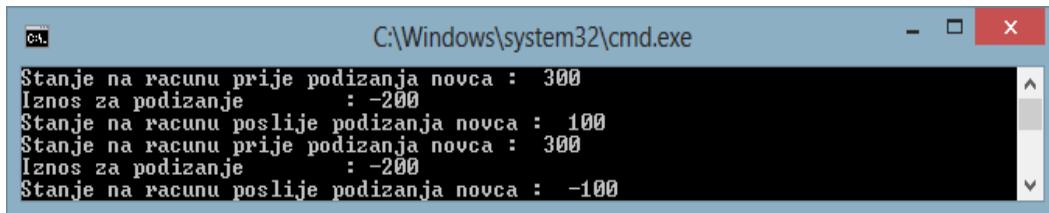
Scenarij izvršavanja koda 13.6:



```
C:\Windows\system32\cmd.exe
Stanje na racunu prije podizanja novca : 300
Iznos za podizanje : -200
Stanje na racunu poslije podizanja novca : 100
Nema dovoljno novca na racunu za transakciju
```

Analizom rezultata uočava se da je obavljena prva transakcija. Druga transakcija je čekala dok se ne završi prva transakcija. Prilikom izvršavanja druge transakcije ustanovljeno je da nema dovoljno novca na računu.

Ukoliko kod ne bi imao `lock` iskaz moguć je sljedeći scenarij izvršavanja:



```
C:\Windows\system32\cmd.exe
Stanje na racunu prije podizanja novca : 300
Iznos za podizanje : -200
Stanje na racunu poslije podizanja novca : 100
Stanje na racunu prije podizanja novca : 300
Iznos za podizanje : -200
Stanje na racunu poslije podizanja novca : -100
```

U ovom slučaju nije bilo sinhronizacije nad zajedničkim resursom što je dovelo do neželjenog scenarija.

13.3. Asinhroni mehanizmi

U okviru ove sekcije objašnjavaju se dodatni asinhroni mehanizmi: skupina niti, taskovi, `async/await` struktura. Ovim mehanizmima omogućeno je asinhrono izvršavanje, odnosno omogućeno je da metoda koja pozove drugu metodu nakon poziva nastavi da se izvršava. Kod sinhronog poziva metoda koja je pozvala drugu metodu čeka da se ta metoda izvrši, i tek nakon završetka izvršavanja pozvane metode nastavlja sa svojim izvršavanjem.

13.3.1. Skupina niti

Kreiranje niti je dosta skupa operacija, jer niti intenzivno koriste resurse sistema. Ima smisla da se nit koristi za izvršavanje kompleksnih operacija računanja, raznih pozadinskih operacija nadgledanja, ali nema smisla da se nit koristi za izvršavanje suviše jednostavnih operacija. Da se smanji broj kreacija niti i broj niti koje zauzimaju resurse, uvedena je forma grupiranja niti u skupinu niti, koja se pretežno koristi za izvršavanje operacija koje dugi traju a pri tome nije presudno njihovo vrijeme izvršavanja. Formiranje skupine niti (*thread pooling*) je forma višenitnosti u kojoj se taskovi dodaju u zajednički skup. Formiranje skupine niti pruža efikasniji način korištenja resursa, ali moguće je da se može čekati jako dugo da neka nit iz skupine niti izvrši neki zadatak. Formirana skupina niti se izvršava kao pozadinska nit.

`ThreadPool` klasa obezbjeduje metode i neke kontrole za rad sa skupinom niti. Jedna od njih je metoda `QueueUserWorkItem` koja dodaje nit u zajedničku skupinu. Kod 13.7 prikazuje kreaciju dvije niti u jednu skupinu sa `ThreadPool.QueueUserWorkItem` metodom. Nitima se dodjeljuje izvršavanje metode `brojac`. Metoda `brojac` se dodjeljuje sa `WaitCallback` delegatom. Nit se počinje izvršavati nakon što postanu raspoloživi resursi potrebni za izvršavanje niti.

```
using System;
using System.Threading;
namespace SkupinaNiti
{
    class Program
    {
        // Metoda koju nit iz skupine niti izvršava, info je parametar koji koristi WaitCallback delegat
        public static void brojac(object info)
        {
            long i;
            for (i = 0; i < 100; i++)
            {
                Console.WriteLine("Brojim i pišem " + i);
            }
        }
        static void Main(string[] args)
        {
            // Dodavanje niti u skupinu niti, i delegiranje zadatka za nit
            ThreadPool.QueueUserWorkItem(new WaitCallback(brojac));

            // Dodavanje druge niti u skupinu niti i delegiranje zadatka za nit
            ThreadPool.QueueUserWorkItem(new WaitCallback(brojac));
            Thread.Sleep(100);
        }
    }
}
```

Kod 13.7: Kreiranje skupine niti korištenjem ThreadPool klase

13.3.2. Task

Task biblioteka obezbjeđuje apstraktни aplikacijski interfejs za niti koje su organizirane u skupinu niti. Klasa koja podržava rad sa taskovima je Task smještena u System.Threading.Tasks. Postoji više preklopljenih konstruktora kojima se može kreirati task i izvršiti dodjeljivanje zadatka tasku. Zadatak dodijeljen niti može biti metoda, akcijski delegat, iskaz i slično. Task niti se izvršava kao pozadinska nit.

Kreiranje i pokretanje asinhronog taska se obavlja sa iskazima:

```
Task task1 = new Task(() => brojac());
task1.Start();
```

ili task se može kreirati i odmah pokrenuti sa Factory.StartNew ili Run metodom:

```
Task task2=Task.Factory.StartNew(() => brojac());

Task task3=Task.Run(() => brojac());
```

U primjeru (kod 13.8) ilustrirana je upotreba Task klase. Pored metoda Factory.StartNew, Run koristile su se i metode RunSynchronously i Wait. RunSynchronously omogućava da se nit

RAZVOJ PROGRAMSKIH RJEŠENJA

pokrene kao sinhrona nit. Sa `Wait` metodom se može postići da nit koja je pokrenula neku nit čeka da se ta nit završi prije prelaska na sljedeći iskaz.

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace NitPool
{
    class Program
    {
        // Metoda koju taskovi izvršavaju
        public static void brojac()
        {
            long i;
            for (i = 0; i < 100; i++)
            {
                Console.WriteLine("Brojim i pišem " + i);
            }
        }

        static void Main(string[] args)
        {

            // I. Način pokretanja taska
            Task task1 = new Task(() => brojac());
            task1.Start();

            // II. Način pokretanja taska
            Task task2=Task.Factory.StartNew(() => brojac());
            task2.Wait(); // glavna nit čeka dok se task2 ne završi

            // III. Način pokretanja taska
            Task task3=Task.Run(() => brojac());

            // IV. Sinhroni način pokretanja niti
            Task task4 = new Task(() => brojac());
            task4.RunSynchronously(); // glavna nit čeka dok se task4 ne završi
        }
    }
}
```

Kod 13.8: Razni načini pokretanja taska

Taskovi sa povratnom vrijednosti

Ako se poziva metoda sa povratnom vrijednosti, potrebno obezbjediti da se prihvati povratni rezultat. Kodom 13.9 se ilustrira poziv niti koja vraća vrijednost. Rezultat se prihvata sa `task.Result`, pri čemu je `task` naziv taska koji je vratio povratnu vrijednost.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace NitPool
{

    class Program
    {

        // Metoda koju task izvršava, metoda vraća povratnu vrijednost
        public static long brojac()
        {
            long i;
            for (i = 0; i < 10000000; i++) ;
            return i;
        }

        static void Main(string[] args)
        {

            //1. Kreiranje taska koji vraća vrijednost
            var task = Task<long>.Factory.StartNew(() => brojac()); ;

            //2. Prihvatanje rezultata u varijablu rez
            long rez = task.Result;

            Console.WriteLine(rez);

        }
    }
}
```

Kod 13.9: Kreiranje taska koji vraća rezultat i prihvata rezultat sa `Result` osobinom

13.3.3. Asinhroni pristup sa `async/await`

C# ključne riječi `async` i `await` omogućavaju da se jednostavno programiraju asinhronne metode. `await` izraz se koristi unutar asinhronne metode koja se obilježava sa `async`. Metoda koja je pozvala asinhronu metodu se poslije poziva nastavlja izvršavati. Istovremeno asinhrona metoda izvršava svoj zadataka na istoj ili različitoj niti.

Da bi se metoda izvršavala kao asinhrona metoda potrebno je slijediti neka pravila i preporuke:

- Ključna riječ `async` se dodaje u deklaracijsko zaglavljje metode prije povratnog tipa.
- Asinhrona metoda može imati bilo koji broj formalnih parametara. Nijedan od njih ne može biti `out` ili `ref`.
- Preporučuje se da se naziv asinhronih metoda završava sa sufiksom `Async`.
- Asinhrona metoda može vratiti objekat tipa `Task` sa ili bez dodatno specificirane povratne vrijednosti. Povratni tip `Task` omogućava metodi koja je pozvala asinhronu metodu da provjeri njeno stanje. Ako je dovoljno da se samo izvrši poziv asinhronne metode bez buduće interakcije sa tom metodom, asinhrona metoda može imati `void` povratni tip podataka.

RAZVOJ PROGRAMSKIH RJEŠENJA

Primjer zaglavlja asinhronne metode koja vraća rezultat:

```
public async Task<int> RacunajSumuAsync( int broj1, int broj2)
```

Ukoliko je metoda članica klase RadiAinhrono tada se može pozvati sa:

```
RadiAinhrono zadatak = new RadiAinhrono();
Task <int> izacunataSuma = zadatak.RacunajSumuAsync(7,5) ;
```

-Asinhrona metoda mora sadržavati barem jedan await izraz. Izraz await označava zadatke koji će se izvršiti asinhrono. Sintaksa await izraza je prikazana ispod i sastoji se od await ključne riječi iza koje slijedi objekat, koji poziva zadatak koji se treba izvršiti. Taj zadatak može ali ne mora biti objekat tipa Task. Task se izvršava asinhrono na tekućoj niti.

Primjeri upotrebe await izraza koji asinhrono izvršavaju iskaze ili pozivaju metode:

```
int suma = await Task.Run(()=>broj1 + broj2);

int rezultat = await Task.Run(()=>suma(broj1,broj2));

await Task.Run(() => Console.WriteLine(rez.ToString()));
```

-Asinhrona metoda može sadržavati i dijelove koda prije i poslije await izraza. To su obično dijelovi koda koji sadrže informacije o tome koja je nit aktivna, vrijednosti varijabli u tekućem vidokrugu i ostale iskaze koji su bitni prije i iza izvršavanja await označenog zadatka.

Unutar koda 13.10 se poziva asinhrona metoda RacunajSumuAsync, koja asinhrono izvršava računanje sume dva broja. Komentarima u kodu su naznačeni bitni dijelovi koda za deklaraciju asinhronne metode, await izraz, poziv asinhronne metode.

```
using System;
using System.Threading.Tasks;

namespace AsinhAwait
{
    class RadiAinhrono
    {

        // deklaracija asinhronne metode (naznačeno sa async)
        public async Task<int> RacunajSumuAsync( int broj1, int broj2)
        {
            //asinhroni (naznačeno sa await) poziv operacije sabiranja
            int suma = await Task.Run(()=>broj1 + broj2);

            return suma;
        }
    }

    // glavni program na sljedećoj strani
}
```

```
class Program
{
    static void Main(string[] args)
    {
        //1. Kreiranje instance klase RadiAsinhrono
        RadiAsinhrono zadatak = new RadiAsinhrono();

        //2.poziv asinhronne metode koja vraća rezultat
        Task <int> izracunataSuma = zadatak.RacunajSumuAsync(7,5) ;

        //3. prikaz rezultata sa Result osobinom klase Task
        Console.WriteLine("Suma je {0}", izracunataSuma.Result);
    }
}
```

Kod 13.10: Poziv i implementacija asinhronne metode RacunajSumuAsync

Prekid async operacije

Može se izvršiti prekid .NET asinhronih metoda. Postoje dvije klase u System.Threading.Tasks koje su dizajnirane za ovu namenu: CancellationToken i CancellationTokenSource. Task koji ima CancellationToken objekt periodično provjerava stanje tokena. Ako je osobina IsCancellationRequested, CancellationToken objekta true, task završava izvršavanje i vraća se u metodu koja je izvršila poziv asinhronne metode. U svrhu ilustracije mogućeg scenarija prekida asinhronne metode slijedi dio koda u metodi (`main`) koja je pozvala asinhronu metodu i dio koda u asinhronoj metodi.

Dio koda u `main` metodi:

```
CancellationTokenSource cts = new CancellationTokenSource();
CancellationToken token = cts.Token;
Task <int> value = ins.RacunajSumuAsync(7,5) ;
cts.Cancel();
```

Dio koda u asinhronoj metodi:

```
if (token.IsCancellationRequested)
    throw new OperationCanceledException(token);
```

13.4. GUI i asinhronie operacije

U okviru ove sekcije izlaže se pristup asinhronih operacija u okviru Windows Forms aplikacija i iniciranje osnovnih vlasničkih niti za upravljanje kontrolama.

13.4.1. Korisnički interfejs i asinhroni poziv metoda

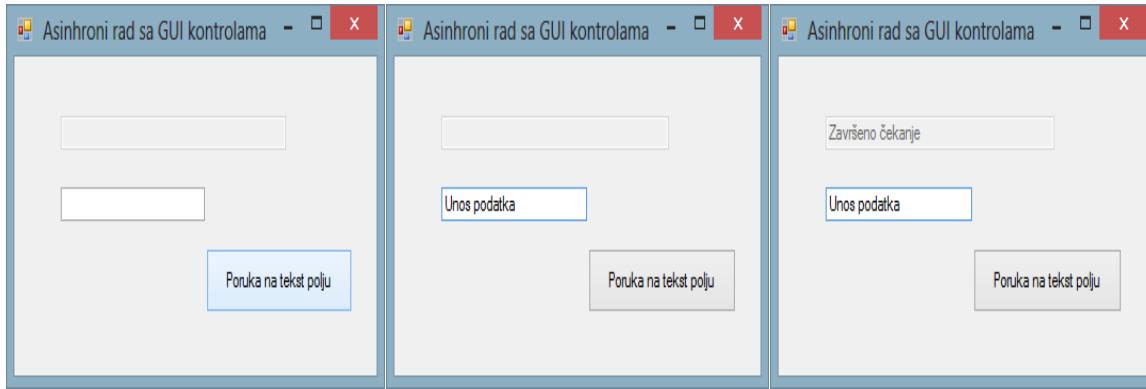
Korisnički interfejs treba da omogući korisniku programskog rješenja stalnu interakciju. Nije poželjno da korisnik čeka da se završi neka akcija i da ima osjećaj da se interfejs 'zaledio'. Na primjeru jednostavne forme se ilustrira koncept asinhronog poziva metode. Forma sadrži dva teksta polja i dugme „Poruka na tekst polje“. Nakon pritiska na dugme, aktivira se događaj, koji ispisuje poruku „Završeno čekanje“ na prvo teksto polje, nakon čekanja od 3000 milisekundi. Drugo teksto polje služi za unos. Cilj je korisniku omogućiti unos na drugo tekstualno polje dok se obrađuje opisani događaj dugmeta „Poruka na tekst polje“. To se može postići sa `async/await` asinhronim pristupom. Unutar klik događaja pomenutog dugmeta može se asinhrono pozvati metoda `CekajAsyn()` u kojoj se implementira čekanje. Na taj način korisnik može da nastavi da radi, omogućen je fokus na sve ostale kontrole forme, u ovom slučaju na drugo teksto polje. Detalji implementacije dati su u kodu 13.11.

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;
namespace GUIAsinhroniPoziv
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        // klik događaj dugmeta Prikazi poruku na tekstu polje-sadrži asinhroni poziv
        private async void asinhroniButton_Click(object sender, EventArgs e)
        {
            // Poziva asinhronu metodu CekajAsyn
            string result = await CekajAsyn();

            // Prikazuje rezultat na prvom tekstu polju
            textBox1.Text += result;
        }
        // Metoda CekajAsync radi asinhrono. UI nit nije blokirana za vrijeme čekanja
        // Moguć je nastavak korisničke interakcije preko GUI-a.
        public async Task<string> CekajAsyn()
        {
            await Task.Delay(30000); // asinhrono izvršavanje metode Task.Delay
            return "Završeno čekanje";
        }
    }
}
```

Kod 13.11: Asinhroni poziv metode `CekajAsyn` u metodi za upravljanje događajem

Scenarij izvršavanja:



Klik na dugme

Obraduje se dogadjaj, korisnik nastavlja da radi Obraden dogadaj

Da se nije izvršio asinhroni poziv metode za implementaciju čekanja (`CekajAsync`), ulazno-izlazna nit bi bila blokirana i korisnik ne bi imao fokus na drugo tekst polje, što bi uzrokovalo čekanje dok se događaj ne obradi i osjećaj da je u pitanju neki problem.

13.4.2. Iniciranje osnovne niti GUI kontrole

Ako postoje dvije ili više niti koje manipuliraju nekom kontrolom, moguće je da se ta kontrola nađe u nekonzistentnom stanju. U okviru neke niti dobro je da se sam rad sa GUI kontrolom obavi sa niti koja je kreirala tu kontrolu. Ta nit se naziva glavna, osnovna, vlasnička nit. Metoda `Invoke` omogućava da se inicira osnovna (vlasnička) nit forme u svrhu izvršavanja neke zahtjevane akcije nad kontrolom od strane drugih niti.

Sljedeći primjer (kod 13.12) prikazuje iniciranje osnovne vlasničke niti dugme kontrole. Kreirana je forma sa jednim dugmetom "Pokreni", jednom `numericUpDown` kontrolom i jednim tekst poljem. Kada se pokrene dugme "Pokreni" tada se inicira događaj koji će računati sumu brojeva do broja unesenog na `numericUpDown` kontrolu. Međurezultati sumiranja se prikazuju na tekstualnom polju. Računanje sume se obavlja u okviru druge niti. Za prikazivanje rezultata sumiranja na tekstualnom polju koristi se `Invoke` metoda koja inicira glavnu nit da prikaže rezultat.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
using System.Threading;
using System.Windows.Forms;

namespace InvokePoziv
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private Thread sumiranje;

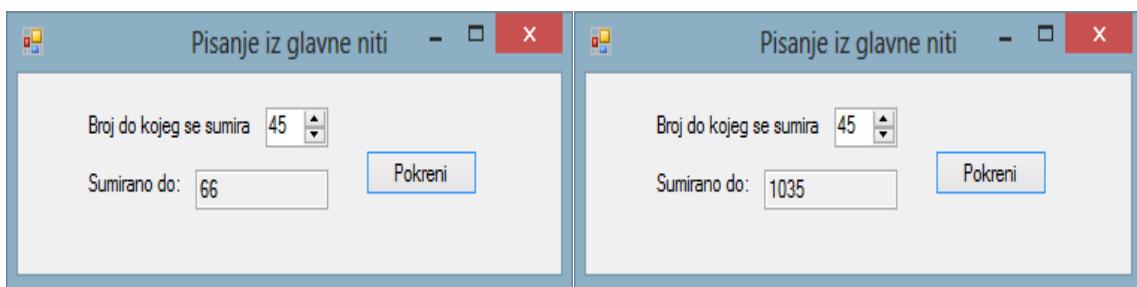
        // događaj za dugme Pokreni, koji na drugoj niti izvršava zadatak
        private void pokreniButton_Click(object sender, EventArgs e)
        {
            // kreiranje niti i dodjeljivanje zadatka niti
            sumiranje = new Thread(() => racunajSumu((long)brojNumericUpDown1.Value));
            sumiranje.IsBackground = true;
            sumiranje.Start();
        }

        // metoda za računanje sume
        private void racunajSumu(long doBroja)
        {
            long suma = 0;
            for (int i = 0; i <= doBroja; i++)
            {
                suma += i;

                Thread.Sleep(200); // usporava sumiranje
                // inicira se vlasnička niti za formu da postavi trenutnu sumu na GUI kontrolu
                this.Invoke(new Action(() => { this.rezulatTextBox2.Text = suma.ToString(); }));
            }
        }
    }
}
```

Kod 13.12: Iniciranje osnovne, vlasničke niti `Invoke` metodom

Scenarij izvršavanja:



Unesen broj do kojeg se sumira, i izvršen klik na Pokreni, među/rezultat na tekstualnom polju

Da se na formi nalazi još kontrola bilo bi omogućena interakcija sa tim kontrolama, jer je implementacija sumiranja izvršena korištenjem niti.

Rezime:

U okviru ovog poglavlja prikazani su mehanizmi koji se koriste za kreiranje više niti u programu. Višenitnost je jako bitna za kreiranje programskih rješenja, jer sa jedne strane programska rješenja moraju da pružaju mnoge funkcionalnosti, a sa druge korisnici žele da pokreću više njih istovremeno. Često su zadaci koje izvršavaju niti međusobno ovisni. U poglavlju su obrađeni mehanizmi sinhronizacije. Obrađeni su i mehanizmi asinhronog poziva, GUI višenitnosti. Postoji još mehanizama za paralelno izvršavanje programskih iskaza koji nisu obrađeni u ovom poglavlju.

Pitanja i zadaci za samostalni rad

1. Objasnite kako se odvija sinhrono izvršavanje metoda.
2. Navedite kojim mehanizmima se može programirati paralelno izvršavanje više zadataka.
3. Šta je nit?
4. Koji .NET *namespace* omogućava rad sa nitima?
5. Navedite osnovnu klasu za rad sa nitima.
6. Objasnite način kreiranja i dodjeljivanja zadatka programskej niti.
7. Kojom metodom se starta niti?
8. Ukoliko se istovremeno izvršava više niti, da li se može desiti scenarij da `main` metoda terminira prije ostalih programskih niti.
9. Prilikom izvršavanja programa sa više niti da li je scenarij izvršavanja uvijek isti.
10. Kako se može izvršiti pauziranje niti?
11. Navedite neke izuzetke koji se mogu desiti prilikom poziva `Sleep` metode.
12. Kojom metodom se vrši prekid izvršavanja niti?
13. Šta se postiže sa `Join` metodom?
14. Objasnite koja je razlika između pozadinske niti i niti u prednjem planu.
15. Kojom osobinom se može postaviti nit kao pozadinska nit?
16. Šta se postiže dodjeljivanjem prioriteta niti?
17. Navedite opcije `ThreadPriority` liste.
18. Objasnite kako se mogu dodijeliti parametri niti.
19. Objasnite ulogu lambda izraza u procesu dodjeljivanja parametara niti.
20. Ukoliko je potrebno da se dijele podaci između više niti, da bi se dijeljeni podaci održali u konzistentnom stanju koje tehnike su neophodne?
21. Navedite šta se postiže tehnikama sinhronizacije.
22. Navedite mehanizmi u okviru .NET za sinhronizaciju podataka.
23. Objasnite `lock` iskaz.
24. Navedite primjer kojim se ilustruje lock iskaz i ekskluzivni pristup podacima od strane jedne niti.
25. Objasnite šta je skupina niti i kako se kreira u okviru .NET-a.

RAZVOJ PROGRAMSKIH RJEŠENJA

26. Kada se preporučuje upotreba skupine niti a kada niti.
27. Navedite šta predstavlja task i kako se kreira u okviru .NET-a.
28. Prikažite načine startovanja taska.
29. Šta se može postići sa `async/await` C# ključnim rječima?
30. Koja pravila i preporuke treba slijediti da bi se metoda izvršavala kao asinhrona metoda?
31. Kako se unutar neke niti inicira osnovna nit kontrole?
32. Zašto je bitno inicirati osnovnu nit kontrole ako više niti pristupa istoj kontroli?
33. U okviru programskog rješenja urađenog za zadatak 35 u okviru poglavlja 11 kreirati nit koja će računati ukupnu naplatu od prodaje karata. Nit se pokreće preko dugmeta PokreniNit sa osnovne forme. Međurezultate i konačni rezultat računanja prikazivati na nekoj kontroli formi.
34. Kreirati nit koja će se pokrenuti sa klikom na dugme Start na nekoj formi i koja će imati zadatak da izbroji koliko je XML dokumenata na c: drajvu. Nit se zaustavlja kada obavi cjelokupno brojanje ili nakon pritiska na dugme Stop na istoj formi. Nakon zaustavljanja niti prikazati koliko je XML dokumenata pronađeno.

Reference

1. Ian Sommerville, Software Engineering (9th Edition), Addison-Wesley, 2010
2. Vidya Vrat Agarwa, Beginning C# 5.0 Databases, Apress, 2012
3. Dan Clark, Beginning C# Object-Oriented Programming, Second Edition, Apress, 2013
4. Jack Purdum , Beginning Object-Oriented Programming with C#, Wrox Press, 2013
5. Karli Watson, Beginning Visual C# 2012 Programming, Wrox Press, 2013
6. Rick Miller, C# Collections: A Detailed Presentation, Pulp Free Press, 2012
7. Jayden Ky, C#: A Beginner's Tutorial, Brainy Software Corp, 2013
8. John Dooley, Software Development and Professional Practice, 2011
9. Bruce Johnson, Professional Visual Studio 2012, Wrox Press, 2013
10. Bharat Bhushan Agarwa, Sumit Prakash Tayam, Software Engineering, Laxmi Publications 2009
11. Jeff Johnsons, Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rule, Morgan Kaufmann Publishers, 2010
12. Wilbert O. Galitz, The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques, John Wiley&Sons, 2007
13. Joe Fawcett, Liam R.E. Quin and Danny Ayers, Beginning XML, 5th Edition, Wrox Press, 2012
14. Trey Nash, Accelerated C#, Apress, 2010
15. Robert W. Sebesta Concepts of Programming Languages (10th Edition) Addison-Wesley, 2012
16. Jenifer Tidwel, Designing Interfaces, O'Reilly Media 2011
17. Saul Greenberg, Sheelagh Carpendale, Nicolai Marquardt, User Experiences: The Workbook Paperback –ecember 28, 2011by, Morgan Kaufmann, 2011
18. Steve Krug, Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability, New Riders, 2014

19. Joseph Albahari, Ben Albahari, C# 5.0 in a Nutshell: The Definitive Reference, O'Reilly Media, 2012
20. Richard Blewettm, Andrew Clymer, Pro Asynchronous Programming with .NET, Apress, 2013
21. Matt Weisfeld, The Object-Oriented Thought Process, Addison-Wesley Professional, 2013
22. Dan ClarkBeginning C# Object-Oriented Programming, Apress; 2013
23. Michael T. Goodrich, Roberto Tamassia, Data Structures and Algorithms, Wiley; 2011
24. Adna Oputić, Dženana Đonko-mentor, Završni rad „ Smjernice za dobar dizajn korisničkog interfejsa“, ETF, 2012
25. <https://msdn.microsoft.com>

PRILOZI

Prilog 1: Primjer konzolne aplikacije

U okviru ovog priloga dat je primjer konzolne aplikacije. Ovim primjerom se ilustrira način uočavanja klasa, definicija klasa, hijerarhija klasa, programska dekompozicija.

Tekst problema

Napisati konzolnu aplikaciju koja će omogućiti knjižari M iznajmljivanje knjiga. Knjige se opisuju nazivom, autorom, godinom izdavanja i identifikacijskim brojem, tipom knjige. Za sada postoje dva tipa knjiga Naučna i Zabavna. Ukoliko je knjiga Naučna zapisuje se i grana nauke, a ukoliko je Zabavna dodatno se zapisuje za koju starosnu dob je namijenjena. Knjige naučnog tipa se mogu iznajmiti do 10 dana za cijenu od 5 KM, a za svaki dodatni dan se plaća 0.5 KM ukoliko je godina izdavanja knjige prije 1990, a ako je poslije 1990 plaća se 0.3KM. Knjige zabavnog tipa se mogu iznajmiti do 5 dana za cijenu od 4 KM, a za svaki dodatni dan se plaća 2% od osnovne cijene ukoliko je starosnu dob iznad 18 godina.

Programko rješenje treba omogućiti unos podataka o knjigama preko odgovarajućeg menija i obračun na zahtjev osobe koja želi iznajmiti knjigu. Osoba navodi koju knjigu želi i koliko dana je želi zadržati a program ispisuje koliko je potrebno platiti za iznajmljivanje te knjige. Definirajte klase i veze između klasa koje će odgovarati opisanom problemu.

S obzirom da se očekuje da će knjižara M uvesti i nove tipove knjiga i obračune, da bi se lakše dizajnirati hijerarhiju klasa tako da je na vrhu hijerarhije apstraktna klasa.

Potrebno je primjenom programske dekompozicije napisati program, raditi u programskom jeziku C# i obavezno koristiti polimorfizam. Testirati program tako da se evidentira po 3 knjige za svaki tip (voditi računa o godini izdavanja i starosnoj dobi) i nakon toga da se i vrši njihove iznajmljivanje na 3 i 7 dana.

P1.1.Analiza problema i dizajn rješenja

U prvom koraku iz postavke problema identificiraju se klasе. Imenice koje se spominju su: knjižara, knjiga (u raznim varijantama) i osoba. Osoba otpada kao potencijalna klasа zbog činjenice da se od programskog rješenja nigdje u postavci ne traži unos klijenata i njihova pohrana, iako u izračunima figurira starost osobe. Mehanizam zaduživanja knjiga svodi se na obračun najma, dok se nigdje ne bilježi raspoloživost (količinska) određenog izdanja neke knjige. Dakle, klasе su knjižara kao kolekcija raznih knjiga, te knjiga u njene dvije varijante.

RAZVOJ PROGRAMSKIH RJEŠENJA

Knjige se specijaliziraju u naučne i zabavne knjige što je očigledno mjesto gdje se koristi nasljeđivanje, dok je knjižara sada polimorfna kolekcija. Imajući u vidu i eksplisitne zahtjeve da je knjiga apstraktna klasa, klase i njihovi atributi su:

- + Knjižara (lista knjiga: lista objekata tipa knjiga)
- + Knjiga (ID: int, autor:string, naziv:string, godina izdavanja:int/datetime)
- + NaučnaKnjiga: Knjiga (grana nauke: string)
- + ZabavnaKnjiga: Knjiga (cijena dob: int)

Sada je potrebno razmotriti mehanizam naplate: podaci koji se koriste prilikom naplate iznajmljivanja knjige su: starosna dob osobe koja iznajmljuje knjigu i broj dana na koje se želi iznajmiti knjigu (u općem slučaju).

Pošto postoje različiti načini obračuna za različite tipove knjiga, potrebna je apstraktna metoda `izracunajIznosZaNaplatu` koju će svaka specijalizacija knjige (izvedene klase) implementirati na sebi svojstven način.

Pseudokod za način naplate:

- za naučne knjige, if `brojDana≤10` `cijena=5` else { if `izdanje≤1990` `cijena=5+prekoračenje*0.5` else `cijena=5+prekoračenje*.3`}
- za zabavne knjige, if `brojDana≤5 || dob≤18` `cijena =4` else if `dob≥18` `cijena=4+prekoračenje*0.08`

P1.2.Implementacija rješenja

Potrebno je kreirati konzolni meni koji će sadržavati minimalno tri opcije: registracija zabavne knjige, registracija naučne knjige i obračun iznosa za naplatu (eventualno i izlaz iz aplikacije).

Prvo slijedi implementacija klase, kod P1.1 sadrži apstraktну klasu `Knjiga` i dvije izvedne klase `KnjigaNaučna` i `KnjigaZabavna`. Sama implementacija je pojednostavljena, npr. nije implementirana provjera novih vrijednosti koje se dodjeljuje atributima u `setterima` - npr. da ne bude moguće postaviti `null` naziv ili godinu izdanja iz budućnosti i slično. Navedene provjere se ne traže eksplisitno zadatkom ali se u izradi stvarnih aplikacija podrazumijevaju. Čitaocima (studentima) se ostavlja za vježbu da implementiraju dodatne provjere.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
public abstract class Knjiga
{
    public int ID { get; set; }
    public String Naziv { get; set; }
    public String Autor { get; set; }
    public DateTime GodinaIzdanja { get; set; }

    public Knjiga(int ID, String Naziv, String Autor, DateTime GodinaIzdanja)
    {
        this.ID = ID;
        this.Naziv = Naziv;
        this.Autor = Autor;
        this.GodinaIzdanja = GodinaIzdanja;
    }
    abstract public Decimal izracunajIznosZaNaplatu(int brojDanaNajma, int starostOsobe);
}
```

Kod P1.1a: Apstraktna klasa Knjiga

Slijedi kod P1.1b za izvedenu klasu KnjigaNaucna.

```
public class KnjigaNaucna : Knjiga
{
    public String GranaNauke { get; set; }
    private static int maxDanaBezDodatneNaplate = 10;
    private static Decimal baznaCijena = 5;
    private static int granicnaGodinaIzdanja = 1990;
    private static Decimal koeficijentZaStarijeOdGranice = 0.5m;
    private static Decimal koeficijentZaMladjeOdGranice = 0.3m;

    public KnjigaNaucna(int ID, String Naziv, String Autor, DateTime GodinaIzdanja,
String GranaNauke) :
        base(ID, Naziv, Autor, GodinaIzdanja)
    {
        this.GranaNauke = GranaNauke;
    }

    override public Decimal izracunajIznosZaNaplatu(int brojDanaNajma, int starostOsobe)
    {
        if (brojDanaNajma <= 0)
            throw new ArgumentException("Broj dana ne može biti negativan broj!");
        if (starostOsobe <= 0)
            throw new ArgumentException("Starost osobe ne može biti negativan broj!");

        int diff = brojDanaNajma - maxDanaBezDodatneNaplate;
        if (diff <= 0)
            return baznaCijena;
        else
        {
            return baznaCijena + diff * ((GodinaIzdanja.Year <= granicnaGodinaIzdanja) ?
koeficijentZaStarijeOdGranice : koeficijentZaMladjeOdGranice);
        }
    }
}
```

Kod P1.1b: Izvedena klasa KnjigaNaucna

RAZVOJ PROGRAMSKIH RJEŠENJA

Kod P1.c prikazuje izvedenu klasu KnjigaZabavna

```
public class KnjigaZabavna : Knjiga
{
    public int CiljanaDob { get; set; }

    private static int maxDanaBezDodatneNaplate = 5;
    private static Decimal baznaCijena = 4;
    private static Decimal koeficijentZaPrekoracenje = 0.08m;
    private static int granicnaDobOsobe = 18;

    public KnjigaZabavna(int ID, String Naziv, String Autor, DateTime GodinaIzdanja, int CiljanaDob) :
        base(ID, Naziv, Autor, GodinaIzdanja)
    {
        this.CiljanaDob = CiljanaDob;
    }

    override public Decimal izracunajIznosZaNaplatu(int brojDanaNajma, int starostOsobe)
    {
        if (brojDanaNajma <= 0)
            throw new ArgumentException("Broj dana ne može biti negativan broj!");
        if (starostOsobe <= 0)
            throw new ArgumentException("Starost osobe ne može biti negativan broj!");

        int diff = brojDanaNajma - maxDanaBezDodatneNaplate;
        if (diff <= 0 || starostOsobe <= granicnaDobOsobe)
            return baznaCijena;
        else
        {
            return baznaCijena + diff * koeficijentZaPrekoracenje;
        }
    }
}
```

Kod P1.1c: Izvedena klasa KnjigaZabavna

Nakon definiranja klase i uspostave osnovnog logičkog modela, definira se kontejnerska klasa Knjižara. Ova klasa je najbolje rješenje za ovaj tip zadatka jer se podaci smještaju u listu (nema direktnog unosa u bazu podataka ili datoteku). Kod P1.2 prikazuje kontejnersku klasu Knjizara.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
public class Knjizara
{
    private List<Knjiga> knjige;

    public Knjizara()
    {
        knjige = new List<Knjiga>();
    }

    public void registrirajKnjigu(Knjiga novaKnjiga)
    {
        if (novaKnjiga == null)
            throw new ArgumentException("Parametar ne smije biti null");

        // provjera da li već postoji ista knjiga
        if (knjige.Any(postojecaKnjiga => postojecaKnjiga.ID == novaKnjiga.ID))
        {
            throw new ArgumentException("Knjiga već unesena");
        }

        knjige.Add(novaKnjiga);
    }

    public Decimal iznosNajmaKnjige(int IDKnjige, int brojDana, int starostOsobe)
    {
        Knjiga registriranaKnjiga = knjige.FirstOrDefault(knjiga => knjiga.ID == IDKnjige);
        if (registriranaKnjiga == null)
            throw new ArgumentException("Ne postoji knjiga sa datim ID!");
        return registriranaKnjiga.izracunajIznosZaNaplatu(brojDana, starostOsobe);
    }
}
```

Kod P1.2: Kontejnerska klasa Knjizara

Slijedi dio koda P1.3 za izbornik (meni) koji ujedno predstavlja i glavni program. Napomena: glavni program se sigurno može bolje dizajnirati, a nisu ni implementirane validacije ulaznih podataka (npr. ako se umjesto očekivane cijelobrojne vrijednosti unese slovo i slično).

RAZVOJ PROGRAMSKIH RJEŠENJA

```
public class Knjizara
{
    private List<Knjiga> knjige;

    public Knjizara()
    {
        knjige = new List<Knjiga>();
    }

    public void registrirajKnjigu(Knjiga novaKnjiga)
    {
        if (novaKnjiga == null)
            throw new ArgumentException("Parametar ne smije biti null");

        // provjera da li već postoji ista knjiga
        if (knjige.Any(postojecaKnjiga => postojecaKnjiga.ID == novaKnjiga.ID))
        {
            throw new ArgumentException("Knjiga već unesena");
        }

        knjige.Add(novaKnjiga);
    }

    public Decimal iznosNajmaKnjige(int IDKnjige, int brojDana, int starostOsobe)
    {
        Knjiga registriranaKnjiga = knjige.FirstOrDefault(knjiga => knjiga.ID == IDKnjige);
        if (registriranaKnjiga == null)
            throw new ArgumentException("Ne postoji knjiga sa datim ID!");
        return registriranaKnjiga.izracunajIznosZaNaplatu(brojDana, starostOsobe);
    }
}
```

Kod P1.3: Kontejnerska klasa Knjizara

RAZVOJ PROGRAMSKIH RJEŠENJA

RAZVOJ PROGRAMSKIH RJEŠENJA

Diskusija:

U ovom zadatku se moglo tražiti da se u rješenju obavezno iskoristi i interfejs. Ukoliko se podsjetimo činjenice da interfejsi služi kao ugovor kojeg klase mogu prihvatiti (implementirati) i onda je klasa obavezna imati metode (i osobine, događaje) specificirane interfejsom. U ovom slučaju može se razmišljati o interfejsu `iNaplativo` koji bi imao metodu naplati, međutim njeni parametri predstavljaju problem. Naime, `iNaplativo` je toliko općenito da se odnosi na bilo što što je moguće naplatiti tj. uvjeti pod kojima se obavlja naplata knjiga u postavljenom zadatku je izuzetno specifična (traži isključivo broj dana najma i starost osobe). Interfejs u takvima uvjetima nije dovoljno generaliziran, pa ga nebi ga trebalo nazvati `iNaplativo` već nešto poput `iNaplativoNaOsnovuStarostiOsobeIBrojaDanaNajma`. Ovaj interfejs bi se mogao odnositi i npr. na razne vrste iznajmljivanja (stanova, skija, i sl.).

Implementacija je predstavljena kodom P1.4 - deklarira se interfejs sa metodom naplati koja prima potrebne parametre (dvije cjelobrojne vrijednosti) i vraća decimalni broj (iznos za naplatu), a potom Knjiga implementira interfejs i deklarira apstraktnu metodu naplati iz interfejsa koja je `public` i onda se u izvedenim klasama vrši *override* metode dok se kôd metoda suštinski ne mijenja. Može se uvesti da knjižara ne sadrži listu knjiga već listu `iNplativih` objekata, ali uz određena ograničenja.

```
public interface iNaplativoNaOsnovuStarostiOsobeIBrojaDanaNajma
{
    Decimal izracunajIznosZaNaplatu(int brojDanaNajma, int starostOsobe);
}
public abstract class Knjiga : iNaplativoNaOsnovuStarostiOsobeIBrojaDanaNajma
{

    public int ID { get; set; }
    public String Naziv { get; set; }
    public String Autor { get; set; }
    public DateTime GodinaIzdanja { get; set; }

    public Knjiga(int ID, String Naziv, String Autor, DateTime GodinaIzdanja)
    {
        this.ID = ID;
        this.Naziv = Naziv;
        this.Autor = Autor;
        this.GodinaIzdanja = GodinaIzdanja;
    }

    abstract public Decimal izracunajIznosZaNaplatu(int brojDanaNajma, int starostOsobe);
}
```

Kod P1.4: Definicija interfejsa

Nikakve dalje promjene nisu potrebne što se tiče knjiga. U knjižari se može držati lista `iNplativo` objekata, međutim problem nastaje zbog činjenice da ih je potrebno porediti po ID

RAZVOJ PROGRAMSKIH RJEŠENJA

broju jer se kroz interfejs vidi samo metoda izračunavanja iznosa za naplatu. U toj situaciji postoje dvije varijante: ili vršiti konverziju između tipova (lošija) ili interfejs proširiti metodom:

Boolean uporedi (iNaplativo saKojimSePoredi)

Navedena metoda bi opet bila apstraktna u klasi Knjiga, dok bi specijalizacije knjige vršile override te metode i obavljali vlastito poređenje. Studentima se ostavlja da sami implementiraju navedeno rješenje.

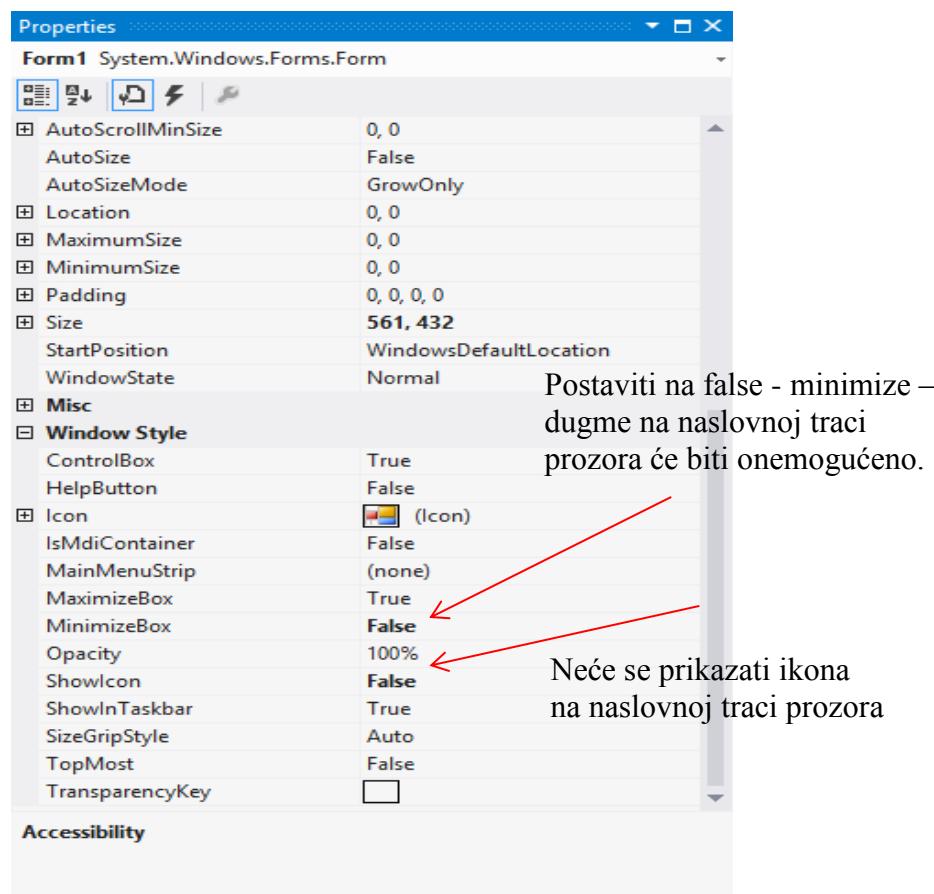
Prilog 2: Primjeri GUI kontrola i događaja

U okviru ovog priloga dati su jednostavnii ilustrativni primjeri za GUI kontrole i događaje.

Primjer 1:

Kreirati formu kao na slici. Potrebno je da se aktivira događaj prilikom podizanja forme koji će na naslovni (*title*) bar prozora napisati naziv dana kada se forma pokreće. Forma nema ikonu. Dugme minimizacije treba biti onemogućeno.

Opis rješenja: Nakon kreiranja Windows Forms aplikacije, na `Properties` prozoru treba izmjeniti osobine `MinimizeBox`, `ShowIcon`.



Kod P2.1 prikazuje kod koji odgovara ovom zadatku uključujući i metodu za upravljanjem događajem `Load` forme.

```
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;

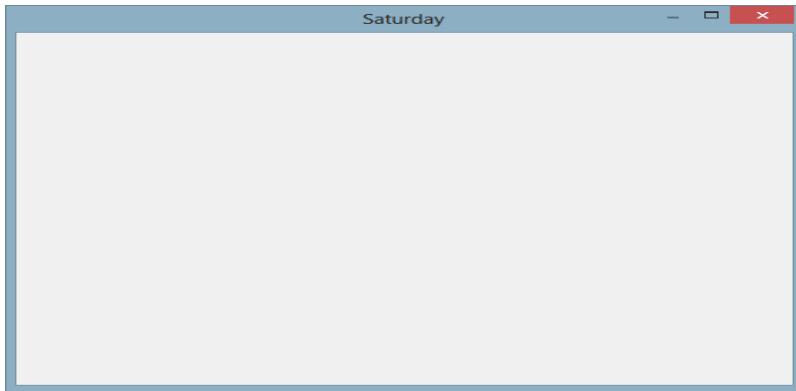
namespace DogadjajLoadForme
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            this.Text = DateTime.Now.DayOfWeek.ToString();
            this.Top = 60;
            this.Left = 60;
        }
    }
}
```

Load događaj-default događaj za formu.
U dizajneru se dobije duplim klikom na površinu forme.
Pokreće se pri prikazivanju forme.

P2.1: Metoda za upravljanje događajem Load piše datum na naslovnu liniju forme

Izvršavanjem ovog programskog rješenja dobija se forma bez ikone i sa onemogućenim dugmetom za minimiziranje forme.

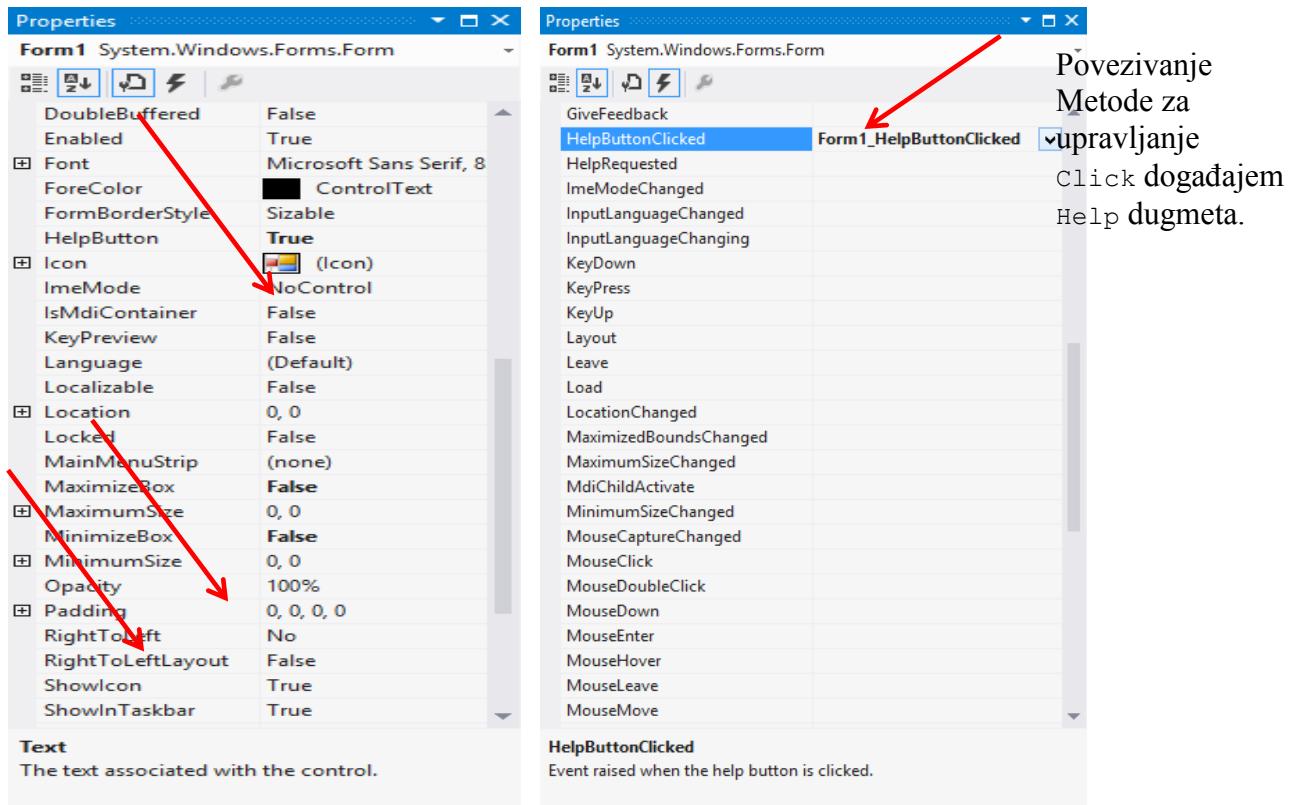


Primjer 2:

Kreirati formu koja ima dugme pomoći (*help button*), nema dugmad za minimiziranje i maksimiziranje prozora. Potrebno je aktivirati događaj na pritisak dugme pomoći. Datí samo obavijest da je aktiviran događaj.

Opis rješenja: Na *Properties* prozoru potrebno je postaviti odgovarajuće osobine na tražene vrijednosti i povezati događaj *Click* dugmeta pomoći sa metodom za upravljanje tog događaja.

RAZVOJ PROGRAMSKIH RJEŠENJA



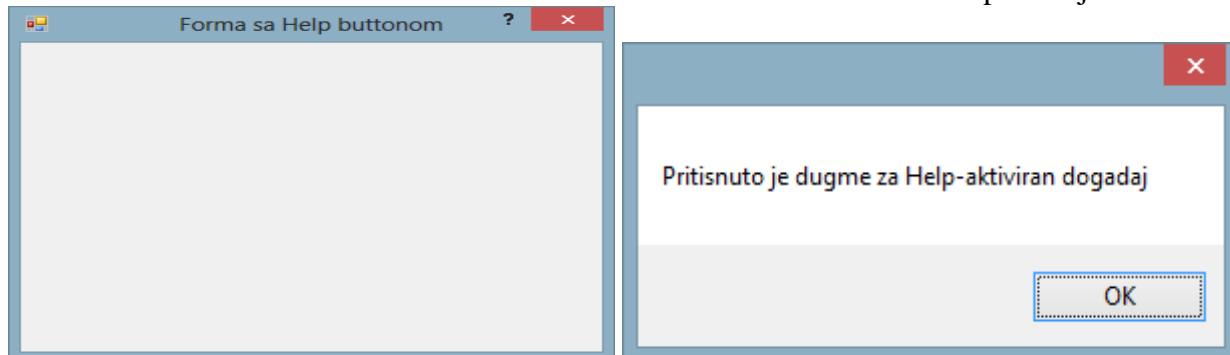
Metoda za upravljanjem događajem kada se klikne na Help ikonu prikazana je kodom P2.2.

```
private void Form1_HelpButtonClicked(object sender, CancelEventArgs e)
{
    MessageBox.Show("Pritisnuto je dugme za Help-aktiviran dogadaj");
    // dodati kod npr. kreirati drugu formu na kojoj je Help sadržaj
}
```

P2.2: Metoda za upravljanje događajem Click na Help ikonu

Prilikom izvršavanja dobija se početna forma:

Klikom na ikonu ? prikazuje se:



RAZVOJ PROGRAMSKIH RJEŠENJA

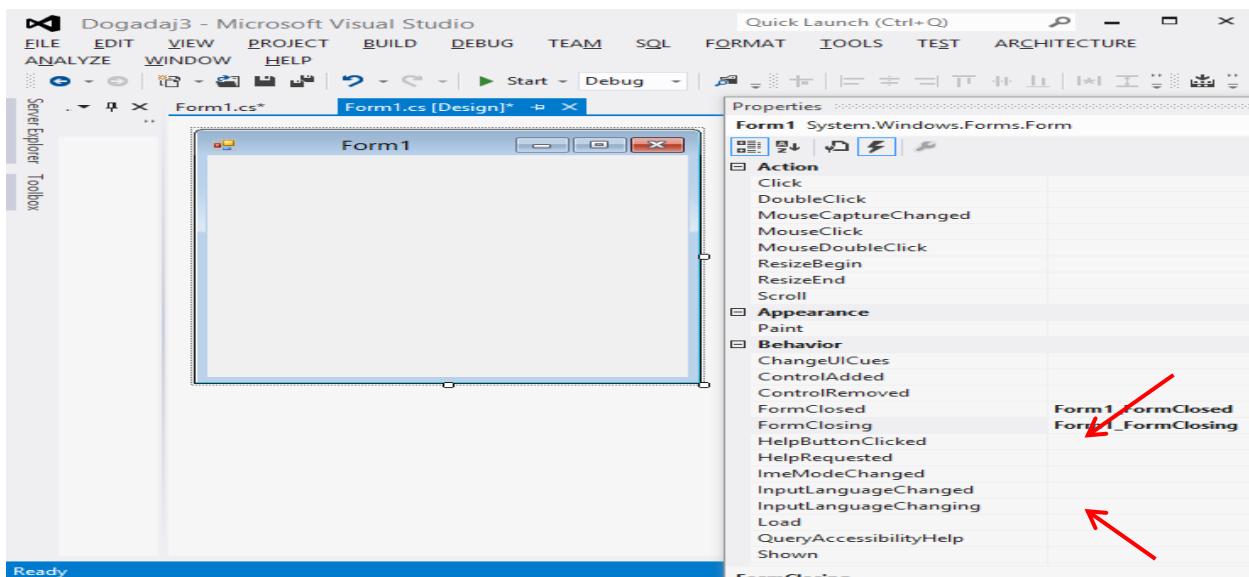
Primjer 3:

Kreirati formu proizvoljne veličine. Potrebno je programirati:

-događaj `Closing`, dešava se kada korisnik klikne na dugme za zatvaranje. Potrebno je onemogućiti zatvaranje forme ako su minute kada se vrši zatvaranje neparne. Poruku „Ne može se zatvoriti forma ako su minute neparne“ napisati na naslovnom baru.

-događaj `Closed`, dešava se nakon izvršenog događaja `Closing`. Nakon izvršavanja događaj `Closed` potrebno je napisati poruku „Forma je zatvorena“.

Opis rješenja: Nakon kreiranja Windows Forms aplikacije na `Properties` prozoru povezati događaje `Closed` i `Closing` sa metodama za njihovo upravljanje.



Metode za upravljanje događajima `Closing` i `Closed` prikazane su kodom P2.3.

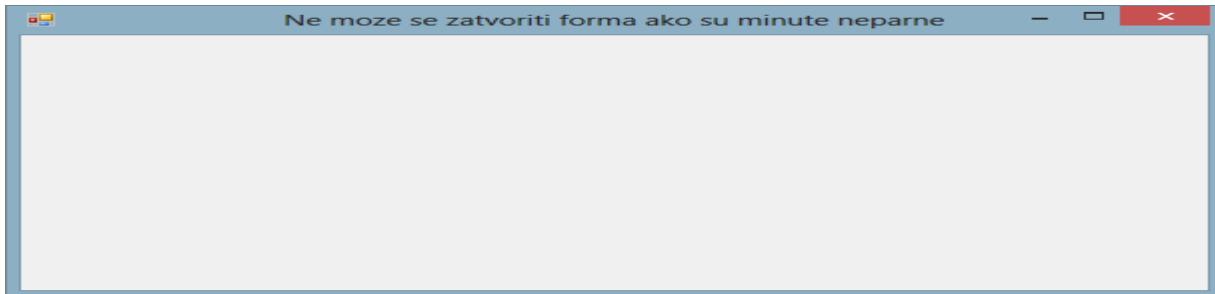
```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    if ((DateTime.Now.Minute % 2) == 1)
    {
        this.Text = "Ne može se zatvoriti forma ako su minute neparne";
        e.Cancel = true;
    }
}

private void Form1_FormClosed(object sender, FormClosedEventArgs e)
{
    MessageBox.Show("Forma je zatvorena");
}
```

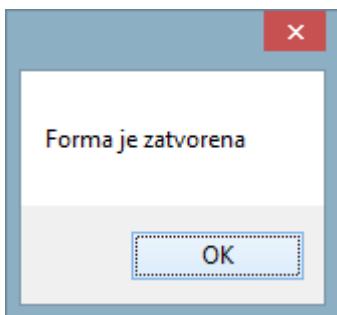
P2.3: Metode za upravljanje događajima `Closing` i `Closed`

RAZVOJ PROGRAMSKIH RJEŠENJA

Prilikom izvršavanja aplikacije i pokušaja da se forma zatvori u 9:03 dobija se forma izgleda:



Prilikom pokušaja da se forma zatvori u 9:04 forma se zatvara i prikazuje se poruka:



Primjer 4:

Kreirati programsko rješenje sa jednom formom. Potrebno je da se prilikom zatvaranja forme pitati korisnika da li je siguran da želi zatvoriti formu. Ukoliko je siguran izvršiti zatvaranje forme, a u ukoliko nije vratiti fokus na formu. Prilikom klika na formu aktivirati događaj koji će napisati poruku.

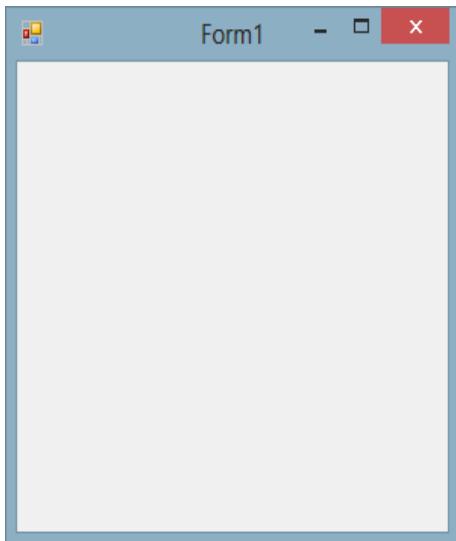
Opis rješenja: Prvo se kreira Windows Forms aplikacija. Nakon toga programiraju se odgovarajuće metode za upravljanje događajima `Closing` i `Click` kreirane forme. Kod ovih metoda dat je sa P2.4.

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    if (MessageBox.Show("Da li želite da zatvorite formu/aplikaciju?", "Zatvaranje
moje aplikacije",
    MessageBoxButtons.YesNo) == DialogResult.No)
    {
        // Prekida se closing event
        e.Cancel = true;
    }
}

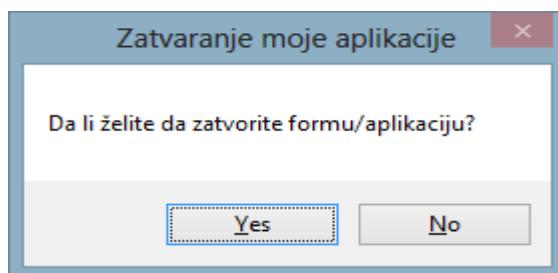
private void Form1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Desio se click event na formi");
    //ubaciti kod ...
}
```

P2.4: Metode za upravljanje događajima `Closing` i `Click`

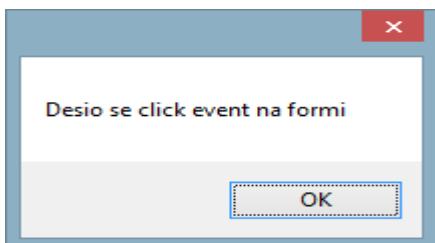
Izvršavanjem se pojavljuje forma:



Nakon klika na x-dugme za zatvaranje forme dobija se:



Ukoliko se desi klik na formu aktivira se događaj Click i ispisuje se poruka:

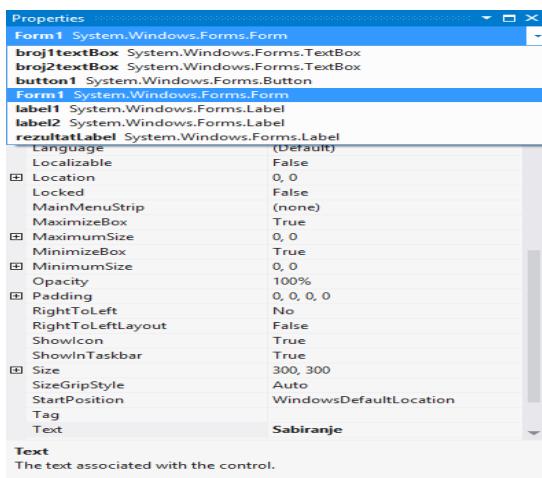


Primjer 5:

Napisati Windows aplikaciju koja će omogućiti unos dva polja, računanje i prikaz sume tih brojeva na labelu forme. Računanje i prikaz sume se dešava nakon klika na dugme Suma na formi.

Opis rješenja: Dizajnirati formu, kreirati sve kontrole i postaviti njihove osobine. Potrebne su kontrole:

RAZVOJ PROGRAMSKIH RJEŠENJA



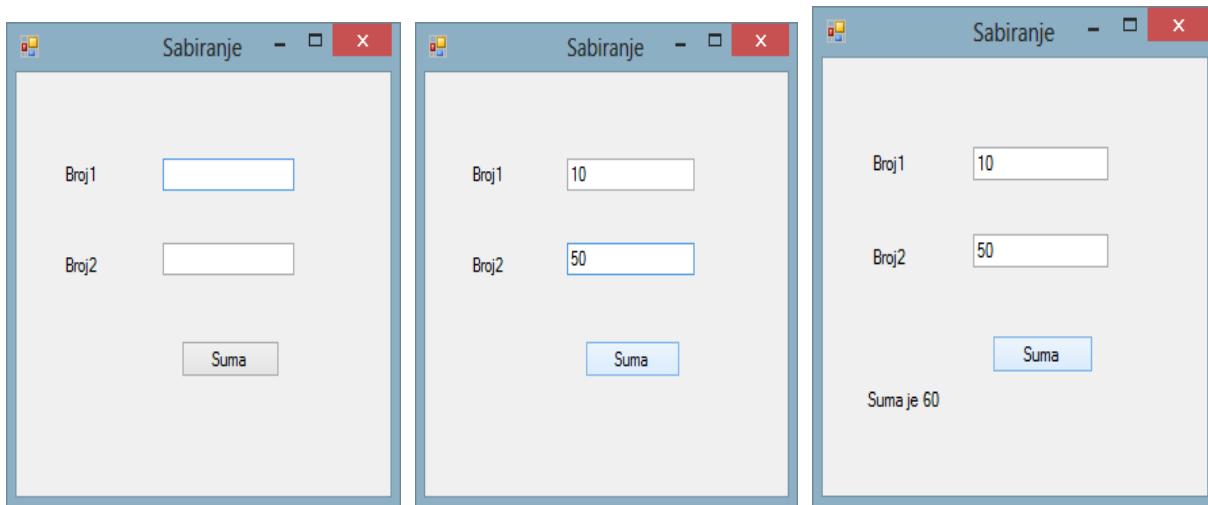
Metoda za upravljanje događajem `Click` za kontrolu dugmeta Suma data je sa P2.5.

```
private void button1_Click(object sender, EventArgs e)
{
    int broj1 = Int32.Parse(broj1textBox.Text);
    int broj2 = Int32.Parse(broj2textBox.Text);

    rezultatLabel.Text = "Suma je " + (broj1 + broj2);
}
```

Kod P2.5: Metoda za upravljanje događajem `Click` dugmeta Suma

Scenarij izvršavanja:



Početna forma,

unose se brojevi,

nakon klika na dugme ispisuje se rezultat

RAZVOJ PROGRAMSKIH RJEŠENJA

Primjer 6:

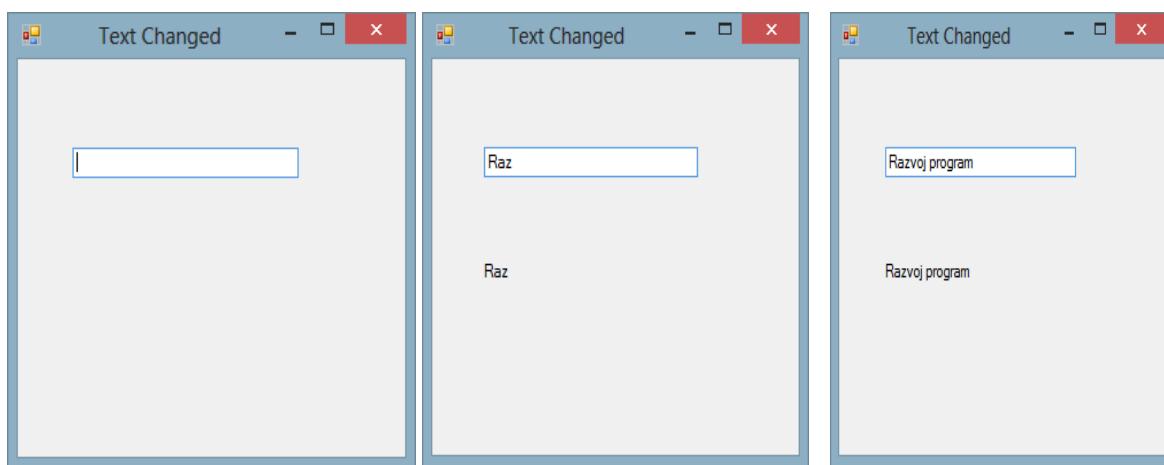
Kreirati formu na kojoj je tekst polje. Potrebno je da se prilikom svake promjene na tekstu polju (npr. korisnik ukuca slovo) promjena odrazi i na labeli koja se također nalazi na formi.

Opis rješenja: Kreirati potrebne kontrole: Form, textBox1, label1, postaviti odgovarajuće osobine. Događaj TextChanged-aktivira se nakon svakog pritiska na tipku koja uzrokuje promjenu na tekstu polju. Metoda za upravljanje događajem TextChanged data je sa kodom P2.6.

```
private void textBox1_TextChanged(object sender, EventArgs e)
{
    label1.Text = textBox1.Text;
}
```

Kod P1.6: Metoda za upravljanje događajem TextChanged-piše tekst na labelu

Scenarij izvršavanja:

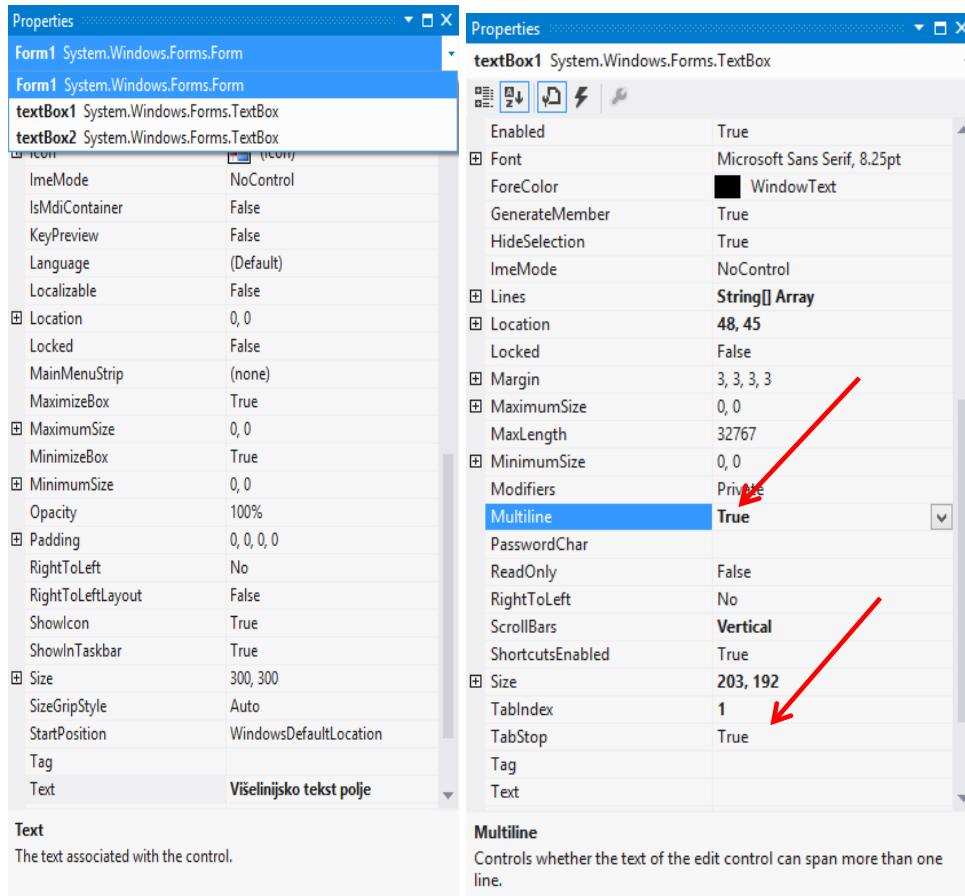


Primjer 7:

Kreirati formu sa dva tekstu polja. Jedno tekstu polje je višelinjsko i služi za unos teksta, drugo je jednolinijsko i u fokusu je prilikom pokretanja forme. Nakon ulaska na višelinjsko tekstu polje treba da se aktivira događaj koji će na jednolinijsko tekstu polje napisati poruku: UNESITE TEKST U VIŠELINIJSKO POLJE.

Opis rješenja: Kreirati kontrole i postaviti njihove osobine (prikazano slikama ispod).

RAZVOJ PROGRAMSKIH RJEŠENJA



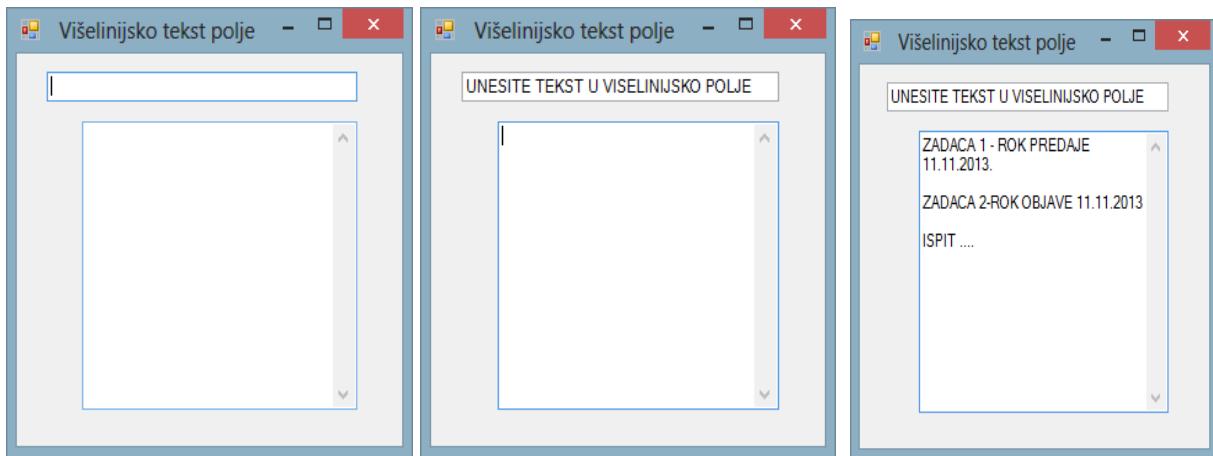
Metoda za upravljanje Enter događajem tekst polja data je kodom P2.7.

```
private void textBox1_Enter(object sender, EventArgs e)
{
    textBox2.Text = "UNESITE TEKST U VISELINIJSKO POLJE";
}
```

P2.7: Kod prilikom ulaska na višelinijsko polje-događaj Enter

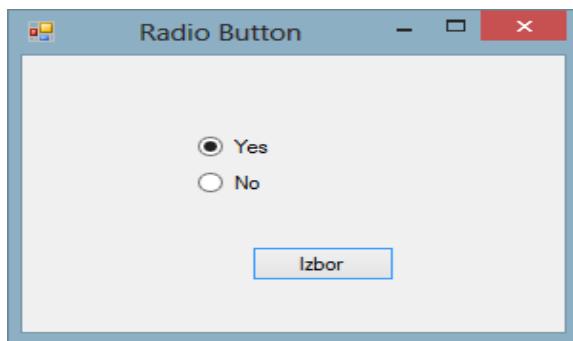
Scenarij izvršavanja:

RAZVOJ PROGRAMSKIH RJEŠENJA



Primjer 8:

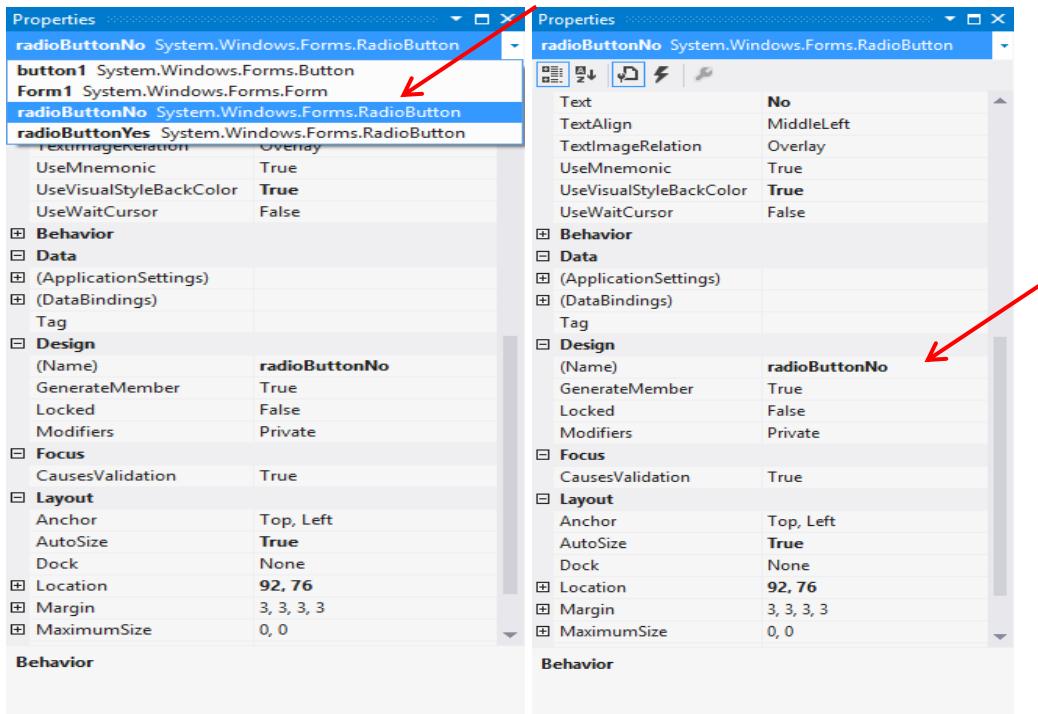
Potrebno je kreirati formu kao na slici.



Forma treba da sadrži radio dugmad (Yes, No). Poruka o tome koje je radio dugme označeno dobija se nakon klika na dugme Izbor.

Opis rješenja: Nakon kreiranja Windows Forms aplikacije kreiraju se i potrebne kontrole prikazane na slici ispod. Metoda za upravljanje događajem `Click` radio kontrole je data sa kodom P2.8.

RAZVOJ PROGRAMSKIH RJEŠENJA



```
private void button1_Click(object sender, EventArgs e)
{
    if (radioButtonYes.Checked)
        MessageBox.Show("Izabrali ste Yes");
    else
        MessageBox.Show("Izabrali ste No");
}
```

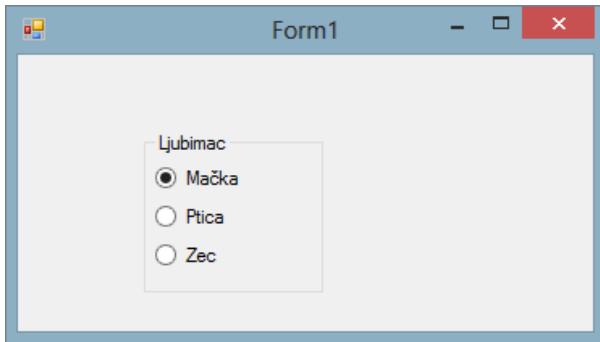
Kod P2.8: Metoda za upravljanje događajem Click dugme kontrole-piše koje je radio dugme izabrano

Primjer 9:

Kreirati grupu radio dugmadi koja nude izbor kućnog ljubimca (mačka, ptica, zec). Nakon izbora ljubimca (selekt odgovarajućeg radio dugmeta) treba da se napiše koji je ljubimac aktiviran na osnovu tekst labele uz radio dugme.

Opis rješenja: Kreira se forma i postave se kontrole. Nakon toga forma izgleda:

RAZVOJ PROGRAMSKIH RJEŠENJA



Događaj `Click` sva tri radio dugmeta se povezuje sa metodom za njegovo upravljanje (u ovom slučaju `radioButton1_Click`) čiji kod je dat sa P2.9.

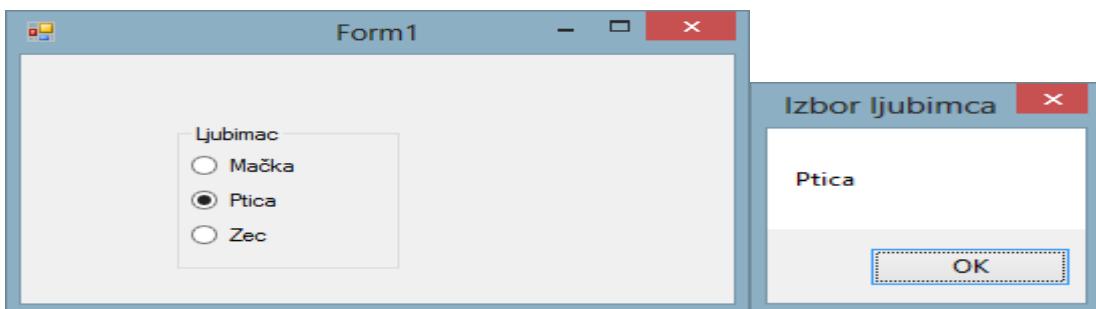
```
private void radioButton1_Click(object sender, EventArgs e)
{
    string result1 = null;

    foreach (RadioButton rb in groupBox1.Controls)
    {

        if (rb.Checked)
        {
            result1 = rb.Text;
            MessageBox.Show(result1, "Izbor ljubimca");
        }
    }
}
```

Kod P2.9: Provjera koje je dugme iz grupe radio dugmadi pritisnuto

Scenarij izvršavanja: Nakon izbora radio dugmeta sa labelom Ptica dobija se obavijest preko prozora poruke o izboru.



RAZVOJ PROGRAMSKIH RJEŠENJA

Primjer 10:

Kreirati formu koja će omogućiti kupovinu do 3 proizvoda (Sveska, Olovka, Gumica). Nakon izbora proizvoda za kupovinu ispisati poruku u okviru prozora poruke koji proizvodi su izabrani.

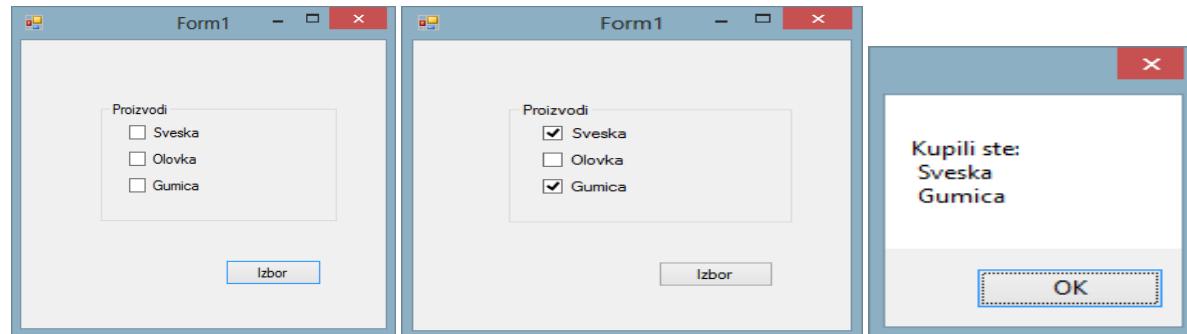
Opis rješenja: Kreirati formu i u okviru *group box* kontrole postaviti 3 *check boxa* za izbor proizvoda. Sa dugmetom Izbor se potvrđuje kupovina. Metoda za `Click` dogadaja dugmeta Izbor ispisuje podatke o proizvodima. Kod metode je dat sa P2.10.

```
private void button1_Click(object sender, EventArgs e)
{
    string proizvod = String.Empty;

    if (checkBox1.Checked)
        proizvod += "\n Sveska";
    if (checkBox2.Checked)
        proizvod += "\n Olovka";
    if (checkBox3.Checked)
        proizvod += "\n Gumica";
    MessageBox.Show("Kupili ste: " + proizvod);
}
```

Kod P2.10. Na osnovu markiranih *check boxova* ispisuju se odabrani proizvodi

Scenarij izvršavanja:



Početna forma, odabrani proizvodi, nakon klika na dugme izbor ispisuju se odabrani proizvodi

Primjer 11:

Napisati program koji će kreirati formu preko koje će se moći unositi dva broja. Unesena dva broja preko prve forme treba proslijediti drugoj formi, nakon klika na dugme Izračunaj. Na drugoj formi prikazati sumu ta dva broja.

Opis rješenja: Rješenje je urađeno bez dizajnera. Korištene su odgovarajuće klase za grafičke kontrole. Izvršena je i dekompozicija projekta. Kod P2.11.1 predstavlja glavni program projekta, P2.11.2 predstavlja početnu formu, a P2.11.3 je kod za drugu formu, na kojoj se piše rezultat.

RAZVOJ PROGRAMSKIH RJEŠENJA

```
using System;
using System.Windows.Forms;
using System.Drawing;
using namespace System;

using MojProjekat;

static Void Main() {
    PocetnaForma p = new PocetnaForma();
    Application.Run(p); // moglo je umjesto (p) pisati (new PocetnaForma)
}
```

Kod P2.11.1 Glavni program projekta

Slijedi kod za početnu formu. Naglašeni su programski iskazi koji se odnose na poziv druge forme i proslijedivanje parametara.

```
using System;
using System.Windows.Forms;
using System.Drawing;
using MojProjekat;

class PocetnaForma {

    Button pozoviFormu = new Button();
    TextBox broj1 = new TextBox();
    TextBox broj2 = new TextBox();

    public PocetnaForma()
    {
        InicijalizirajFormu();
    }

    void InicijalizirajFormu()
    {
        this.ClientSize = Drawing::Size(120, 95);

        pozoviFormu.Text="Izracunaj";
        pozoviFormu.Location = new Point(37, 66);
        pozoviFormu.Size = new Size(75, 23);
        pozoviFormu.Click +=new EventHandler(pozoviFormu_click);
        this.Controls.Add(pozoviFormu);

        broj1.Location = new Point(12, 14);
        broj1.Size = new Size(100, 20);
        this.Controls.Add(broj1);

        broj2.Location = new Point(12, 40);
        broj2.Size = new Size(100, 20);
        this.Controls.Add(broj2);
    }
}
```

RAZVOJ PROGRAMSKIH RJEŠENJA

```
void pozoviFormu_Click( Object o, EventArgs e)
{
    float b1 = Convert.ToSingle(broj1.Text);
    float b2 = Convert.ToSingle(broj2.Text);

    // proslijeđivanje parametara konstruktoru
    RezultatForma r = new RezultatForma(b1,b2);
    r.Show(); // ShowDialog()
}
```

Kod P2.11.2: Početna forma za unos dva broja koja se proslijeđuju drugoj formi

I na kraju kod P2.11.3 za drugu, rezultantnu formu.

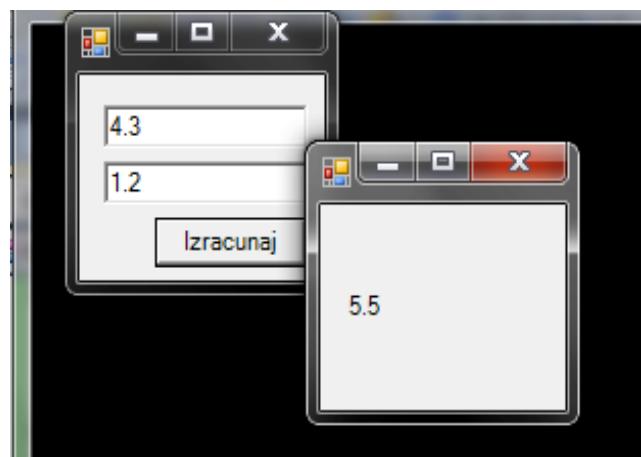
```
using System;
using System.Windows.Forms;
using System.Drawing;

namespace MojProjekat {
    class RezultatForma : Form
    {
        public RezultatForma(float b1, float b2) {
            this.ClientSize = new Size(120, 95);

            Label ispis = new Label();
            ispis.Location = new Point(12, 40);
            ispis.Size = new Size(100, 20);
            ispis.Text = (b1+b2).ToString();
            this.Controls.Add(ispis);
        }
    }
}
```

Kod P2.11.3: Rezultantna forma-forma koju je pozvala druga forma

Scenarij izvršavanja:



POPIS SLIKA

Slika 1.1: Model vodopada

Slika 3.1: Kreiranje projekta RacunDekompozicija

Slika 3.2: a)Kreiranje biblioteke -`dll` b)Kreirani `dll`

Slika 5.1: Ekran iz 1970 godine

Slika 5.2: Ekran u 1980 godini

Slika 5.3: Ekran u 1990 godini

Slika 5.4: Ekran 2000 godine

Slika 5.5: Dijeljeni prozori

Slika 5.6: Preklapajući prozori

Slika 5.7: Kaskadni prozori

Slika 5.8a: Kreiranje; 5. 8b: Početni prozor forme u dizajn modu projekta

Slika 5.9: Prozor za postavljanje osobina

Slika 5.10: Toolbox: Komponente i kontrole za Windows Forms-Visual Studio

Slika 5.11: Dugme na formi i editor za postavljanje osobina

Slika 6.1: Forma sa kontrolama u dizajn modu

Slika 8.1: Primjer menija sa naznačenim elementima

Slika 8.2: Kreiranje menija sa dizajnerom

Slika 8.3: SDI i MDI forma

Slika 9.1: Prikaz principa udaljenost, sličnost i simetrija

Slika 9.2. Pretraga filmova u aplikaciji Videoteka

Slika 9.3: Korištenje ubrzanja sa tastature u aplikaciji Videoteka

Slika 9.4: Dodavanje novog filma u aplikaciji Videoteka

Slika 9.5: Tekst labele uz kontrolu u aplikaciji Videoteka

Slika 9.6: Korištenje dugih natpisa u aplikaciji Videoteka

Slika 9.7: Korištenje crvene boje za indikaciju neispravnog statusa u aplikaciji Videoteka

Slika 9.8: Korištenje žute boje za indikaciju upozorenja u aplikaciji Videoteka

Slika 9.9: Vodič kroz sistem u aplikaciji Videoteka

Slika 12.1: Hijerarhijska struktura XML dokumenta

POPIS TABELA

Tabela 5.1: Hronološki pregled razvoja grafičkog korisničkog interfejsa

Tabela 5.2. Form osobine (*properties*), metode (*methods*) i događaji (*events*).

Tabela 5.3: Često korištene osobine i događaji dugme kontrole

Tabela 6.1: Dio osobina kontrole i klase labela

Tabela 6.2: Osobine i događaji `TextBox` kontrole

Tabela 6.3: Osobine i događaji `CheckBox` kontrole

Tabela 6.4: Osobine i događaji `RadioButton` kontrole

Tabela 6.5: `GroupBox` osobine

Tabela 6.6: `Panel` osobine

Tabela 6.7: Događaji tastature i njihovi argumenti

Tabela 6.8: Događaji mišem i njihovi argumenti

Tabela 7.1: Često korištene osobine i događaji `PictureBox` kontrole

Tabela 7.2: `ListBox` osobine, metode i događaji

Tabela 7.3: `ComboBox` osobine i događaji

Tabela 7.4: `NumericUpDown` osobine i događaji

Tabela 7.5: `MonthCalendar` osobine i događaji

Tabela 7.6: `LinkLabel` osobine i događaji

Tabela 7.7: `TabControl` osobine i događaji

Tabela 8.1: `MenuStrip` i `ToolStripMenuItem` osobine i događaji

Tabela 9.1. Korištenje boja za indikaciju statusa

Tabela 9.2: Vrste kontekstualne pomoći

Tabela 10.1: Osobine `ErrorProvider` kontrole

Tabela 10.2: Odabrana lista standardnih izuzetaka

Tabela 10.3: Korisne osobine `Exception` klase

Tabela 11.1: Klase za rad sa datotekama i direktorijima u `System.IO`

Tabela 11.2: Izuzeci pri radu sa datotekama i direktorijima

Tabela 11.3: Metode klase `File` za kreiranje, otvaranje, brisanje, pomjeranje, zamjenu

Tabela 11.4: Metode klase `File` za čitanje i pisanje

Tabela 11.5: Osnovne metode za rad sa direktorijima

Tabela 11.6: Metode za čitanje iz i pisanje u objekte tipa `FileStream`

Tabela 11.7: Značenje `Close`, `Dispose`, `Flush` metoda

Tabela 11.8: Metode `StreamWriter` klase

Tabela 11.9: Metode `StreamReader` klase

RAZVOJ PROGRAMSKIH RJEŠENJA

Tabela 12.1: Osobine za formatiranje i metode klase `XmlTextWriter` za pisanje u XML dokumente

Tabela 12.2: XML DOM klase

Indeks pojmoveva

A	B
apstraktna klase, 68	base, 66
apstraktni tipovi podataka, 68	biblioteka, 55
ActiveMDIChild osobina, 159	binarni operatori, 27
AND (&) bitni operator, 27	BlickStyle osobina, 182
AND (&&) logički operator, 27	BlickRate osobina, 182
anonimna metoda, 77	Boolean izraz, 27
aritmetički operatori, 27	Boolean operatori, 27
Array klasa, 31	boxing, 28
as operator, 68	break iskaz, 33
asinhroni mehanizmi, 259	built-in konverzija, 24
async/await, 262	Button klasa, 102
	byte, 22
C	D
C prog. jezik, 17	DataGridView kontrola, 235
C++, 17	DateTimePicker kontrola, 180
C#, 17	datoteka, 202
catch klauzula, 190	debug, 18
cast operator, 24	decimal, 23
char, 22	deklaracija varijabli, 23
CheckBox kontrola, 115	dekrement, 27
checked ključna riječ, 29	dekompozicija, 41
CheckedListBox kontrola, 130	delegate ključna riječ, 21
cjelobrojni tip podataka, 22	delegati, 75

RAZVOJ PROGRAMSKIH RJEŠENJA

click dogadjaj, 103	destruktor, 41
close metoda, 100	deserializacija, 220
CLR (Common Language runtime), 18	Directory klasa, 203
Console klasa, 24	dll (Dynamic Link Library), 55
const polje, 32	dijalog boxovi, 95
ComboBox kontrola, 128	dogadjaj (<i>event</i>), 106
ComboBox klasa, 128	do iskaz, 34
	DOM – Document Object Model, 237
	dugme (<i>button</i>) kontrola, 102
	dynamic, 28
E	F
enkapsulacija, 14	finally blok, 190
enum ključna riječ, 25	File klasa, 205
ErrorProvider kontrola, 182	FileStream klasa, 210
event, 106	for iskaz, 35
exception, 178	foreach, 35
Exception klasa, 182	forma, 96
eksplicitna konverzija, 24	Form klasa, 100
	funkcijska paradigma, 15
G	H
generički koncepti, 80	heap, 27
Gestalt principi, 162	hijerarhija, 63
get, 47	HTML, 17
GroupBox kontrola, 119	
GUI interfejs, 146	

RAZVOJ PROGRAMSKIH RJEŠENJA

I	K
ikona, 169	klasa, 41
identifikator, 21	ključna riječ, 21
if iskaz, 32	kolekcije podataka, 81
implicitna konverzija, 24	konstanta, 32
inkrement, 27	konstruktor, 42
inicijalizacija, 22	kontekstualni meniji, 152
instanca, 41	kontrola-GUI, 111
interface, 69	
interfejs-grafički , 87	
čint tip podataka, 22	
Invoke metod, 266	
is operator,68	
IsMDI osobina, 156	
iteracijski iskazi, 34	
iterativna paradigma, 15	
L	M
labela, 111	main metoda, 20
Label klasa, 111	MDI interfejs, 155
lambda, 78	meni, 146
lista, 81	MenuStrip kontrola, 149
ListBox kontrola, 130	metoda, 37
logički operatori, 27	MessageBox klasa, 120
logička paradigma, 15	MessageBox.Show klasa, 120
LinkLabel kontrola, 139	Microsoft Visual Studio .NET, 20
	MonthCalendar kontrola, 138
	multidimenzionalni nizovi, 31

RAZVOJ PROGRAMSKIH RJEŠENJA

N	O
<i>namespace</i> , 20	objekat, 41
nasljeđivanje, 63	<i>object</i> , 28
.NET Framework, 18	OR (I) bitni operator, 27
<i>new</i> , 23	OR (II) logički operator, 27
niz, 30	<i>out</i> parametar, 45
NumericUpDown kontrola, 137	<i>override</i> , 66
	<i>OpenFileDialog</i> , 154
	operatori, 27
P	R
panel, 119	radio dugme, 118
parametri, 37	RadioButton kontrola, 118
<i>Partial</i> , 59	<i>Readonly</i> , 50
PictureBox kontrola, 128	<i>ref</i> parametar, 45
pointer, 30	referencni tipovi, 27
polimorfizam, 65	<i>return</i> iskaz, 36
<i>pop-up</i> meni, 152	
primitivni tipovi podataka, 22	
programiranje vođeno događajima, 87	
programsко rješenje, 7	
<i>properties</i> , 47	
prozor (<i>window</i>), 92	
<i>private</i> specifikator, 42	
<i>protected</i> specifikator, 42	
<i>public</i> specifikator, 42	

RAZVOJ PROGRAMSKIH RJEŠENJA

S	T
SaveFileDialog kontrola, 154	Tab kontrola, 141
SDI (Single Document Interface),155	tag, 226
salead klasa, 41	testiranje,10
Sbyte, 22	testiranje upotrebljivosti sistema, 176
set,47	tekst polje,112
serijalizacija datoteka,220	Text osobina, 96
Serijalizacija XML, 224	TextBox kontrola, 112
short, 22	tip podataka, 22
static, 50	throw iskaz, 193
statusna linija, 185	Toolbox, 97
StatusStrip kontrola, 185	ToString metoda, 57
string, 28	ToolStripMenuItem osobine,149
struct, 25	TreeView kontrola, 239
struktura, 25	try blok, 190
StreamReader klasa, 216	try/catch/finally iskaz, 190
StreamWriter klasa, 214	TryParse metoda, 179
System namespace, 23	
System.Collection, 81	
System.Drawing, 96	
System.IO, 203	
System.Threading, 248	
System.Threading.Tasks, 261	
System.Windows.Forms, 96	
System.XML,231	

RAZVOJ PROGRAMSKIH RJEŠENJA

U	V
unarni operator, 27	validacija, 178
<i>unboxing</i> , 28	Validating događaj, 180
<i>unchecked</i> , 29	Validate događaj, 180
<i>using</i> izraz, 56	<i>var</i> , 29
	variјable, 21
	<i>virtual</i> ključna riječ, 65
	vrijednosni tipovi, 22
W	X
while izraz, 34	XML, 226
Windows Form aplikacija, 20	XmlElement klasa, 237
window, 92	XmlTextReader klasa, 234
	XmlTextWriter klasa, 232
	XmlValidating klasa, 231