Увод 3
Објектно оријентисано програмирање 4
1.1 Увод у ООП 4
1.2. Процедурално наспрам објектно оријентисаног програмирања 5
1.3. Објекти 6
1.4. Класе 7
1.4.1. Атрибути и методи 8
1.4.2. Класни дијаграми 8
1.5. Енкапсулација 9
1.5.1. Интерфејси 9
1.5.2. Реализација 10
1.6. Конструктори 10
1.7. Наслеђивање 11
1.7.1. Надређене и подређене класе 12
1.7.2. Релација Јесте 12
1.8. Полиморфизам 13
II. .NET Framework 14
2.1. О .NET-у 14
2.2. Историјат настанка .NET технологије 15
2.3. Основне карактеристике .NET развојног окружења 15
2.3.1. Common Language Ryntime (CLR 16
2.3.2. Microsoft Intermediate Language (MSIL) код 17
2.3.3. Just In Time Compilers (JIT-ers) 18
2.3.4. Common Language Specifications (CLS) 19
2.3.5. Common Type System (CTS) 19
2.3.6. Garbage Collection (GC) 20
2.3.7. Common Language Infrastructure (CLI) 20
2.3.8. Framework Class Library (FCL) 21
2.4. Приступ подацима у .NET-у 23
2.4.1. ODBC 23
2.4.2. DAO 24
2.4.3. RDO 24
2.4.4. OLE DB 24
2.4.5. ADO 25
2.5. Склопови 26
2.6. .NET програмски језици 26
2.7. Предности и мане .NET развојног окружења 28
III. Програмски језик C# 30
3.1. Основе C# језика 30
3.2. Декларисање и додељивање променљивих 32
3.2.1. Именовање променљивих 32
3.2.2. Декларисање променљивих 32
3.2.3. Рад са примитивним типовима података 33
3.3. Оператори у C# језику 34
3.4. Једноставна селекција помоћу If...Else 35
3.5. Рад са исказима Switch...Case 35
3.5.1. Поштовање правила исказа Switch 36
3.6. Коришћење исказа Do и While 37

3.7. Рад са For исказима 37
3.7.1. Исказ Foreach 38
3.8. IntelliSense и HotCompiler 38
3.9. Рад са контролама 39
3.9.1. Контрола ListBox 39
3.9.2. Контрола ComboBox 41
3.9.3. Контрола TreeView 41
3.9.4. Контрола ImageList 42
3.9.5. Контрола ListView 42
3.9.6. Контрола MenuStrip 43
3.9.7. Контрола ToolStrip 44
3.9.8. Стандардни оквири за дијалог 44
IV. ADO.NET 46
4.1. Увод у ADO.NET 46
4.2. Преглед грађе ADO.NET -а 47
4.3. .NET добављачи података (Data Provider) 47
4.4. Компоненте добављача података 48
4.4.1. Класе Connection 49
4.4.2. Класе Command 52
4.4.3. Објекат DataReader 56
4.4.4. Objekat DataAdapter 58
4.5. Објекат DataSet 63
4.5.1. Својства објекта DataSet 65
4.6. Објекат DataTable 65
4.6.1. Својства објекта DataTable 67
4.6.2. Колекција Columns 68
4.6.3. Колекција Rows 70
4.7. Објекат DataView 71
4.7.1. Својства објекта DataView 72
4.7.2. Својство RowStateFilter 73
4.7.3. Методи објекта DataView 74
V. Водич за израду практичних примера 75
5.1. SQL SELECT упит 75
5.2. Data Binding 79
5.3. DataReader класа 82
Закључак 85

Увод

Као што сам наслов најављује, овај рад бави се приступом базама података у .NET развојном окружењу. Међутим, да би се то на веродостојан начин објаснило неопходно је било направити увод тј. описати главне карактеристике ”појмова” као што су објектно оријентисано програмирање, . NET окружење, програмски језик C# и ADO. NET компонента.

Сврха дипломског рада ја да, кроз теорију и практичне примере, прикаже које су све могућности и на које све начине можемо успоставити везе са базама података преко ADO. NET класа, коришћењем C# програмског језика и напредних објектно оријентисаних могућности.

Први део је посвећен објектно оријентисаном програмирању, тј. разумевању у потпуности концепта објектно оријентисаног програмирања. Кроз ово поглавље је описана главна разлика између процедуралног и објектно оријентисаног програмирања, објашњени су њени елементи као што су објекти, класе, енкапсулација, наслеђивање и други.

У другом делу дипломског рада, дат је опис рада .NET Framework-а. Дате су основне карактеристике развојног окружења .NET као и начини приступа подацима. Изложени су најчешће коришћени програмски језици у оквиру окружења, као и предности и мане са којима се сусреће.

Трећи део рада обухвата програмски језик C#. У овом одељку видећемо како се декларишу променљиве, употребљавају оператори, користе условни искази иф и искази за итерацију, као што је њхиле. Набројане су и најчешће коришћене контроле и њихове керактеристике.

Четврти део представља уједно и најбитнији део, јер подробно објашњава ADO.NET технологију приступа подацима. Како се ова компонента састоји од више објеката, отуда и разлог за детаљан опис сваког од њих, али и прилика да се изнесу њихове класе, методе и својства.

У изради рада коришћени су графици, табеле и прикази који су помогли да на што илустрованији начин прикаже ова материја.

Објектно оријентисано програмирање

1.1 Увод у ООП

Објектно оријентисано програмирање (ООП) је начин приступа реализацији софтвера као моделу реалног света. Објектно оријентисано програмирање представља сасвим нов начин размишљања у односу на тзв. традиционално алгоритамско програмирање. Проблеми се идентификују као објекти који нешто раде и апстракције које представљају скупове објеката истих својстава. Код оваквог начина програмирања много више времена се троши на пројектовање, а мање на само програмирање, што је био случај код обичног процедуралног (традиционалног) програмирања. ООП у центар рада поставља проблем који решава, а не питање конкретног програмског решења. Акценат је на објектима (деловима система) који нешто раде, а не на алгоритмима тј. на томе како нешто ради. ООП пружа неколико битних концепата, а то су: класе, објекти, енкапсулација, методе и својства, конструктори и деструктори, наслеђивање и полиморфизам.

Решавање проблема парадигмом објектно-оријентисаног програмирања, је врло слично људском начину размишљања и решавању проблема. Састоји се од идентификовања објеката и постављање објеката који ће користити одговарајућу секвенцу за решење одређеног проблема. Ради се о дизајну објеката чија ће понашања као јединица и у њиховој међусобној интеракцији, решити одређени проблем. Интеракција између објеката се састоји у размени порука, где одређена порука усмерена према одређеном објекту, покреће енкапсулиране операције у том објекту, чиме се решава део обично ширег и сложенијег проблема. Уопштено гледано, објектно-оријентисано решавање проблема се састоји из четири корака:

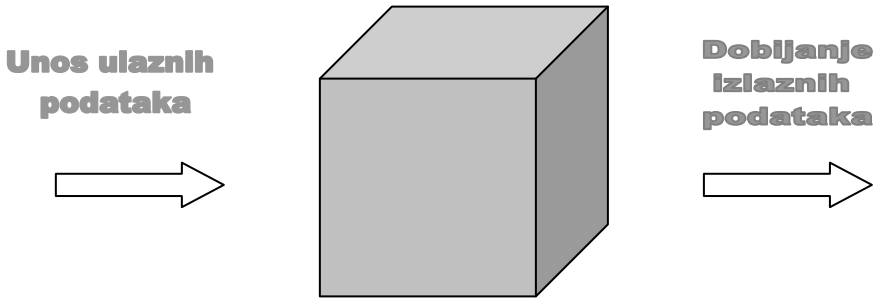
1. идентификовање проблема
2. идентификовање објеката који су потребни за његово решење
3. идентификовање порука које ће објекти међусобно слати и примати
4. креирање секвенце порука објектима, које ће решавати проблем

1.2. Процедурално наспрам објектно оријентисаног програмирања

Пре него што дубље уронимо у предности ОО програмирања, размотримо важније питање: Шта је, у ствари, објекат? Ово је истовремено и сложено и једноставно питање. Сложено је, јер

прелазак на потпуно нов начин размишљања није нимало једноставан. Једноставно је, јер већина људи већ сада „размишља помоћу објеката".

На пример, када посматрамо неку особу, особу видимо као целину, као објекат. Та особа има својства, као што су боја очију, понашање и својствен начин хода. У својој основној дефиницији, објекат је целина која садржи и податке и понашање. Ово је кључна разлика између методологије традиционалног програмирања, односно процедуралног програмирања, и ОО програмирања. У процедуралном програмирању, код се пише унутар функција или процедура. У идеалном случају, те процедуре постају„црне кутије" (Слика 1), што подразумева уношење улазних података и добијање излазних података. Подаци се смештају у засебне структуре, а обрађују се унутар функција или процеса.



Слика 1: Црна кутија[3]

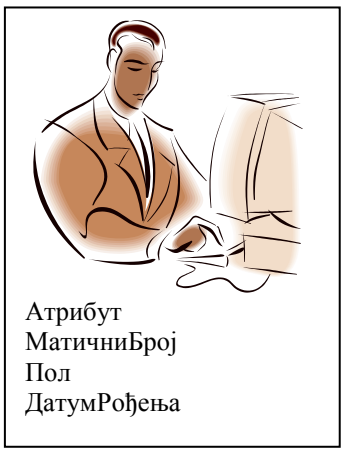
Процедурално програмирање представља камен темељац још из бронзаног доба рачунарства, па зашто бисмо га онда мењали? Најпре, у процедуралном језику се подаци одвајају од процедура, а често су подаци глобални, па је лако променити податке који су ван нашег домета. То значи да је приступ подацима неконтролисан и непредвидив. Друго, пошто немате контролу над тиме ко све може приступити подацима, тестирање и отклањање грешака је отежано. У објектима су ови проблеми решени обједињавањем података и понашања у једном лепом, потпуном пакету. Они нису само примитивни типови података, попут целобројног типа и низа знакова. Објекти садрже целине, као што су цели бројеви и низови знакова, али такође садрже

методе (у ОО методологији, функције се зову *методи*) са којима можете радити над типовима података. Такође, можете контролисати приступ члановима објеката. То значи да неки чланови, попут типова података и метода, могу бити скривени од осталих објеката.

Због чега је ОО различито од процедуралног програмирања? Процедурално програмирање одваја податке програма од операција које баратају подацима. На пример, ако желимо послати податке преко мреже, шаљу се само релевантни подаци, а очекује се да програм на другом крају мреже зна шта и како да ради са тим подацима. Основна предност ОО програмирања је то што се подаци и операције које баратају подацима налазе у истом објекту. На пример, када се објекат преноси мрежом, на одредиште стиже цео објекат, односно подаци и понашање.

1.3. Објекти

Ако узмемо да је најједноставнија дефиниција објекта да је објекат скуп података и операција на тим подацима, лако је повући паралелу са светом око нас и видети да све може бити идентификовано као објекат: (скоро) све може бити дефинисано као скуп материје (подаци) и ствари које та материја може да ради (операције). Објект репрезентује концепт из реалног света и као такав представља главницу ОО програмирања.

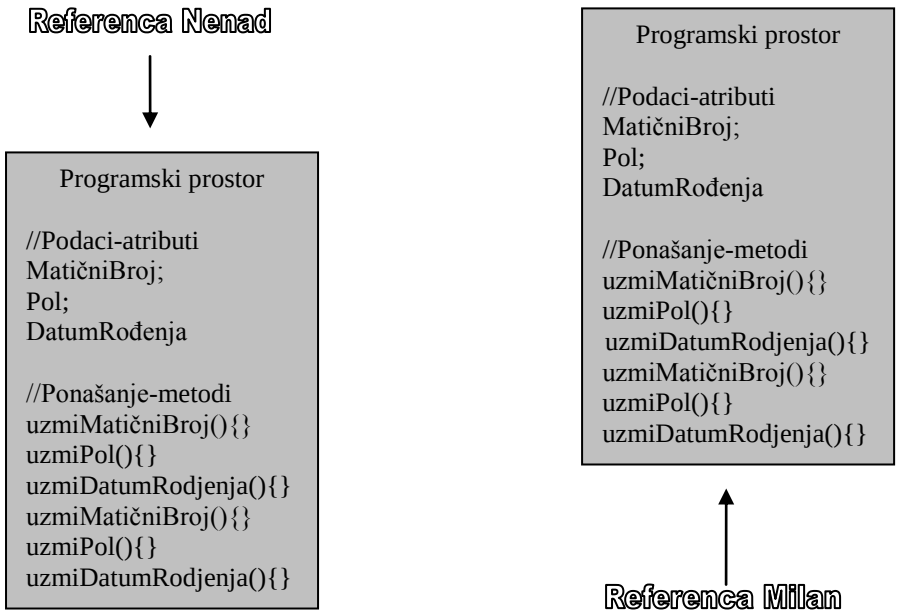


Подаци смештени у објекту представљају стање објекта. У терминологији ОО програмирања, ти подаци се зову *атрибути*. У нашем примеру (Слика 2), атрибути, односно особине службеника, могли би бити матични број, датум рођења, пол, број телефона, и слично. Ови атрибути садрже информације које разликују појединачне објекте.

Слика 2: Атрибути службеника[3]

Понашање објекта је оно што објекат може да ради. У процедуралним језицима, понашање се дефинише процедурама, функцијама и подпрограмима. У терминологији ОО програмирања, ова понашања се садрже у *методима*, а метод позивамо тако што објекту пошаљемо поруку. У нашем примеру `службеника`, сматрамо да је једно од потребних понашања објекта `Службеник`, задавање и враћање вредности разних атрибута. Стога, сваки атрибут ће имати одговарајуће методе, као што су `задајПол()`

и `узмиПол()`. У овом случају, када неком другом објекту требају ове информације, он може послати поруку објекту службеника и питати га којег је пола. Када се направи објекат, кажемо да је направљен примерак објекта, односно да је објекат формиран. Стога, ако направимо три службеника, ми заправо правимо три примерка класе `Службеник`. Сваки објекат садржи сопствену копију атрибута и метода (Слика 3). Објекат службеник Ненад (Ненад је његов идентитет), има сопствене копије свих атрибута и метода који су дефинисани у класи `Службеник`. Објекат службеник, који се зове Милан, има сопствену копију атрибута и метода. Заједничко за оба наведена објекта је да имају засебне копије атрибута `ДатумРођења` и метода `узмиДатумРођења`.



Слика 3: Програмски простори[3]

Што се тиче објекта, морамо бити свесни да не постоји физичка копија метода за сваки објекат. У ствари, сваки објекат показује на један исти физички код. Међутим, то је питање реализације, које се решава на нивоу платформе коју чине програмски преводилац и оперативни систем. Са концепцијског гледишта, можемо замислити да је објекат потпуно независтан и да има сопствене атрибуте и методе.

1.4. Класе

Класа је основна организациона јединица програма у објектно оријентисаном језику. Класа представља тип, скуп објеката истих својстава. Када правимо примерак објекта, класу употребљавамо као основу, која дефинише начин израде објекта.

Да бисмо објаснили класе и методе, вреди употребити пример из области релационих база података. У табели базе података, дефиниција саме табеле (име поља, опис и употребљени тип података) може се упоредити са класом (метаподаци), а објекти би се могли довести у везу са редовима табеле (подаци).

Не смемо заборавити да сваки објекат има сопствене атрибуте (могу се упоредити са пољима) и понашања (слично функцијама и подпрограмима). Класа дефинише атрибуте и понашања које поседују сви објекти направљени помоћу те класе. Класе су засебне компоненте и могу се појављивати самостално, или у оквиру библиотеке. Пошто се објекти праве помоћу класа, закључујемо да класе морају дефинисати темеље објеката (податке, понашање и поруке). Укратко, класу морамо испројектовати пре него што кренемо у стварање објекта.

1.4.1. Атрибути и методи

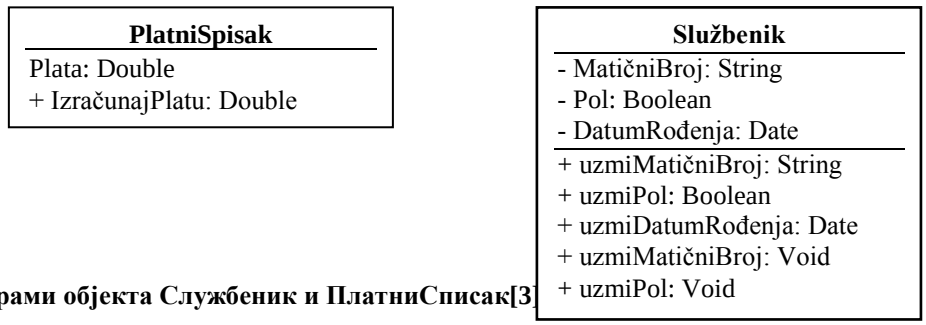
Као што смо већ видели, подаци класе представљају се атрибутима. Свака класа мора дефинисати атрибуте који ће чувати стање сваког од објеката формираних на основу те класе. У примеру, класа `Особа` дефинише својства за атрибуте `Име` и `Адреса`. Сваки објекат, формиран од дате класе, има исте методе. Методи могу реализовати понашања која се позивају из других објеката (на пример, поруке) или пружати унутрашње понашање класе. Унутрашња понашања су методи којима не могу приступати други објекти, споља (на пример, приватни методи). У ранијем примеру са класом `Особа`, понашања су `узмиИме()`, `задајИме()`,`узмиАдресу()`и `задајАдресу()`. Ови методи омогућавају осталим објектима да прегледају и измене вредности атрибута у датом објекту. Ово је у ОО системима сасвим уобичајена пракса. У већини случајева, приступ атрибутима унутар неког објекта контролише сам тај објекат – ни један објекат не би требало непосредно да мења вредности атрибута другог објекта.

1.4.2. Класни дијаграми

Класни дијаграми су најчешће коришћени дијаграми. Можемо разликовати битне (основне) класне дијаграме и напредне. Класни дијаграм описује типове објеката у систему и разне врсте статичких релација које постоје међу њима. Показују својства и операције класе, као и ограничења која се намећу на начин повезивања објеката (Слика 4).

Сваки класни дијаграм се рашчлањује у два засебна одељка (не посматрајући само име). Први одељак садржи податке (атрибуте), а други одељак садржи понашања (методе).

У класном дијаграму објекта `Службеник`, у одељку са атрибутима, налазе се `МатичниБрој`, `Пол` и `ДатумРођења`, док се у одељку са методима налазе методи који баратају овим атрибутима:



Слика 4: Класни дијаграми објекта Службеник и ПлатниСписак[3]

1.5. Енкапсулација

Једна од основних предности коришћења објеката састоји се у томе што објекат не мора да открива све своје атрибуте и понашања. У добро испројектованим ОО софтверу (заправо, оно што је опште прихваћено као добро), објекат треба да открије само интерфејсе преко којих се са њиме може комуницирати. Детаље, који нису важни за коришћење објекта, треба сакрити од осталих објеката. Ово се зове *енкапсулација*. На пример, објекат који израчунава квадрат неког броја, мора обезбедити интерфејс за добијање резултата. Међутим, унутрашњи атрибути и алгоритми, који се користе за израчунавање квадрата, не треба да буду доступни позивајућем објекту. Када се пројектују робустне класе, увек треба размишљати о енкапсулацији.

1.5.1. Интерфејси

Као што је раније објашњено, интерфејси су основно средство општења између објеката. Захтеви за пројектовање класе одређују интерфејсе, који омогућавају исправно формирање и рад објеката. Било које од понашања, које објекат нуди, мора бити позивано поруком, коришћењем једног од расположивих интерфејса. Интерфејс треба да потпуно опише како корисник класе комуницира са класом.

Ако погледамо малопре поменути пример – рачунање квадрата неког броја. У овом примеру, интерфејс се састоји од два дела:

- Како формирати објекат Квадрат

- Како објекту послати вредност чији квадрат треба да израчуна и како од њега добити коначни резултат

У основи, интерфејси не садрже атрибуте, само методе. Као што је овде раније објашњено, ако корисник треба да приступи неком атрибуту, онда треба направити метод који враћа атрибут. Ако корисник жели вредност атрибута, позива се метод који враћа вредност атрибута. На овај начин, атрибут је енкапсулиран у објекту, па га само матични објекат може променити или очитавати. Ово је од велике важности, нарочито у поступку тестирања. Ако контролишемо приступ атрибуту, када се јаве проблеми, не морамо бринути о праћењу сваког дела који може нехотице изменити атрибут.

1.5.2. Реализација

Интерфејс чине само јавни атрибути и методи. Корисник не би требало да види било који део реализације објекта – са објектом комуницира искључиво посредством интерфејса. Узимајући претходни пример можемо закључити да корисника не занима како се квадрат израчунава, докле год је резултат тачан. Због тога, начин изражавања, односно реализације, може се изменити, а да то ни на који начин не поремети корисников код.

1.6. Конструктори

За људе који структурно програмирају, *конструктори* представљају потпуно нов концепт. Конструктори не постоје у језицима који нису ОО, рецимо у C-у и Basic-у. У Јави и језику C++, као и у осталим ОО језицима, конструктори су методи за које не постоји тип враћене вредности и зову се, исто као и класа, којој припадају. Када се формира нови објекат, позивање конструктора је једна од ствари које се тада дешавају.
Пример:

```
Taksista  mojTaksista = new  Taksista ( );
```

Службена реч `new` ствара нови примерак класе Таксиста и на тај начин резервише потребну меморију. Тада се позива конструктор, при чему се аргументи прослеђују посредством листе аргумената. Програмер мора извршити одговарајућу иницијализацију унутар конструктора. Због тога ће кôд `new Taksista()` формирати објекат `Taksista` и позвати метод `Taksista`, који је конструктор.

Најважнија функција конструктора јесте иницијализовање меморије, која се додељује када се наиђе на службену реч `new`. Укратко, кôд који се налази у конструктору треба да нови објекат постави у његово почетно, стабилно стање.

Ако при писању класе у њу не укључимо конструктор, класа ће се још увек преводити и још увек је можемо користити. Ако класа нема изричито наведени конструктор, као што је у Јави и језику C++, преводилац ће направити подразумевани конструктор. Подразумевани конструктор позива једино конструктора надређене класе. Можда је подразумевани конструктор довољан у неким случајевима, ипак, у већини случајева мора се извршити нека врста иницијализације меморије. Без обзира на ситуацију, сматра се да је увек добро у класу укључити бар један конструктор. У сваком случају, ако у класи постоје атрибути, њих треба иницијализовати у конструктору.

1.7. Наслеђивање

Једно од најмоћнијих својстава ОО програмирања јесте вишеструка употребљивост програмског кода. Процедурално програмирање омогућује известан степен вишеструке употребљивости програмског кода – можемо написати процедуру и потом је користити колико год пута желимо. Међутим, ОО програмирање чини веома важан корак даље, јер омогућава дефинисање односа између класа. На тај начин, поред олакшавања вишеструке употребљивости програмског кода, олакшава се и целокупно пројектовање. Срж је у организовању класа и разврставању заједничких особина различитих класа. *Наслеђивање* је основно средство које пружа ову функционалност.

Наслеђивање дозвољава класи да наследи атрибуте и методе неке друге класе. То вам омогућава да направите потпуно нове класе, при томе апстрахујући заједничке атрибуте и понашања.

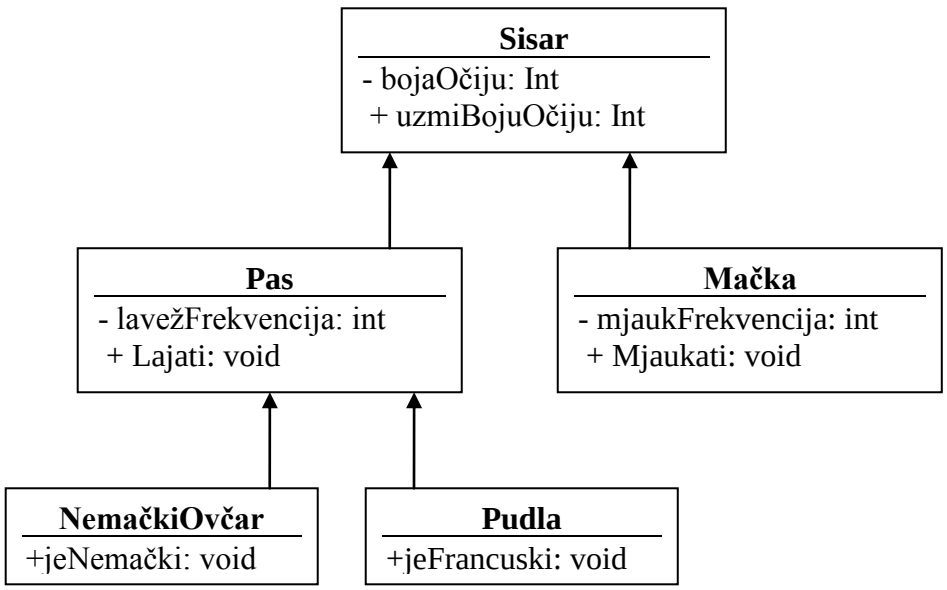
Један од главних проблема који се јављају у ОО програмирању јесте разврставање заједничких особина различитих класа. На пример, имамо класу `Pas` и класу `Mačka`. Свако од ових класа има атрибут за боју очију. У свету процедуралног програмирања, атрибут за боју очију би се садржао у програмском коду обе процедуре. У ОО програмирању, атрибут за боју очију био би апстрахован све до нивоа класе `Sisar` – заједно са свим осталим заједничким атрибутима и методима. У овом случају класе `Pas` и `Mačka` наслеђују класу `Sisar` (Слика 6).

1.7.1. Надређене и подређене класе

Наткласа, односно надређена, родитељска класа садржи све атрибуте и понашања који су заједнички за све класе које је наслеђују. Стога, класе `Pas` и `Mačka` наслеђују све заједничке атрибуте и понашања од класе `Sisar`. Класа `Sisar` се сматра родитељском, односно надређеном класом, док су класе `Pas` и `Mačka` подређене класе, односно класе потомци.

Наслеђивање пружа богате могућности и предности приликом пројектовања. Када пројектујемо класу `Mačka`, класа `Sisar` нам пружа велики део потребне функционалности. Наслеђивањем објекта `Sisar`, `Mačka` већ има атрибуте и понашања које је чине истинским сисаром. Прецизније, класа `Mačka`, између осталог, мора садржати атрибуте и понашања која су својствена само мачки.

Стабло наслеђивања (Слика 6) може поприлично нарасти. Када се употпуне класе `Sisar` и `Mačka`, остали сисари се лако могу додати. Моћ механизма наслеђивања лежи у апстракцији и техникама организације.

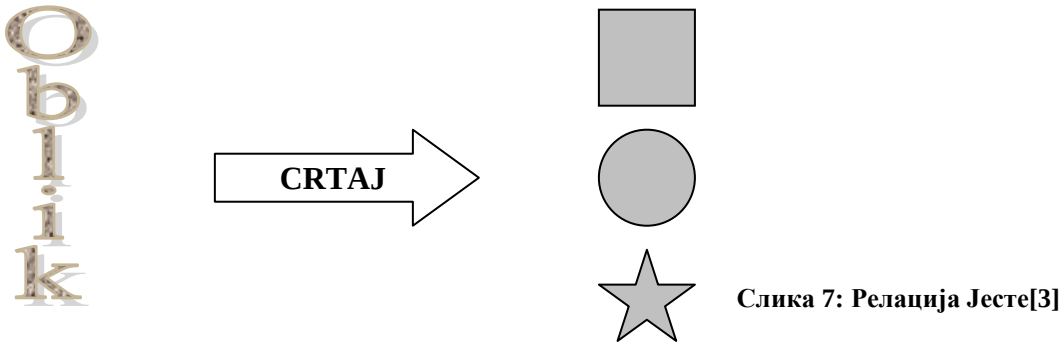


Слика 6: UML Дијаграм[3]

1.7.2. Релација Јесте

Имамо пример са класом `Oblik` и класама које је наслеђују: `Krug`,`Kvadrat`,и `Zvezda`. Овај однос се често назива релацијом **јесте**, јер круг јесте облик. Када подређена класа наследи надређену класу, она може да уради све што може да уради надређена класа. Када треба нацртати неки од облика, само треба позвати метод

`Crtaj`, без обзира који је облик у питању. У овом примеру, сваки појединачни објекат је одговоран за своје цртање (Слика 7).



Слика 7: Релација Јесте[3]

1.8. Полиморфизам

Полиморфизам у преводу значи појављивање у више облика. Иако је полиморфизам уско везан са наслеђивањем, често се засебно наводи као једно од највећих предности објектно оријентисаних технологија. Када се објекту пошаље порука, објекат мора имати дефинисан метод за пружање одговора на поруку. У хијерархији наслеђивања, све подређене класе наслеђују интерфејсе од својих надређених класа. Међутим, пошто је свака класа засебна целина, свака од њих може захтевати себи својствен одговор на исто питање. На пример, ако узмемо у обзир класу `Oblik`, са понашањем `Crtaj`. Када неком кажете да нацрта облик, одмах ће питати: Који облик? Он или она не може нацртати неки општи облик јер је то сувише апстрактно. Морамо задати неки одређени облик. Да бисмо то урадили, морамо обезбедити реализацију, на пример, у класи `Krug`. Иако `Oblik` има метод `Crtaj`, `Krug` занемарује овај метод и обезбеђује сосптвени метод `Crtaj( )`. *Надјачавање* (engl. Overriding), у основи значи замењивање родитељске реализације, реализацијом потомка.

На пример, претпоставимо да имамо низ од три облика – `Krug`, `Kvadrat` и `Zvezdu`. Иако са свима поступате као са објектима наслеђеним из класе `Oblik` и свима шаљемо поруку `Crtaj`, коначни исход је другачији, јер `Krug`, `Kvadrat` и `Zvezda` обезбеђују конкретну реализацију. Укратко, свака класа је способна да другачије одговори на исти метод `Crtaj` и да сама себе нацрта.

II. .NET Framework

2.1. О .NET-у

.NET је стратешка иницијатива Microsofta за наредну деценију намењена за развој клијентских и серверских апликација. Microsoft.NET развојно окружење је такав скуп софтвера који помаже повезивање информација, људи, рачунарских система и хардверских уређаја. Његови делови су клијенти (на пример, Windows XP), сервери (Microsoft SQL server) и развојни алати (Visual Studio.NET 2008). Microsoft- ово .NET развојно окружење представља такав платформ за развој софтвера која подржава приступ брзом развоју апликације (RAD – Rapid Application Development), платформску независност и мрежну транспарентност. Microsoft.NET развојно окружење садржи више таквих технологија која су дизајнирана за брзи развој Интернет и интранет апликација. Поред тога подржава развој Windows и Web апликација, компонената и сервиса. Microsoft.NET развојно окружење је довољно опширан да може да преведе више програмских језика високог нивоа. Постоји више алата намењених за развој у .NET окружењу, од којих је најзначајнији Visual Studio.NET, Microsoft- ово развојно окружење (IDE – Integrated Development Environment).

Microsoft.NET развојно окружење обезбеђује најопширније и најефикасније алате за израду, имплементацију и одржавање Web сервиса захваљујући технологијама заснованим на XML-у. Захваљујући XML заснованим Web сервисима апликације комуницирају преко Интернета и размењују податке независно од оперативног система и програмског језика. Применом алата .NET развојног окружења пројектанти прилично једноставно могу развијати такве Windows апликације који спајају XML засноване Web сервисе са традиционалним апликацијама, унапређујући тако комуникацију и сарадњу унутар и изван организације.

Visual Studio .NET је стандардизовано окружење за развој клијентских и серверских апликација и сервиса, на језику који сам пројектант изабере, и то брже и лакше него икад. Заснован је на стандардима који су данас широко прихваћени, као што су XML, SOAP, HTTP и HTML. У Visual Studio .NET-у XML је присутан свуда, а пројектант не мора да зна где ни како је уграђен. За пројектанта је све транспарентно; једино о чему треба да води рачуна јесте писање кода.

2.2. Историјат настанка .NET технологије

World Wide Web, данас најпознатији и најкоришћенији део Интернета, започео је свој развој 1992. године. Својим убрзаним развојем увелико је допринео огромној популарности Интернета у свету. Данас већина људи говорећи о Интернету, заправо говори о Web-у і Web страницама. Паралелно са развојем Weba, текао је и развој технологија које су омогућавале његову имплементацију у сва подручја компјутерског света, али и шире.

Историјски гледано, први у низу Web програмских језика био је PHP (Hypertext Pre-Procesor) створен 1995. године. Нешто мало пре популаризације PHP-а 1997. године, Microsoft је издао прву верзију свог web програмског језика названог ASP (Active Server Pages). Пошто је Microsoft- ов ASP још и 2000. године по својим могућностима увелико заостао за PHP- ом који је под Open Source лиценцом значио бесплатно, управо због тога Microsoft је престао развијати стару технологију и окренуо се изради нове.

Неки од .NET технологија настала су током израде Microsoft- ове Java верзије. Када је 1998. године Microsoft одлучио да више неће применити Sun Java технологију, дотле израђени Microsoft Java++ производи послужили су као основа .NET пројекта. Неки сматрају да .NET- ов CLR код потиче од OmniVM, производа Colusa Software- а, коју је Microsoft откупио 12. марта 1996. године.

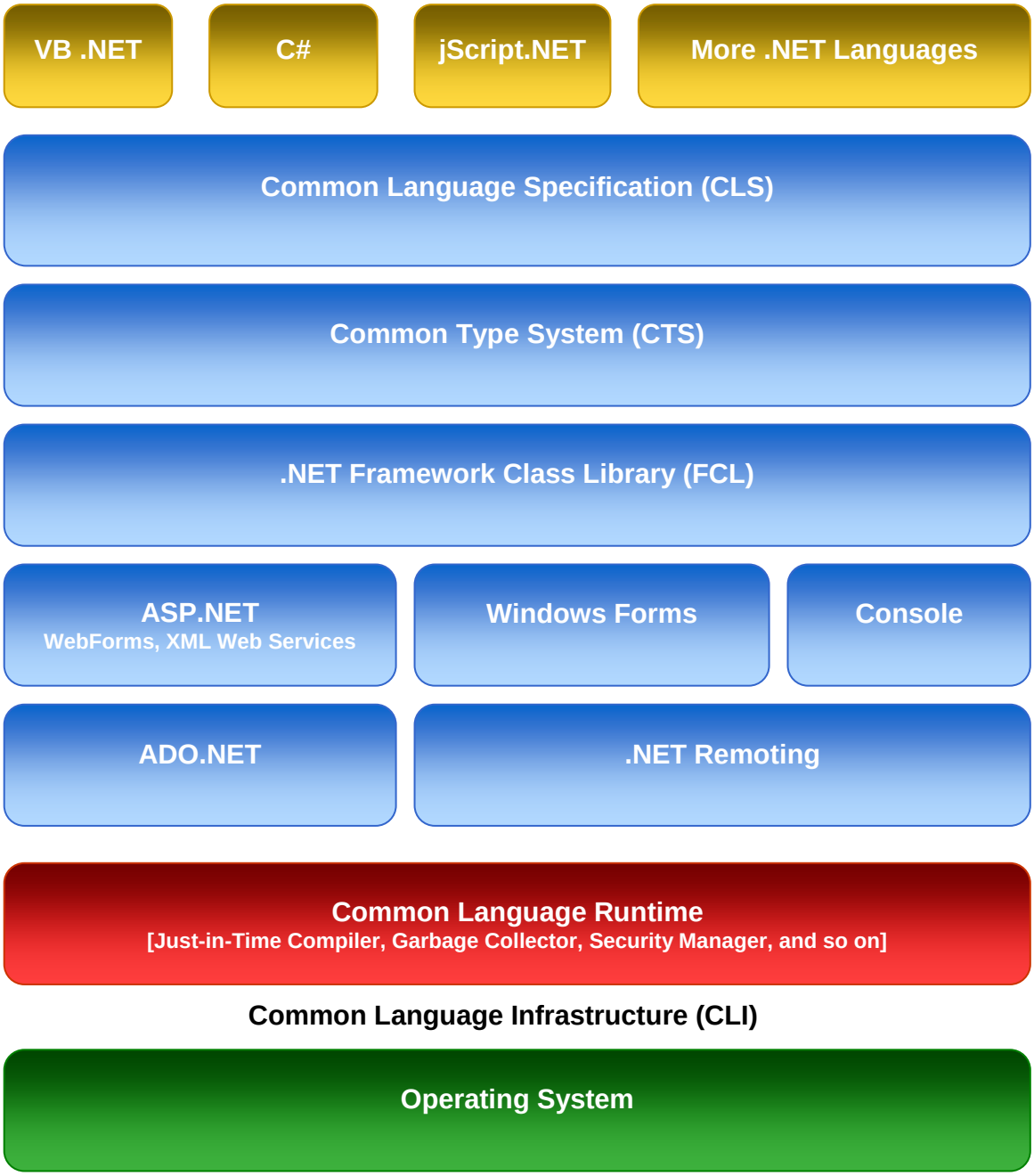
Почетком 2001. године Microsoft је објавио основну архитектуру своје нове технологије названу .NET. Средином 2002. године финализиран је .NET Framework 1.0 і MS Visual Studio 2002. .NET као софтверска платформа постала је доступна од 2002. године.

2.3. Основне карактеристике .NET развојног окружења

У **.Net** архитектури і **.Net Framework** окружењу неопходно је разјаснити појмове као што су:

- **The Common Language Ryntime (CLR),**
- **MSIL Code(Microsoft Intermediate Language),**
- **Jyst In Time Compilers (JITers),**
- **The Framework Class Library (FCL),**

- **The Common Language Specification (CLS),**
- **The Common Type System (CTS),**
- **Garbage Collection (GC),**
- **Common Language Infrastryctyre (CLI).**



Слика 7: Основна структура Microsoft® .Net Framework-a

2.3.1. Common Language Ryntime (CLR)

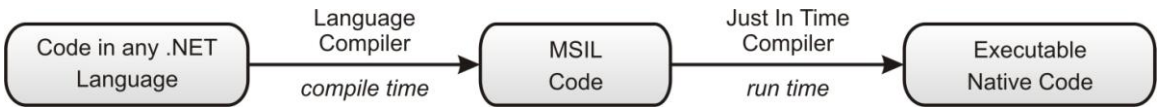
Темељ на коме је изграђено .NET развојно окружење јесте CLR (engl. Common Language Ryntime). CLR представља извршно окружење које управља .NET кодом у

току извршавања.CLR је пројектован тако да подржава код који може бити написан у различитим језицима (Слика 8).

Слика 8: Комуникација помоћу CLR слоја

2.3.2. Microsoft Intermediate Language (MSIL) код

Да би се постигла подршка за међуплатформске језике, сви .NET програми компајлирани су пре самог процеса развијања у језик ниског нивоа који се назива међујезик (engl. Intermediate Langyage - IL). Microsoft - ова имплементација овог језика назива се Microsoft међујезик (engl. Microsoft Intermediate Langyage) или скраћено MSIL (Слика 9). Затим се овај IL код у право време компајлира у матични код у време извршавања. То значи да се .NET извршни и DLL кодови, независно од оригиналног језика изворног кода, увек развијају у IL, те стога не постоји разлика између компоненти које су првобитно биле написане у програму C# и оних написаних у VB.NET-у. Ова карактеристика подржава медујезичку интероперабилност (нпр. Способност да се изведе класа која је написана у једном језику из класе која је написана у било ком другом .NET језику). Такође омогућава извођење апликација без модификације у неку другу подржану платформу (тренутно Windows 9x/ME; Windows NT4, Windows 2000 или Windows XP).



Слика 9: Посредство MSIL кода

2.3.3. Just In Time Compilers (JIT-ers)

Када је при извршавању нашег програма неопходно извршити егзекуцију MSIL кода, CLR позива JIT компајлер који компајлира MSIL код у извршни код (.exe на пример) који је прилагођен одговарајућем хардверу и оперативном систему. JIT компајлери се у многоме разликују од класичних компајлера јер они компајлирају MSIL код у извршни само када је то неопходно при извршавању програма. На пример када је за извршавање нашег програма неопходно позвати неку функцију JIT компајлер преводи MSIL код у извршни правовремено, баш у том тренутку (Jyst In Time). На тај начин део програмског кода који није позван од стране програма при његовом извршавању се не преводи у извршни код. У случају да је део програмског кода већ компајлиран и позива се поново, CLR преузима исту (претходно компајлирану) копију без поновног компајлирања.

Пошто JIT компајлери препознају специфичне хардверске карактеристике (процесор) и верзију оперативног система они су способни да изврше веома ефикасну оптимизацију програмског кода што резултује у поузданим и ефикасним апликацијама са великом брзином извршавања.

Microsoft обезбеђује компајлере за пет .NET језика:

- C# - нови C језик који је дизајниран специјално за .NET радно окружење.
- Visual Basic.NET – верзија језика Visual Basic која је ажурирана за .NET развојно окружење (на пример, са потпуно објектно оријентисаним карактеристикама, структурираним руковањем изузецима и још много карактеристика које VB програмери захтевају годинама!).
- Управљани C++ – C++ са „управљаним проширењима” за подршку .NET карактеристика које не могу бити имплементиране употребом постојећих карактеристика језика. За разлику од остала три језика, C++ компајлер не иде бесплатно у пакету са .NET Framework SDK, већ са програмом Visual Studio.NET.
- JScript.NET – Microsoft имплементација скрипт језика JavaScript која је ажурирана за употребу у .NET развојном окружењу.
- J# - првобитно Visual J++ (укључујући и Microsoft проширења за програм Java попут COM подршке) за .NET развојно окружење.

2.3.4. Common Language Specifications (CLS)

Сви .NET компатибилни програмски језици могу икористити повољности које нуде CLR слој и FCL библиотека класа. Оно што један програмски језик чини .NET компатибилним је Common Langyage Specification (CLS) – мали скуп општих захтева који се тичу саме структуре програмског језика. Пошто је MSIL веома опширан, односно богат програмски језик, није неопходно да сваки од језика из .NET окружења обухвати све његове особине тј његову комплетну функционалност, већ само одговарајући мањи подскуп његових особина. Ова особина омогућава да се под окриљем .NET окружења данас извршавају многи процедурални програмски језици. Као пример стандарда које CLS прописује су: непостојање глобално декларисаних функција, нема показивача, нема вишеструког наслеђивања класа и сл. Најважнија последица наведеног је да, ако се наш програмски код налази унутар ограничења које поставља CLS он може бити искоришћен у било ком од .NET програмских језика.

2.3.5. Common Type System (CTS)

Common Type System (CTS) представља основу на којој су изграђене међујезичке карактеристике CLR-а. Да би класе које су дефинисане у различитим језицима могле међусобно да комуницирају, потребан им је заједнички начин представљања података – заједнички скуп типова података. Сви претходно дефинисани типови који су доступни у IL, дефинисани су у CTS-у. То значи да се сви подаци у .NET коду складиште у истим типовима података, с обзиром на то да су сви .NET кодови компајлирани у IL-у.

CTS прави разлику између две основне категорије типова података – типови вредности (engl. value types) и типови референце (engl. reference types). Типови вредности (укључујући и већину уграђених типова, као и структура и набрајања) садрже сопствене податке. На пример, променљива типа целог броја складишти цео број директно у програмски стек. Типови референце (укључујући и String и Object, као и низове и већину кориснички дефинисаних типова попут класа и интерфејса) складиште само референцу за сопствене податке у стеку – сами подаци складиште се у различитим областима меморије које су познате као хип (engl. heap). Разлика између поменутих типова је евидентна у случају преношења параметара методи. Сви параметри методе се према подразумеваним вредностима преносе помоћу вредности, а не референце. Међутим, у случају типова референце, вредност не представља ништа више од референце за локацију на хипу где се складиште подаци. Као резултат,

параметри типа референце понашају се слично као аргументи које преноси референца – променом вредности промењиве унутар методе такође ће утицати на оригиналну промењиву.

2.3.6. Garbage Collection (GC)

Једна од најбитнијих функција које нам обезбеђује CLR јесте процес сакупљања отпадака (engl. garbage collection).

Наиме, приликом формирања објекта у програмима попут C i C++, меморија која се користи за поменути процес мора бити ослобођена како би се могла опет употребити. У случају неизвршења, долази до тзв. „цурења меморије" – неупотребљена меморија коју систем не може повратити. Наиме, што је количина пропуштене меморије већа, то су перформансе апликације све лошије. Међутим, управо због тешке уочљивости грешке и њеног временски продуженог дејства, праћење поменутих грешака представља веома тежак процес. CLR решава наведени проблем имплементирањем сакупљача отпадака (engl. garbage collector). У одредјеним временским интервалима (када нема простора у расположивој меморији), сакупљач отпадака проверава све референце објекта и ослобада меморију у објектима који су изашли из подручја важења и који нису доступни апликацијама. Поред тога што се решава проблем „цурења" меморије, овим процесом програмер је ослободен експлицитног уништавања објеката.

2.3.7. Common Language Infrastructure (CLI)

Упркос томе што је .NET развојно окружење тренутно доступно само у Windows платформама, Microsoft је поднео подскуп .NET развојног окружења (CLI, заједничка језичка инфраструктура) Европском удружењу произвођача рачунара (ECMA) као предлог за један од стандарда. Наиме, посебне верзије CLI су у процесу развијања за оперативне системе FreeBSD иі Linyx. Слично томе, спецификације за програмске језике C# и IL (првобитни назив за општи међујезик [engl. Common Intermediate Langyage] или CIL) су такође предложене удружењу ECMA, па це стога и CLI имплементације које не припадају породици Microsoft имплементирати наведене спецификације.

2.3.8. Framework Class Library (FCL)

Следећа карактеристика представља најважнију карактеристику од свих поменутих – огроман скуп библиотека класе за обављање сваког могућег задатка у програмирању. Класе и остали типови у оквиру .NET радног окружења организоване су у просторе имена (engl. namespaces) сличне онима у Java пакетима. Поменути простори имена могу бити угнеждени унутар других именованих простора, чиме нам је омогућено да идентификујемо своје класе и разликујемо их од класа са истим називом које потичу од друге компаније.

Заједно са .NET радним окружењем, Microsoft обезбеђује огроман скуп класа и других типова, највише у оквиру именованог простора System или у оквиру једног од многих угнеждених именованих простора. У ову категорију спадају и примитивни типови попут целих бројева, где типови C# int и VB.NET Integer представљају само псеудониме за тип System.Int32. Међутим, у поменуту групу спадају и класе које се користе за Windows апликације, Web апликације, сервисе директоријума, приступ датотекама, укључујући и приступ подацима. Поменуте класе приступа подацима познате су као ADO.NET. Наиме, .NET програмирање представља ефективно програмирање са библиотекама .NET класа – немогуће је написати програм у језику C# или VB.NET који не користи ове библиотеке.

.NET Framework библиотеке класа укључују класе које подржавају следеће функције:

- основне и кориснички дефинисане врсте података;
- подршка за рад са изузецима;
- улаз/излаз и стреам операције;
- комуникација са подсистемима;
- приступ подацима;
- могућност за креирање Windows заснованим GUI апликацијама;
- могућност за креирање web клијент и сервер апликација;
- подршка за креирање web сервиса;

Све класе имплементиране у .NET библиотеци класа су смештене у namespace-овима. Сваки namespace садржи класе и друге типове који су повезани са одређеним

задатком или сетом задатака – улазно излазним операцијама, креирањем web апликација, радом са подацима и XML-ом, итд.

Табела 1: Најважнији namespace-ови[1]

Namespace	Опис
System	Садржи основне калсе и базе класа које дефинишу уобичајену вредност и упиђуј на типове података, догађаје, интерфејсе, обрада изузетака, конвертовање података, математичке функције, као и гарбаге цоллецтион и управљањем окружења апликације.
System.IO	Обезбеђује клсе које подржавају асинхроно и синхроно читање из и уписивање у стреам податке и датотеке. Садржи класе као што су: FileStrem, MemoryStream, Path и Directory.
System.CodeDom	Ова библиотека омогућује способност за креирање кодова и њихово покретање.
System.Configuration	Обезбеђује инфраструктуру за рад са конфигурацијом података.
System.Collections	Садржи интерфејсе и класе које дефинишу различите колекције објеката, као што су листе, низови, хасх табеле, стекови
System.Threading	Обезбеђује класе и интерфејсове које подржавају мултипрограмирање.
System.Reflection	Садржи класе које омогућавају динамичке повезивање и регулисању разгледање учитаних типова, метода и поља.
System.Security	Имплементира безбедносни систем, укључујући основне класе за дозволе. Укључује класе као што су: SecurityManager, PermisisonSet i CodAccessPermission.
System.Net	Обезбеђује подршку за интернет програмирање. Укључује класе као што су: NttpWebReqyest, IPAddress, Dns i Connection.
System.Data	Садржи класе које имплементирају ADO. NET. Child namespace укључује OleDb i SqlClient.
System.Diagnostic	Даје могућност за дијагнозу апликације.
System.Managment	Даје могућност за добијање информација (као на пример: колико је слободног места на диску остало, колика је искоришћеност CPU – а итд.)
System.Media	Пружа могућност за репродуковање системског звука и wave фалова.
System.Messaging	Омогућује повезивање, надгледање и управљање низом порука у мрежи и слање, пријем и читање порука.
System.Runtime	Омогућава управљање апликацијом за време извршавања или управљање CLR – ом.

2.4. Приступ подацима у .NET-у

У почетку, програмски приступ базама података спроводио се кроз матичне библиотеке, попут DBLib за SQL Server и Oracle Call Interface (OCI) за Oracle. Управо овим је и омогућен брз приступ базама података зато што није постојао додатни слој – кодови су били написани за директан приступ бази података. Међутим, то је подразумевало да су пројектанти морали да науче нови скуп рутина API за сваки систем базе података којој желе да приступе, а у случају да је требало ажурирати апликацију како би функционисала у различитом систему базе података, сви кодови приступа подацима су морали да се промене.

2.4.1. ODBC

Microsoft је раних 90-их година у сарадњи са другим компанијама развио тзв. повезивост отворених база података (engl. Open Database Connectivity) или скраћено ODBC, и то као решење претходно напоменутом проблему. Овим се обезбедио заједнички слој приступа подацима који се могао користити за приступање скоро сваком систему управљања релационом базом података (RDBMS).

Наиме, ODBC користи специфичан RDBMS управљачки програм за комуницирање са извором података. ODBC управљачки програм (engl. Driver Manager) учитава и управља поменутим управљачким програмом (понекад су у питању само овојнице матичних API позива).

Овим је такође обезбеђена карактеристика попут удруживања веза (engl. connection pooling) – способност да се поново употребе повезивања, уместо уништавања истих при њиховом затварању и стварања новог повезивања сваки пут када се приступи бази података. Наиме, апликација комуницира са управљачким програмом кроз стандардни API, те стога (теоријски гледано), ако желимо да ажурирамо апликацију како бисмо се повезали са различитим системом управљања релационом базом података (RDBMS), потребно је да променимо само детаље повезивања (у пракси, постојале су одредене разлике у SQL дијалекту). Међутим, једна од најбитнијих карактеристика ODBC-а је чињеница да је ODBC био отворени стандард прихваћен чак и од стране асоцијације Open Source. Као резултат, ODBC управљачки програми развијени су за многе системе базе података којима се не може директно приступити помоћу каснијих технологија приступа подацима.

2.4.2. DAO

Један од проблема везаних за ODBC јесте да је дизајниран за употребу из језика ниског нивоа попут C++. Наиме, упоредо са растом важности програма Visual Basic, расла је и потреба за технологијом приступа подацима који би се користили на једноставнији начин из VB-а. Овај проблем решен је у VB 3 стварањем објеката приступа подацима (Data Access Objects – DAO). Наиме, DAO је обезбедио једноставан модел објекта за комуницирање са Jetom, механизмом базе података у позадини Microsoft Access радне површине базе података. DAO је оптимизован за Access (иако се може употребити за повезивање са ODBC изворима података) и представља најбржи начин за комуницирање са Accessom из VB.

2.4.3. RDO

Због оптимизације за Access, DAO је постао преспор за употребу са ODBC изворима података. Да би решио проблем, Microsoft је у верзији VB 4 (32-битна верзија) представио тзв. Удаљене објекте података (Remote Data Object – RDO). Наиме, RDO обезбеђује једноставан модел објекта сличан DAO моделу који је дизајниран специјално за приступ ODBC изворима података. У суштини, RDO представља танак омотач преко ODBC API.

2.4.4. OLE DB

Још један велики земљотрес у свету технологија приступа подацима изазвало је издање OLE DB-а. По структури, OLE DB личи на ODBC: комуникација са извором података врши се преко OLE DB добављача (сличан концепту ODBC управљачких програма), који су дизајнирани за сваки подржани тип извора података понаособ. Наиме, OLE DB добављачи имплементирају скуп COM интерфејса који дозвољавају приступ подацима у стандардном формату ред/колона. Апликација која користи поменуте податке позната је под називом OLE DB корисник (engl. OLE DB consumer). Као што ови стандардни добављачи података изводе податке из извора података и чине их доступним кроз OLE DB интерфејсе, тако и OLE DB садржи одређени број добављача услуга. Овим се формира "средњи низ" OLE DB архитектуре, обезбеђујући услуге које се употребљавају са добављачем података. У поменуте услуге спадају и удруживање веза, регистровање трансакција (способност да се аутоматски региструју MTS/COM компоненте у оквиру MTS/COM трансакције), постојаност података,

клијентску манипулацију подацима (Client Cursor Engine или CCE), хијерархијске скупове записа (обликовање података) и прављење података на удаљеној машини.

Међутим, стварна иновација која се крила иза концепта OLE DB била је Microsoft-ова стратегија за универзални приступ подацима (engl. Universal Data Access – UDA). Наиме, идеја која се налази иза поменутог концепта UDA јесте да се подаци складиште на многим местима: е-пошта, Excel табеларни прорачун, Web странице итд. – као уосталом и у традиционалним базама података, па би стога требало да смо у могућности да програмски приступимо свим поменутим подацима, и то кроз једну унификовану технологију приступа подацима. Управо OLE DB представља основу за Microsoft-обу имплементацију ове стратегије. Број OLE DB добављача постепено се повећавао да би се покрили како системи релационих база података (чак и MySQL база података поседује OLE DB добављач), тако и нерелациони извори података попут Exchange 2000 Web Store, датотеке Project 2000 и IIS виртуелни директоријуми. Међутим, Microsoft је пре појаве ових добављача обезбедио широки спектар подршке за OLE DB – OLE DB добављач за ODBC управљачке програме. Стога је од самог почетка било јасно да се OLE DB могао користити за приступ било ком извору података који садржи ODBC управљачки програм. Уосталом, поменута тактика прихваћена је и у ADO.NET-у.

2.4.5. ADO

ActiveX Data Objects (ADO) представља технологију која је свој назив позајмила програму ADO.NET (иако су разлике између поменутих технологија прилично велике). Наиме, ADO је само један од OLE DB корисника – танак слој који омогућава корисницима језика високог нивоа попут VB-а и скрипт језика да приступе OLE DB изворима података кроз једноставан модел објекта; ADO је за OLE DB скоро исто што је и RDO био за ODBC. Популарност ADO-а лежи у чињеници да је омогућио великом броју пројектаната Visual Basic-а, ASP-а и Visual J++ једноставан приступ подацима на различитим локацијама. Ако је OLE DB био основа на којој је изграђен UDA, ADO је био одора у којој се он представљао већини пројектаната. ADO и даље представља исправан избор за пројектанте који раде у .NET развојном окружењу.

2.5. Склопови

.NET код се развија као склоп (engl. assembly). Склопови се састоје из компајлираних IL кодова и притом морају да садрже једну примарну датотеку (осим у случају динамичких склопова који се у потпуности складиште у меморији). Поменута датотека може бити извршна датотека (.exe), DLL или компајлирана ASP.NET Web апликација или Web сервис.

Склоп може да садржи ресурсне датотеке (попут слика и икона) и остале модуле кода, као и свака примарна датотека. Међутим, као најбитније, склопови могу да садрже метаподатке. Поменути метаподаци састоје се из два дела: тип метаподатка садржи информацију о свим извезеним типовима и њиховим методама који су дефинисани у склопу. .NET склопови, као и IL кодови, садрже део који се зове манифест. Овај део садржи метаподатке склопа (engl. assembly metadata) или информацију о самом склопу, као што су број верзије и број изградње.

Поменути метаподаци дозвољавају склопу да у потпуности буде самоописив. Наиме, склоп садржи све информације потребне за инсталирање и активирање апликације, те стога нема потребе за библиотекама типа или регистара уноса. Поменути процес инсталирања може бити једноставан као и процес копирања склопа на циљну машину. Самим тим што склоп садржи информације о верзији, више верзија исте компоненте могу се инсталирати заједно на истој машини. Управо ова могућност решава проблем познат као „DLL пакао” (engl. DLL Hell), где апликација која инсталира нову верзију постојеће компоненте прекида програме који користе стару верзију.

2.6. .NET програмски језици

CLI је тако дизајниран да подржава било који објектно оријентисани програмски језик, који користи заједнички објект модел и библиотеку основних класа. .NET тренутно подржава више од четрдесет програмских језика. Многи језици су значајно подешени како би се уклопили у .NET развојно окружење. Призвођачи су често користили ову прилику да побољшају постојеће и да додају друге језичке функције.

.NET програмски језици се деле у две групе: на уграђене језике, и на оне који су доступни из спољних извора.

У групу уграђених програмских језика спадају:

- C#
- Jscript.NET, подешена верзија JScript језика
- J#, прелазни језик Java и J++ програмерима у правцу .NET развојног окружења
- Managed C++, подешена верзија C++ језика
- Visual Basic .NET, побољшана објектно оријентисана верзија Visual Basic језика

Неки од језика доступних из спољних извора су:

- Cobol
- Delphi 8 i Delphi 2005
- Fortran
- Lisp
- Mercury
- Perl
- Smaltalk itd.

Нови концепт ових .NET језика је објектно - оријентисано програмирање, који поједностављује вишекратну употребу и интероперабилност компонената. Све класе у објектном окружењу су потпуно објектно- оријентисане. Захваљујући потпуно објектно- оријентисаном BCL-у могу се обезбедити OO особине, као што су полиморфизам и наслеђивање помоћу компонената написаних на различитим језицима. То представља кључни фактор за поједностављивање развоја у великим компанијама у којима неки пројектанти могу да користе Visual Basic.NET, док други можда користе Cobol.NET или C#. Без обзира на језик за који се опредељује, исте могућности су доступне свима.

Сви језици, без обзира на посебност одређеног језика и даље морају да се преводе у CIL, који затим интерпретира машина за извршавање (извршна компонента) која је, пак, специфична за одређени процесор. То значи да су сви језици равноправни. Сви језици морају да подржавају функције .NET развојног окружења или се иначе не би сматрали за .NET језике.

У неким језицима постоје специфичности којих у другим језицима нема. То могу да буду одређене уграђене функције, резервисане речи или семантика језика, међутим када се формира датотека њен садржај је увек CIL (Common Intermediate Language). Преводилац, који је специфичан за сваки језик, обезбеђује да код не крши правила рада

са појединим типовима и да ће успешно проћи поступак исправности CIL. То не значи да нека правила не могу да се прекрше; па би требало проучити CLS и CTS како би били сигурни да се користе типови усклађени са CLS-ом.

2.7. Предности и мане .NET развојног окружења

Постоји неколико критика које указују на мане .NET развојног окружења:

- Увођењем .NET развојног окружења стари Visual Basic језик је замењен новим Visula Basic .NET језиком што је довело до нејасноћа и неспоразума међу програмерима.
- Постоје неке некомпатибилности унапред и уназад између верзије .NET. Међутим, ради избегавања ових проблема омогућено је истовремено функционисање више верзија развојног окружења.
- Оне апликације које се извршавају у неком развојном окружењу обично захтевају више ресурса система у односу на функционално сличне апликације које директније приступају ресурсима рачунара. Међутим, неке апликације су перформантније баш у .NET окружењу у односу на њихово изворно окружење захваљујући JIT извршавању и другим аспектима CLR-а.
- Постоји забринутост међу програмерима због чињенице да из .NET склопа, произведеног од стране .NET развојног окружења, могу се открити неке програмерске технике и алгоритми које користи апликација, а тиме могу да се губе пословне тајне и омогуће је заобићи лиценсне контролне механизме.

Дизајнерима софтвера .NET развојно окружење представља значајну промену, пошто он укључује такве особине и одговорности у оквир оперативног система, који су раније били обезбеђени појединачно, помоћу програмских језика и алата из разних извора. Укључивање ових особина у оперативни систем значи појаву значајних предности, као што су:

- Обезбеђивање приступачности особина развојног окружења свим програмима писаним у било ком .NET програмском језику.
- Програмерима је омогућен једнаки приступ особинама развојног окружења, без обзира на програмски језик који користе.

- Загарантовано је једнако понашање и извршавање апликација у оквиру развојног окружења, без обзира на коришћени програмски језик.
- Сам оперативни систем гарантује понашање програма што иначе не би било могуће.
- Смањење комплексности и ограничења међусобне комуникације између програма, чак иако су ови програми написани у различитим .NET програмским језицима.

III. Програмски језик C#

Рачунар нема способност да размишља и његов рад се темељи на извршавању наредби (инструкција) које му програмер задаје у облику програма. Под програмом се подразумева скуп инструкција које изводи и контролише рачунар. Да би се остварила комуникација између човека и машине користи се програмски језик. Рачунар разуме машински језик заснован на бинарној азбуци и логици. Писање програма на машинском језику је компликовано па су се зато развили програмски језици које пружају програмерима више комфора у раду и који су лакши за учење.

C# је разоружавајуће једноставан језик, са само 80 резервисаних речи и десетак уграђених типова података, али је веома изражајан када је потребно имплементирати модерне програмске идеје. Подржава структуирано, објектно оријентисано програмирање засновано на компонентама, што бисмо и очекивали од модерног језика насталог на основу Jave и C++-а.

Језик C# створио је мали тим предвођен двојицом цењених Microsoft-ових инжењера, Андерсом Хајлсбергом и Скотом Вилтамутом. Хајлсберг је и творац Turbo Pascala, популарног језика за програмирање на личним рачунарима, и вођа тима који је осмислио Borland-ов Delphi – једно од првих успешних интегрисаних развојних окружења за програмирање клијентско - серверских апликација.

3.1. Основе C# језика

Срж сваког објектно оријентисаног језика јесте подршка за дефинисање класа и рад с њима. Помоћу класа дефинишемо нове типове, што омогућава да језик проширимо како бисмо направили што бољи модел проблема који покушавамо да решимо. Ц# садржи *резервисане речи* (engl. keywords) за декларисање нових класа и њихових метода и својстава, као и за *капсулирање*, *наслеђивање* и *полиморфизам* – три камена темељца објектно оријентисаног програмирања.

У језику C#, све што се односи на декларацију класе налази се у самој декларацији. За дефиницију класа нису потребне засебне датотеке заглавља (engl. header files) нити датотеке на језику за дефинисање интерфејса (engl. Interafce Definition

Language, IDL). Уз то, C# подржава нови HML стил уметања документације што поједностављује израду референтне документације о програму прилагођеном Web-у.

C# подржава и *интерфејс*е – средство за склапање уговора с класом о пружању услуга предвиђених тим интерфејсом. У језику C#, класа сме да наслеђује само од једног родитеља, али може да имплементира више интерфејса. Када имплементира интерфејс, C# класа се обавезује да ће извршавати оно што интерфејс предвиђа.

C# подржава и с*труктуре,* које су се значајно измениле од C-а и C++-а. Структуре на C#- у су ограничени, не превише захтевни типови чије инстанце мање оптерећују оперативни систем и меморију од класа. Структура се не може извести из класе и обрнуто, али се помоћу структуре може имплементирати интерфејс.

C# у потпуности подржава *делегате* који омогућавају индиректно повезивање метода. У неким другим језицима, то је другачије решено (на пример, помоћу показивача не функције у C++- у), али делегати су проверени референтни типови који капсулирају методе са одређеним потписима и типовима резултата.

C# подржава компонентно оријентисане елементе, попут својстава, догађаја, и саставних делова декларација (на пример, атрибута). Компонентно оријентисано програмирање подразумева чување метаподатака у оквиру кода класе. Метаподаци описују класу, укључујући њене методе и својства, безбедносне ознаке и друге атрибуте (који, на пример, одређују да ли се класа може серијализовати); кôд садржи логику неопходну за извршавање функција класа. То значи да преведена класа садржи све информације о себи. Зато у извршном окружењу програма које зна како да чита метаподатке класе и кôд, нису потребне друге информације да би се користила та класа. C# и CLR омогућавају да корисник сам класи дода метаподатке помоћу наменских атрибута. Корисник може и да чита метаподатке о класи користећи CLR типове који подржавају рефлексију.

Када преводимо кôд, правимо *програмски склоп* (engl. assembly). То је скуп датотека које програмер види као једну датотеку с динамичким повезивањем (DLL), или као извршну датотеку (EXE). У окружењу .NET, програмски склоп је основна јединица за поновно коришћење кода, прављење више верзија, безбедност и примену. CLR подржава бројне класе за управљање програмским склоповима.

На крају, рецимо да C# подржава и следеће:

- Директан приступ меморији помоћу показивача у стилу C++-а
- Резервисане речи за издвајање несигурних операција

- Упозоравања ”скупљача смећа” окружења CLR да не уништава објекте на које показују показивачи док се ти објекти не ослободе

3.2. Декларисање и додељивање променљивих

C# је веома строг у погледу декларисања променљивих. *Променљива* је локација у меморији где се чува нека вредност. Променљиву можемо замислити као кутију у којој се чува привремена информација. Свакој променљивој у програму морамо дати јединствено име. Име променљиве се користи да би се референцирало на ведност коју она садржи.

3.2.1. Именовање променљивих

Документација радног окружења Microsoft.NET садржи неколико препорука у вези са именовањем променљивих:

- Не сме се користити подвлака.
- Не смеју се правити идентификатори који се разликују само по величини слова. На пример, не сме се за истовремено коришћење направити једна променљива која се зове *мојаПроменљива* и друга која се зове *МојаПроменљива*.
- Име променљиве почиње малим словом.
- Ако се идентификатор састоји од више речи, почињемо другу и сваку следећу реч великим словом (ово се назива *camelCase* нотација).

3.2.2. Декларисање променљивих

Као што смо већ рекли, променљиве представљају неке ”кутије” у меморији у којима могу да се чувају вредности. C# може да обрађује много различитих типова вредности као што су на пример, цели бројеви, бројеви са покретним зарезом, низови знакова итд. Када декларишемо променљиву, треба навести о којој врсти података је реч коју ће она чувати.

Тип и име променљиве декларишемо у исказу декларације. На пример, следећим исказом се декларише да променљива по имену старост садржи вредности типа инт (за целе бројеве). Наравно, ту је и тачка зарез:

```
int starost;
```

Тип променљиве, `int`, је пример кључне речи. То је име једног од примитивних C# типова - целобројног типа. Пошто декларишемо своју променљиву, можемо јој доделити вредност. Следећим исказом се променљивој старост додељује вредност 42. И у овом случају се види да је знак тачка- зарез обавезан.

```
starost = 42;
```

Симбол = је оператор додељивања који вредност са његове десне стране додељује променљивој која се налази са леве стране. Након тог додељивања, променљива старост може се у коду користити за реферисање вредности коју садржи. Следећим исказом се на конзолу исписује 42:

```
Console.WriteLine(starost);
```

3.2.3. Рад са примитивним типовима података

C# садржи низ уграђених типова које називамо *примитивним типовима података*.У следећој тебели неведени су најчешче коричћени примитивни типови података. Типови *int* и *long* служе за чување целих бројева (*long* може да прихвати веће бројеве али захтева више меморије); типови *float* и *double* служе за чување бројева са децималним зарезом (*double* може да прихвати број са прецизношћу двоструко већом од броја у променљивој типа *float*, отуда и име тог типа); тип *decimal* је предвиђен за чување новчаних вредности до 28 значајних цифара; тип *char* служи за појединачне знакове (у Unicode облику); тип *string* чува низ знакова, а тип *bool* чува вредност тачно или нетачно.

Табела 2: Примитивни типови података[1]

Тип	**.Net тип података**	**Величина у bytes**	**Величина**
byte	Byte	1	од 0 до 255
sbyte	Sbyte	1	од -128 до 127
short	Int16	2	од -32,768 до 32,767
ushort	Uint16	2	од 0 до 65,535
int(default)	Int32	4	од -2,147,483,648 до 2,147,483,647
long	Int64	8	од -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807
ulong	Ulong64	8	од 0 до

			18,446,744,073,709,551,615
float	Single	4	од ± $1.5x10^{-45}$ до ±$3.4x10^{38}$
double	Double	8	од ±$5.0x10^{-324}$ до ±$1.7x10^{308}$
bool	Boolean	1	Вредности тачно/нетачно
char	Char	2	Појединачни карактер
decimal	Decimal	12	од $1.0x10^{-28}$ до $7.9x10^{28}$

3.3. Оператори у C# језику

Операције које програм може изводити зависе од проблема који покушавате да решите. Оператори који се могу употребити за аритметичке операције су:

Табела 3: Оператори аритметичких операција[1]

Оператори	Опис
+	Сабирање
-	Одузимање
*	Множење
/	Дељење
%	Остатак од дељења, познат као модул
++	Инкрементирање(увећање)вредности за 1
--	декрементирање(смањивање)вредности за 1

Постоји велики број оператора али су најчешће у употреби следећи:

Табела 4: Најчешће коришћени оператори[1]

Оператори	Опис
==	Једнако
>=	Веће од или једнако
>	Веће од
!=	Различито, није једнако
<=	Мање од или једнако
<	Мање од

Ови оператори се користе за извођење логичких провера, односно утврђивање да ли је нешто истинито или није. Постоје и додатни оператори када је потребно да се у једној линији кода упореди више од једне вредности. А ти оператори су:

Табела 5: Додатни оператори[1]

Оператори	Опис
&	AND
|	OR
^	XOR
!	NOT
&&	Логичко I
||	Логичко ILI

3.4. Једноставна селекција помоћу If...Else

Када треба у коду да се доносе одлука, користе се искази if...else. Помоћу исказа if...else може се извршити селекција одговарајуће акције, односно ако је задовољен одређени услов извршавање програма се одвија у одређеном смеру.

Синтакса исказа *if* и *else* је следећа (*if* и *else* су кључне речи):

if (Израз)

Исказ- 1

else

Исказ- 2

Ако је Израз тачан, извршиће се Исказ- 1; иначе је Израз нетачан па се извршава Исказ- 2. Ако клаузула *else* не постоји, у случају да је Израз нетачан, не дешава се ништа.

3.5. Рад са исказима Switch...Case

Када се пише каскадни исказ *if*, понекад се види да су сви ти искази *if* веома слични; сви они испитују идентичан израз. Једина разлика је у томе што сваки *if* пореди

резултат израза са другачијом вредношћу. У таквим ситуацијама, често може каскадни исказ *if* да се претвори у исказ *switch* како би програм био ефикаснији и јаснији за читање.

Синтакса исказа *switch* је следећа (*switch*, *case* и *default* су кључне речи);

```
switch  ( контролниИзраз )
{
case  константанИзраз :
    искази
    break;
case  константанИзраз :
    искази
    break;
.....
 default :
    искази
    break;
}
```

Овде се контролниИзраз израчунава само једном и затим се извршавају искази испод лабеле *case* чији је константанИзраз једнак резултату контролног израза, све до исказа *break*. Исказ *switch* се затим завршава а програм се наставља након затворене витичасте заграде исказа *switch*. Ако ниједан константанИзраз није једнак вредности контролног израза, извршиће се искази испод опционе лабеле default.

3.5.1. Поштовање правила исказа Switch

Искази switch су веома корисни али, нажалост, не можемо их употребити увек када бисмо хтели. Искази switch морају да поштују следећа правила:

- Исказ switch може се применити само на примитивне типове података и на стрингове. Ако је потребно да се промени исказ switch на неке друге типове података, онда ћемо морати да употрбимо исказ *if*.
- Лабеле *case* морају да буду константни изрази, као што је 42 или ”42”. Ако је потребно да се за време извршавања израчуна вредност лабела *case*, мораћемо да употребимо исказ *if*.

- Лабела *case* морају да буду јединствени изрази. Другим речима, није дозвољено написати две исте лабеле *case* са истом вредношћу.
- Синтаксу лабеле *case* морамо да поновимо за сваку појединачну вредност ако нам треба да извршимо исте исказе за неколико вредности.

3.6. Коришћење исказа Do и While

Исказ *while* се користи да би се један исказ извршио више пута, све док је један Булов израз тачан. Синтакса исказа *while* је следећа:

while (буловИзраз)
исказ

Булов израз се израчунава, па ако је тачан извршава се исказ и поново се израчунава Булов израз. Ако је израз још увек тачан, исказ се понавља и поново се израчунава израз. Тај поступак се наставља све док Булов израз не постане нетачан када се извршавање исказа *while* прекида.

Исказ *while* и исказ *for* испитују Булов израз на почетку петље. То значи да се тело петље неће извршити, чак ни један једини пут, у случају да испитивање већ први пут даје резултат нетачно.Понекад ће нам затребати да се тело петље изврши бар једном. У том случају потребан нам је исказ у којем се израз испитује на крају петље, некон што се тело петље изврши. То је сврха исказа *do*. Синтакса исказа до је следећа:

do
исказ
while (буловИзраз) ;

3.7. Рад са For исказима

Када се пролази кроз неки скуп више пута и када се зна број пролазака(понављања), добро је употребити *for* исказ. Синтакса за *for* исказ изгледа овако:

For ([inicijalizatori] ; [izraz] ; [iteratori])

иницијализатори – променљива дефинисаног типа која се користи да се више пута понови неко извршавање

израз – логички израз који се користи да оконча понављања када логичка вредност израза постане неистинита

итератори – овај сегмент исказа изводи итерације над променљивом која је претходно била иницијализована тј. увећава њену вредност

3.7.1. Исказ Foreach

Када је потребно обавити понављање над скупом (или низом), користимо следећу синтаксу:

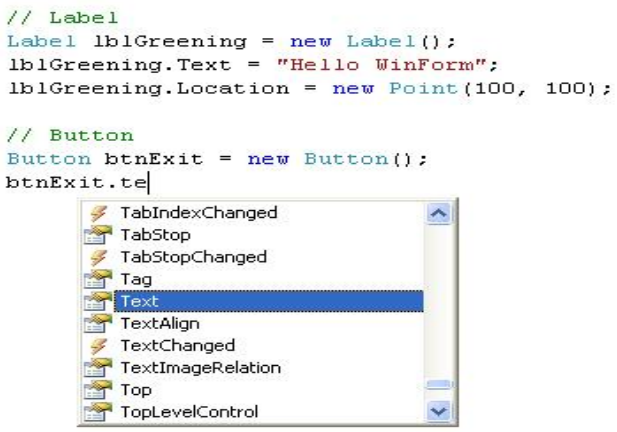
foreach (тип идентификатор ин израз)

исказ

Где је тип појединачни елемент у скупу дефинисан изразом. Идентификатор ће бити назив који ће бити додељен индивидуалном елементу који ће се користити у исказу или исказима у оквиру итерације.

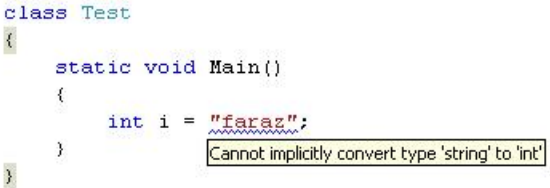
3.8. IntelliSense и HotCompiler

Као што смо рекли, Visual C# садржи едитор за писање кода. Текст едитор се учитава са IntelliSense и hot compiler. На пример, када укуцамо тачку после назива објекта аутоматски ће се појавити IntelliSense са листом свих својства, метода, итд. тог објекта чиме се олакшава куцање кода (Слика 10).



Слика 10: Функционисање IntelliSense

Hot compiler нам показује синтаксну грешку приликом куцања кода (Слика 11).



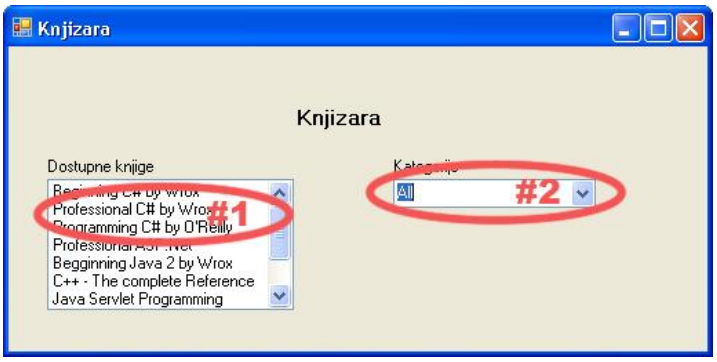
Слука 11: Функционисање Hot Compiler-а

Visual C# садржи интегрисани компајлер тј.преводилац који преводи и израђује апликацију. Можемо да преводимо појединии фајл или комплетни пројекат. Током превођења апликације могле су се десити неке грешке приликом куцања програма. Проналажење и поправљање грешака у програму назива се дебаговање (bug - грешка).

3.9. Рад са контролама

3.9.1. Контрола ListBox

Контрола ListBox може садржати листу опција и где их корисник може селектовати. ListBox садржи својство Items помоћу којих се опције постављају у објекту. У том својству опције се могу постављати и брисати.



Слика 12: ListBox(#1) i ComboBox(#2)

Контролу ListBox можемо додати на форму, контрола се налази у оквиру са алатима(ToolBox). Као што смо рекли ListBox садржи својство Items, опције можемо читати, додавати и брисати(Слика 12).

3.9.1.1. Додавање ставки (опција) у ListBox

Да би смо додали ставке потребно је прво додати контролу. Затим селектовати контролу ListBox и у оквиру Пропертиес пронаћи својство Items. Појавиће се прозор у коме је могуће додавати ставке. Други начин убацивања ставки је писањем одређеног кода у ListBox користећи методу Add(). Претпоставимо да је потребно направити листу књига. Променимо својство Name у „lbxBooks“. Код изгледа овако:

```
lbxBooks.Items.Add("Programming C#");
```

```
lbxBooks.Items.Add("Professional C#");
```

или ако желимо да унесемо више ставки онда је потребно користити методу AddRange():

```
lbxBooks.Items.AddRange(new string[] {
                        "Beginning C# by Wrox",
                        "Professional C# by Wrox",
                        "Programming C# by O'Reilly",
                        "Professional ASP.Net",
```

3.9.1.2. Промена ставки у ListBox-у

Својство Items у контроли ListBox обезбеђује и могућност промене постојеће ставке које се налази у Items помоћу следећег кода:

```
lbxBooks.Items[0] = "Program Development in Java";
string book1 = (string)  lbxBooks.Items[1];
```

Први ред кода нам мења постојећи текст у „Program Development in Java“ док други ред кода нам даје могућност само читања ставке.

3.9.1.3. Брисање ставки из ListBox-а

Ставке у ListBox-у се могу обрисати коришћењем методе Remove() или RemoveAt() на следећи начин:

```
lbxBooks.Items.Remove("Progamming C#");
```

Овим кодом брише се ставка која се налази у контроли ListBox. Може се користи и метода RemoveAt() где се брише прва ставка у контроли.

```
lbxBooks.Items.RemoveAt(0);
```

Постоји и могућност брисања свих ставки у контроли:

```
lbxBooks.Items.Clear();
```

3.9.1.4. Додавање догађаја у ListBox

Најчешће коришћен и најважнији догађај у ListBox је догађај SelectedIndexChanged. Покреће се када корисник селектује одређену ставку. Да би се догађај додао потребно је два пута кликнути контролу ListBox која се налази на форми.

```
private void lbxBooks_ SelectedIndexChanged (object sender, System.EventArgs e)
{
    MessageBox.Show("Селектована је ставка "+
lbxBooks.SelectedItem.ToString(),"Селектована опција");
}
```

3.9.2. Контрола ComboBox

Контрола ComboBox је веома слична контроли ListBox. Комбиновани оквир, падајућа листа за прегледање елемената листе. ComboBox је представљен у .NET као класа System.Windows.Forms.ComboBox. ComboBox има три облика које можемо мењати у својству „DropDownStyle“. У том својству се налазе следећи облици: Simple, DropDown, DropDownList (Слика 13).

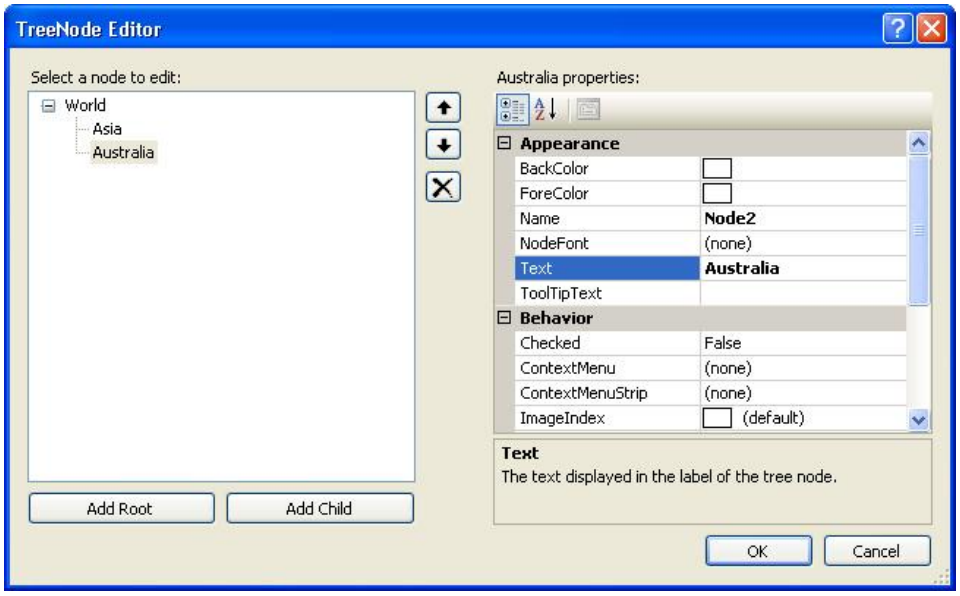
Слика 13: Облици контроле ComboBox

Стил Simple нам приказује листу ставки, овај облик контроле се користи у контроли FontDialog кога ћемо касније разјаснити. Стил DropDown представља падајућу листу за прегледавање елемената. Корисник може да дода нову ставку. Стил DropDownList је потпуно исти као и стил DropDown с том разликом што се не даје могућност исписивања нове ставке.

3.9.3. Контрола TreeView

Контрола TreeView користи се за хијерархијско представљање ставки. Представља се у .NET као класа System.Windows.Forms.TreeView. Најлакши начин додавања и брисања ставки је коришћењем својства „Nodes“. Када кликнемо својство

Nodes појавиће се прозор који се назива TreeNode Editor. Дугме Add Root додаје први хијерархијски ниво док друго дугме Add Child додаје наредни ниво (Слика 14).



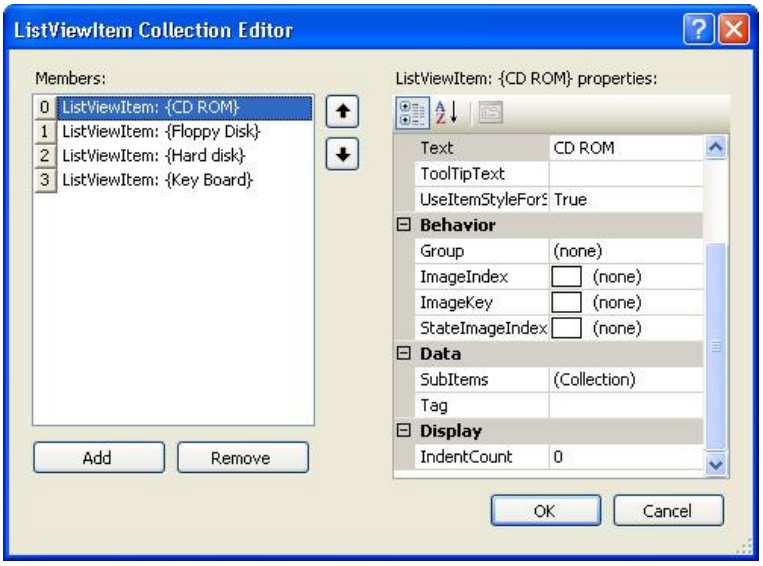
Слика 14: TreeNode Едитор

3.9.4. Контрола ImageList

ImageList је невидљива контрола. Користи се за складиштење слика које ће да користе друге контроле као што су TreeView или ListView контроле. Када се селектује контрола ImageList из оквира са алатима и приликом стављања контроле на форму неће се графички приказати да је додата већ ће се на дну прозора појавити икона. Можемо мењати њена својства тако што селектујемо икону. Најосновније својство ове контроле је „Name“. Контрола ImageList има неколико својства, својство „Images“ меморише слике. Када се кликне на својство „Images“ појавиће се прозор Images Collection Editor у коме се могу додати различите слике са екстензијом .bpm, .jpg и .gif.

3.9.5. Контрола ListView

Контрола ListView је веома важна и интересантна контрола. Користи се за приказивање листе ставки са могућношћу приказивања икона различитих величина: Large Icons, Small Icons, List и Details. Visual C# омогућава веома једноставно додавање листе у контроли ListView. За уношење ставки је потребно кликнути својство „Items“. Отвориће се прозор ListView Collection Editor (Слика 15).



Слика 15: ListView Editor

Користи се дугме Add за додавање ставки. Свака ставка има својство „Text“ које можемо мењати као и својство „ImageIndex“ у коме се налазе иконе за сваку ставку. Учитање слика се врши на следећи начин: кликне се својство „ImageIndex“ где ће се појавити падајућа листа са постојећим иконама. Свака ставка може да се обрише притиском на дугме Remove.

3.9.6. Контрола MenuStrip

Убацивање менија на форму се врши на исти начин као и све остале контроле. Контрола MenuStrip је представљен у .NET као класа System.Windows.Forms.MenuStrip. Кликом на контролу можемо променити назив менија и додавати опције.

Неколико важних тачака у вези са мену контролом:

- Можете променити назив менија и име сваке опције коришћењем оквира Properties
- Постоји могућност правити разне пречице за директно активирање појединих команди путем тастатуре
- Можете поставити линију уместо назива опције нпр: линију између опције Save и Exit
- Можемо додати догађај тако што се два пута кликне контрола MenuStrip

3.9.7. Контрола ToolStrip

Убацивање контроле на форму се врши на исти начин као и све остале контроле. Класа ToolStrip садржи колекцију дугмади који се могу додавати/брисати. Када се кликне на својство Buttons у оквиру Properties појављује се прозор за додавање или брисање дугмади. Можемо додати догађај тако што се два пута кликне контрола.

3.9.8. Стандардни оквири за дијалог

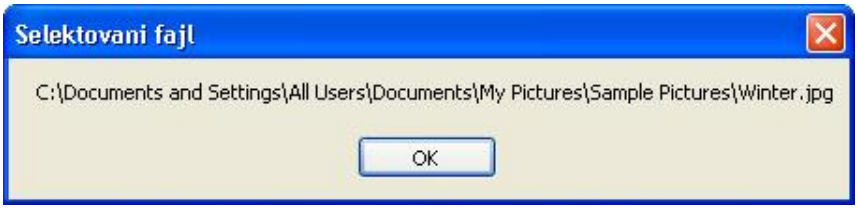
Под стандардним оквирима за дијалог се обично подразумевају следећи оквири: OpenFileDialog, SaveFileDialog, FontDialog, ColorDialog.

3.9.8.1. Контрола OpenFileDialog

Овај оквир за дијалог се најчешће користи у Windows окружењу који служи за отварање фајлова. Рад са контролом ћемо илустровати на следећи начин поставићемо на форми дугме. Притиском на њега отвориће се оквир за диалог, након тога где ће се приказати порука MessageBox. Код изгледа овако:

```
private void btnOpenFile_Click(object sender, System.EventArgs e)
{
    DialogResylt res = openFileDialog.ShowDialog();
    if (res == DialogResylt.OK)
    {
        MessageBox.Show(openFileDialog.FileName, "Selektovan fajl");
    }
}
```

За приказивање оквира OpenFileDialog користили смо метод ShowDialog(). Овај метод приказује оквир са поруком где се притиском на дугме ОК затвара дијалог. Када покренемо овај програм добићемо следећи изглед:



3.9.8.2. Контрола SaveFileDialog

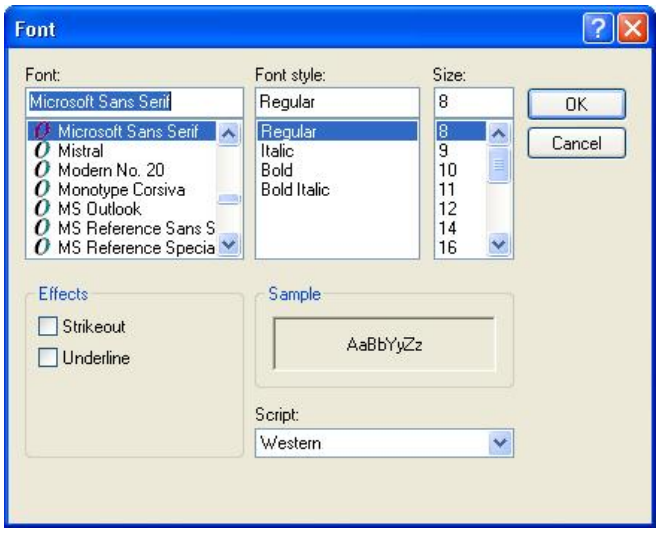
Контрола SaveFileDialog има исто својство као и контрола OpenFileDialog. Овај оквир за дијалог омогућава кориснику да меморише одређени фајл на одређеном месту. Следећи код нам приказује оквир SaveFile и порука са именом фајла:

```
private void btnSaveFile_Click(object sender, System.EventArgs e)
{
    DialogResylt res = saveFileDialog.ShowDialog();
    if (res == DialogResylt.OK)
        {
            Messagebox.Show(saveFileDialog.FileName, "Memorisan fajl");
        }
}
```

Добијамо следећи изглед:

3.9.8.3. Контроле FontDialog и ColorDialog

Контрола FontDialog, као и сам оквир за дијалог, увелико је сложенији од осталих контрола. Разлог је чињеница да постоји више опција за контролу. Контрола FontDialog омогућава кориснику да одабере фонт, стил и величину слова. Контрола ColorDialog даје могућност кориснику да селектује одређену боју (Слика 16).



Слика 16: FontDIalog

IV. ADO.NET

4.1. Увод у ADO.NET

Када, при изради неке једноставне апликације користимо Visual Studio.NET за успостављање везе са подацима и за рад са подацима, ми заправо незнамо шта је тада Visual Studio.NET направио у позадини. У позадини свих тих превлачења, VS је користио ADO.NET објекте за приступ и приказивање података.

У скоро свим апликацијама потребно је да се на неки начин приступа подацима. До скоро су програмери користили стандардне инфраструктуре за приступ подацима, као што су ODBC, OLEDB и ADO. Увођењем .NET окружења, Microsoft је направио нов начин за рад са подацима који се назива ADO.NET.

ADO.NET представља потпуно нову методологију за приступ подацима - нову технологију, нови дизајн, а направљен је од нуле. Пре свега треба разјаснити следеће – ADO.NET није скраћеница за ActiveX Data Objects.NET. Зашто? Зато што:

- ADO.NET се налази *унутар* .NET окружења и не представља спољни ентитет.
- ADO.NET *није* колекција ActiveX компоненти.

Име ADO.NET направљено је по аналогији са именом ADO из прилично очигледног разлога: пошто је Microsoft осмислио нову технологију на сличан начин као ADO технологију (када је у питању функционалност), а пошто су лакоћа коришћења и основни најнижи ниво функционалности исти, Microsoft је желео да програмери лако користе ADO.NET и да при томе не мисле како морају да „уче све из почетка”.

Пошто су изнете спецификације за .NET окружење, одмах се видело да се ADO неће уклопити у то. ADO је био приступачан за апликације за рад са подацима као спољни пакет, а следио га је модел који није компатибилан са .NET окружењем – COM интерфејс (спољна компонента коју може да користи било која апликација), који је захтевао да апликација која покушава да користи његове сервисе експлицитно укључи референцу на COM интерфејс. Насупрот томе, .NET апликације су осмишљене тако да деле један модел у коме су све библиотеке, методи и класе интегрисани у само једно окружење, организовани у логичке просторе имена и јавно декларисани за све апликације које желе да их користе. То је прагматичан приступ који се сматра лако

прилагодљивим, па је одлучено да технологија приступа подацима у .NET окружењу треба да има нов облик и названа је ADO.NET.

ADO.NET је осмишљен за прилагођавање повезаном и раздвојеном свету Интернета – свету са којим ADO није могао да ради. Такође је ADO.NET осмишљен тако да покуша да обухвати широко прихваћени XML стандард, много више него што је случај код ADO-а, с обзиром на то да је занимање за XML нагло порасло тек пошто је направљен ADO.

ADO.NET ради и у повезаном и у раздвојеном окружењу. Можемо да се повежемо на базу података и да останемо повезани док само читамо податке, а затим да затворимо везу, што је сличан процес као у оквиру ADO-а. ADO.NET се појављује у пуном сјају у раздвојеном свету. Ако треба да изменимо или допунимо податке у бази података, одржавање отворене везе на начин на који то радимо у оквиру ADO-а има велику цену, с обзиром на то да сервер мора да одржава све везе. ADO.NET решава тај проблем тако што пребацује ту страну приступа подацима у раздвојени модел. Подаци које желимо да изменимо или допунимо шаљу се од сервера ка клијенту и локално кеширају на рачунару клијента. Можемо изменити податке и када будемо спремни за ажурирање базе података, да пошаљемо податке назад серверу где се управља свим конфликтима.

4.2. Преглед грађе ADO.NET -а

ADO.NET модел објекта састоји се из две основне компоненте: скупа података (engl. DataSet) који није повезан са изворoм података и који не захтева познавање порекла података које садржи и .NET добављача података (engl. *.NET data provider*). Наиме, .NET добављачи података нам омогућавају да се повежемо са изворoм података и да извршимо CQЛ команду.

4.3. .NET добављачи података (Data Provider)

Познато нам је да добављачи података представљају скуп објеката пројектованих за комуникацију са одређеним изворoм података и апстрактно добављање података, сем ако база у позадини није SQL Server, а у том случају подаци се добављају на специфичан начин.

За разлику од претходних OLEDB и ADO добављача, сваки .NET добављач података обезбеђује одређену архитектуру и рутине за приступ подацима, што значи да се помало међусобно разликују. И поред врло малих разлика, сви .NET добављачи података су веома слични када су у питању функционалност, резултат и позиви метода. То је сјајна карактеристика ADO.NET-а, пошто лако можемо да се пребацимо са једног добављача на други уз веома мале измене у коду.

Постоје три .NET добављача података: за SQL сервер, OLEDB изворе података и за ODBC изворе података. Сваки добављач постоји у именованом простору у оквиру именованог простора System.Data и састоји се од одређеног броја класа.

Табела 6: Листа .NET добављача који су нам на располагању[6]

System.Data.SqlClient	Добављач података за SQL Server 7.0 и новије верзије.
System.Data.OleDb	Добављач података за изворе података који су компатибилни са OLEDB- ом као што су, на пример, Microsoft Access и раније верзије SQL Servera.
System.Data.Odbc	Користи се за директан приступ Microsoft Open Database Connectivity (ODBC) управљачким програмима. Овај процтор имена не испоручује се иницијално са .NET окружењем, али се може преузети са Microsoft-ове Веб локације

4.4. Компоненте добављача података

Сваки .NET добављач података састоји се од четири главне компоненте:

- *Connection* – користи се за повезивање са извором података.
- *Command* – користи се за извршавање команде у вези са извором података и за издвајање објеката DataReader или DataSet, као и за извршавање команди *Insert*, *Update* или *Delete*.
- *DataReader* – повезани скуп резултата само за прослеђивање и читање.
- *DataAdapter* - користи се за попуњавање скупа DataSet са подацима из извора података, као и за ажурирање извора података.

Ове компоненте су засебно имплементиране помоћу .NET добављача, не постоји класа Connection. Уместо тога, SQL Server и OLE DB добављачи имплементирају класе SqlConnection и OledbConnection. Поменуте класе директно потичу из System.ComponentModel.Component, јер не постоји апстрактна класа Connection, већ се имплементира исти интерфејс IDbConnect (у именованом простору System.Data).

4.4.1. Класе Connection

Класе повезивања (engl. connection classes) веома су сличне објекту ADO Connection и користе се за представљање везе са специфичним извором података. Класе повезивања складиште информације које су потребне ADO.NET-у за повезивање са извором података у облику познатог низа повезивања (као и у програму ADO). Својство ConnectionString интерфејса IDbConnection садржи информације попут корисничког имена и лозинке корисника, назив и локацију извора података са којим се повезује итд. Класе повезивања такође садрже методе за отварање и затварање веза, као и за почетак трансакција и, на крају, својства за подешавања периода за временску контролу везе и враћања везе у првобитно стање (отворено или затворено).

Табела 7: Значајна својства SqlConnection[1]

Својство	Значење	Подразумевана вредност
Connection-String	Знаковни низ који се користи за повезивање са извором података приликом извршавања метода open.	Празно
Connection-TimeOut	Максимално време које ће објекат Connection покушавати да успостави конекцију пре него што испали изузетак.	15 секунди
Database	Име базе података која ће се отворити пошто се успостави конекција.	Празно
DataSource	Локација и датотека која садржи базу података.	Празно
PacketSize	Величина мрежних пакета који се користе за комуникацију са SQl сервером.	8192 бајта
ServerVersion	Верзија Сервера, који обезбеђује OLE DB снабдевача података.	Празно
State	Вредност типа ConnectionState која означава тренутно стање конекције.	Затворено

Две верзије објекта Connection излажу нешто другачији скуп својстава, објекат SqlDbConnection нема својство Provider, а објекат OleDbConnection нема својство PacketSize или WorkStationID.

4.4.1.1. Својство Connection String

ConnectionString је најважније својство објекта Connection. У ствари, остала својства су само за читање и постављају се на основу вредности својстава ConnectionString.

Сва својства ConnectionString имају исти формат. Састоје се од скупа кључних речи и вредности, при чему су парови раздвојени знаковима тачка зарез, а читаво својство се ограничава или једноструким или двоструким наводницима. Имена кључних речи не зависе од величине слова, али вредности могу да зависе у зависности од извора података. Одређивање садржаја ConnectionString-а може бити мало проблематичан из разлога што је он увек јединствен у односу на снабдевача података.

Својство ConnectionString може се поставити само док је конекција затворена. Када се својство постави, објекат Connection ће проверити синтаксу знаковног низа, а затим поставити и остала својства. Ваљаност својства ConnectionString проверава се у целости приликом отварања конекције. Ако објекат Connection открије неко погрешно или неподрживо својство, испалиће изузетак (или OleDbException или SqlDbException, у зависности од објекта који се користи).

4.4.1.2. Методи објекта Connection

И објекат SqlConnection и објекат OleDbConnection излажу исти скуп метода:

Табела 8: Методи објекта Connection[1]

Метод	Опис
BeginTransaction	Отпочиње Трансакцију на бази података.
ChangeDataBase	Мења тренутну базу података на отвореној конекцији.
Close	Затвара конекцију са извором података.
CreateCommand	Ствара и враћа објекат Data Command који је придружен конекцији.
Open	Установљава конекцију са извором података.

Методи објекта Connection који се најчешће користе су *Open* и *Close*, који отварају и затварају конекцију. Метод BeginTranscation почиње обраду трансакције на конекцији. Метод CreateCommand може се користити за прављење ADO.NET објекта Data Command.

4.4.1.3. Отварање и затварање конекције

Методе Open и Close аутоматски позивају два објекта који користе објекат Connection, DataAdapter и DataCommand. Такође се по потреби могу експлицитно позивати из кода.

Ако се на конекцији метод Open позове из објекта DataAdapter или објекта Data Command, они ће конекцију оставити у стању у којем је била пре позива метода. Ако је на пример, конекција била отворена пре позива метода DataCommand.Fill,остаће отворена и када се операција Fill заврши. Са друге стране, ако је конекција била затворена пре позива методе Fill, DataAdapter ће је затворити када заврши са радом.

Ако се метод Open позове експлицитно, конекција са извором података ће остати отворена све док се експлицитно и не затвори. Неће бити затворена аутоматски, чак и ако објекат Connection изађе из свог опсега важења.

4.4.1.4. Руковање догађајима на конекцији

Објекат Connection и за OLE DB и за SQL сервер обезбеђује два догађаја: StateChange и InfoMessage.

Догађај StateChange се јавља сваки пут када се промени стање објекта Connection. Догађај свом руководиоцу прослеђује аргументе StateChange - EventArgs, који поседује два својства: OriginalState и CurrentState.

Табела 9: Догађаји на конекцији[1]

Стање	Значење
Broken	Конекција је отворена, али није функционална. Може се затворити и поново отворити.
Closed	Конекција је затворена.
Connecting	Конекција је у поступку успостављања везе, али још увек није отворена.

Executing	Конекција извршава команду.
Fetching	Конекција враћа податке.
Open	Конекција је отворена.

4.4.2. Класе Command

Класе команди излажу интерфејс IDbCommand и сличне су ADO објекту Command. Употребљавају се за извршавање SQL исказа или сачуваних процедура у извору података. Класе команди, као и ADO објекат Command, такође поседују својство CommandText које садржи текст команде коју је потребно извршити у односу на извор података, као и својство CommandType које указује на то да ли је команда SQL исказ, назив сачуване процедуре или назив табеле. Постоје три засебна извршна метода – ExecuteReader који враћа објекат DataReader,затим метод ExecuteScalar који враћа одређену вредност и, на крају, метод ExecuteNonQuery који се употребљава када се подаци не враћају као одговор упиту (на пример, за исказ SQL UPDATE)

Команде за податке поседују четири различите верзије конструктора:

Табела 10: Верзије конструктора[1]

Својство	Опис
New()	Прави нови, подразумевани примерак команде за податке.
New (Command)	Првави нову команду за податке у којој је својству CommandText додељен знаковни низ који се налази у параметру Command.
New (Command, Connection)	Прави нову команду за податке у којој је својству CommandText додељена вредност која се налази у параметру Command, а својству Connection додељена вредност која се налази у параметру Connection
New (Command, Connection, Transaction)	Прави нову команду за податке у којој је својству CommandText додељена вредност која се налази у параметру Command, својству Connection додељена вредност која се налази у параметру Connection, а у својству Transaction додељена вредност која се налази у параметру Transaction

4.4.2.1. Својства објекта Command

Својства која поседује команда за податке:

Табела 11: Својства објекта Command[1]

Својство	Опис
CommandText	Sql нарадба или ускладиштена процедура која треба да се изврши.
CommandTimeout	Време (у секундама) за чекање на одговор од извора података.
CommandType	Означава како би својство CommandText требало да се протумачи. Подразумева се вредност text.
Connection	Објекат Connection на којем команда за податке треба да се изврши.
Transaction	Трансакција у којој ће се команда извршити.
UpdateRowSource	Одређује начин на који се резултати примењују на објекту DataRow када се објекат Command користи у методу упдате објекта DataAdapter.

Својства CommandText, које је знаковни низ, садржи или стварни текст команде која треба да се изврши на конекцији или име ускладиштене процедуре из извора података.

Својство CommandTimeout одређује време које ће команда чекати на одговор од сервера пре него што генерише грешку. Ово је време које протекне пре него што објекат Command почне да прима резултате, а не време које је потребно команди да се изврши. Извору података може бити потребно 10 или 15 минута да врати све редове неке огромне табеле, али под условом да је први ред враћен у току задатог периода CommandTimeout неће се генерисати никаква грешка.

Својство CommandType говори објекту Command на који начин треба да протумачи садржај својства CommandText. Могуће вредности су:

Табела 12: Могуће вредности CommandText-а[1]

Својство	Опис
StoredProcedure	Име ускладиштене процедуре.
TableDirect	Име табеле.
Text	Текстуална SQL наредба.

Својство Connection садржи референцу на објекат Connection на којем ће се команда извршити. Објекат Connection мора припадати истом простору имена као и објекат Command, односно, објекат SqlCommand мора да садржи референцу на објекат SqlConnection, а објекат OleDbCommand мора да садржи референцу на објекат OleDbConnection

Својство Parameters објекта Command садржи колекцију параметара за SQL наредбу или ускладиштену процедуру наведену у својству CommandText.

Својство Transcation садржи референцу на објекат Transcation и служи да упише команду у ту трансакцију.

Својсво UpdateRowSource одређује начин на који се резултати примењују на објекат DataRow када се објекат Command изврши у мотоду *Update* објекта DataAdapter. Могуће вредности својства UpdateRowSource су:

Табела 13: Вредности својства UpdateRowSource[1]

Својство	Опис
Both	И излазни параметри и први ред који је објекат Command вратио биће пресликани на измењени ред.
FirstReturnedRecord	Први ред који је објекат Command вратио биће пресликани на измењени ред.
None	Сви враћени параметри или редови се одбацују.
OutputParameters	Излазни параметри објекта Command биће пресликани на измењени ред.

Када је команду за податке аутоматски генерисао Visual Studio, тада је None подразумевана вредност својства UpdateRowSource. Када се објекат Command генерише у време извршавања или га корисник направи у време дизајнирања, подразумевана вредност је Both.

4.4.2.2. Методи објекта Command

Методи које поседује објекат Command:

Табела 14: Методи објекта Command[1]

Метод	Опис
Cancel	Отказује извршавање команде за податке.

CreateParameter	Прави нови параметар.
ExecuteNonQuery	Извршава команду на објекту Connection и враћа број измењених редова.
ExecuteReader	Шаље вредност својства CommandText објекту Connection и прави објекат DataReader.
ExecuterScalar	Извршава упит иврађа прву колону првог реда из скупа резултата
ExecuteXmlReader	Шаље вредност својства CommandText објекту connection и прави објекат XMLReader.
Prepaer	Прави припремљену (компајлирану) верзију команде на извору података.
ResetCommandTimeout	Својство CommandTimeout Поново поставља подразумевану вредност.

Најважнији методи су методи Execute: *ExecuteNonQyery, ExecuteReader, ExecuteScalar, ExecuteXmlReader.*

ExecuteNonQuery се користи у случајевима када SQL наредба или ускладиштена процедура која треба да се изврши не враћа ниједан ред.

ExecuteScalar се користи за SQL наредбе и ускладиштене процедуре које враћају појединачну вредност.

Метод *ExecuteReader* користи се за SQL наредбе и ускладиштене процедуре кје враћају више редова. Метод ствара објекат DataReader. Метод *ExecuteReader* се може извршити без параметара, или му се може проследити вредност CommandBehavior која омогућава да се прецизно контролише начин на који ће се команда извршити. Вредности команде CommandBehavior су:

Табела 15: Вредности CommandBehavior[1]

Метод	Опис
CloseConnection	Затвара вредност придруженог објекта Connection када се затвори објекат DataReader..
KeyInfo	Означава да упит враћа колону и податке о примарном кључу.
SchemaOnly	Враћа само шему базе података, без утицаја на редове у извору података.
SequentialAccess	Резултатима сваке колоне и сваког реда приступаће се секвенцијално.
SingleResult	Враћа само појединачну вредност.

Већина вредности и параметара CommandBehavior сами себе објашњавају. KeyInfo и SchemaOnly су корисни у случајевима када није могуће одредити структуру скупа резултата команде пре њеног извршавања.

Понашање SequentialAccess омогућава апликацији да чита колоне које садрже велике бинарне вредности коришћењем метода *GetBytes* или *GetChars* објекта DataReader, док понашање SingleResult и SingleRow може оптимизовати само снабдевач података.

4.4.3. Објекат DataReader

Компонента DataReader представља одговор ADO.NET -а на повезани скуп записа (engl. recordset) у програму ADO. Међутим, DataReader је само за прослеђивање и читање (engl. forward-only, read-only) – не постоји могућност његове употребе за ажурирање извора података. Самим тим нам је омогућен екстремно брз приступ подацима које желимо да поновимо само једном, те је стога препоручљива употреба објекта DataReader (уместо објекта DataSet) где год је то могуће. Наиме, објекат DataReader може бити враћен само из позива методу ExecuteReader одређеног командног објекта; не постоји могућност директног формирања објекта. Управо због овога је потребно експлицитно формирање командног објекта, за разлику од принципа који важи у програму ADO, где је могуће позвати објекат RecordSet без експлицитног стварања објекта Command. Самим тим, ADO.NET модел објекта је транспарентнији од линеарне хијерархије програма ADO.

4.4.3.1. Својства објекта DataReader

Својства објекта DataReader

Табела 16: Својства објекта DataReader[1]

Сбојство	Опис
Depth	Дубина угнежђавања за текућу реду у хијерархијским скуповима резултата. SQL Сервер увек враћа вредност нула.
FieldCount	Број колона у текућем реду.
IsClosed	Означава да ли је објекат DataReader затворен.

Item	Вредност колоне.
RecordAffected	Број редова који су измењени, додати или обрисани.

Својство Item подржава две верзије: Item (Name) која као параметар узима знаковни низ који одређује име колоне и Item (Index) која као параметар има вредност типа Int32 која служи као индекс у колекцији колона.

4.4.3.2. Методи објекта DataReader

Методи објекта DataReader

Табела 17: Методи објекта DataReader[1]

Метод	Опис
Close	Затвара објекат DataReader.
GetType	Враћа вредност специфициране колоне која је једнака типу колоне.
GetDataTypeName	Враћа име типа извора података.
GetName	Враћа име специфициране колоне.
GetSchemaTable	Враћа објекат DataTable који описује структуру објекта DataReader.
GetValue	Враћа вредност специфициране колоне као њен природни тип података.
GetValues	Враћа све колоне из текућег реда.
NextResylt	Помера објекат DataReader. на следећи резултат.

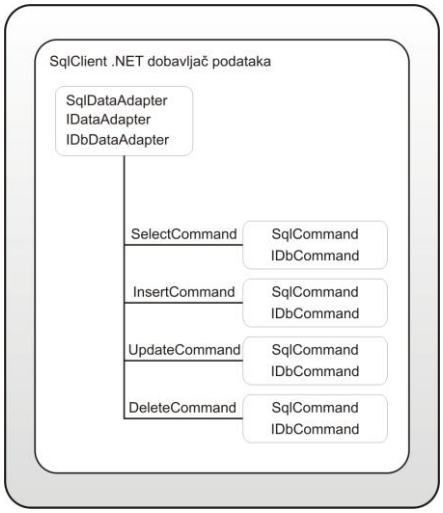
Метод *Read* враћа следећи ред из скупа резултата. Када се објекат DateReader први пут отвори, он се поставља на почетак датотеке, пре првог реда, а не на први ред. Потребно је позвати метод *Read* пре него што се врати први ред скупа резултата.

Метод *NextResult* се користи када SQL наредба или ускладиштена процедура враћају више скупова резултата. Он поставља објекат DataREader на почетак следећег скупа резултата. И у овом случају објекат DataReader је постављен пре првог реда и потребно је позвати методу *Read* пре него што се приступи неким резултатима.

Метод *GetValues* враћа све колоне из текућег реда као низ објеката, док метод *GetValue* враћа појединачну вредност чији је тип једнак неком од типова података из . .NET Framework-a.

4.4.4. Objekat DataAdapter

Последња компонента .NET добављача података је DataAdapter. Објекат DataAdapter понаша се као мост између неповезаног објекта DataSet и извора података. Захваљујући томе ова компонента открива два интерфејса: IDataAdapter који дефинише методе за популарисање објекта DataSet са подацима из извора података, као и за ажурирање извора података са променама објекта DataSet које су обављене у клијенту. Други поменути интерфејс, IDataAdapter дефинише четири својства типа IDbCommand. Свако од поменутих својстава подешава или враћа командни објекат који спецификује коју је команду потребно извршити када се постави упит извору података или када се исти ажурира.



У случају да се покуша ажурирати извор података, а да притом исправна команда није специфицирана, доћиће до генерисања грешке. На пример, ако се покуша позвати Update или DataSet тамо где је додат нови ред, а да претходно нисте специфицирали командни објекат InsertCommand за објекат DataAdapter, појавиће се следећа порука о грешци: *Unhandled Exception: System.InvalidOperationException: Update requires a valid InsertCommand when passed DataRow collection with new rows.*

.

4.4.4.1. Својства објекта Data Adapter

Својства објекта DataAdapter:

Табела 18: Својства објекта DataAdapter[1]

Својство	Опис
AcceptChangesDuringFill	Одређује да ли се на објекту DataRow позива метод AcceptChanges када се дода у објекат DataTable.
DeleteCommand	Команда која се користи за брисање редова из извора података.
InsertCommand	Команда која се користи за уметање редова у извор података.
MissingMappingAction	Одређује радњу која ће се обавити када се долазећи подаци не могу упарити са неком постојећом табелом.
MissingSchemaAction	Одређује радњу која ће се обавити када се долазећи

	подаци не могу упарити са шемом неког постојећег објекта DataSet.
SelectCommand	Команда која се користи за бирање редова из извора података.
TableMappings	Колекција објеката DataTableMappings који одређује релацију између колона из објекта DataSet и извора података.
UpdateCommand	Команда која се користи за ажурирање редова у извору података.

4.4.4.2. Команде објекта DataAdapter

Сваки објекат DataAdapter садржи референцу на четири објекта Command, од којих има својство CommandText који садржи SQL наредбу која ће се извршити.

Ако се објекат DataAdapter направи коришћењем чаробњака DataAdapter Configuration Wizard или превлачењем табеле, погледа или ускладиштене процедуре из Server Explorer-a, Visual Studio ће за сваку наредбу покушати да аутоматски генерише својство CommandText. SQL наредба се може изменити у прозору Properties.

Потребно је спецификовати својство CommandText једино за објекат SelectCommand док .NET Framework може да генерише наредбу за ажурирање, уметање и брисање.

За генерисање команди Visual Studio интерно користи објекат CommandBuilder. По потреби и у коду се може направити примерак објекта CommandBuilder и он се може користити за генерисање команди.

4.4.4.3. Колекција TableMappings

Објекат DataSet незна одакле потичу подаци које садржи, а објекат Connection не зна шта се догађа са подацима које враћа. Објекат DataAdapter одржава везу између ова два објекта. При томе користи колекцију TableMappings.

На највишем нивоу колекција TableMappings. садржи један или више објеката DataTableMapping.. Обично постоји само један објекат DataTableMapping, јер већина објеката DataAdapter враћа само један скуп записа. Међутим ако објекат DataAdapter

рукује са више скупова записа, као на пример у случају ускладиштене процедуре која враћа више скупова резултата, постојаће по један објекат DataTableMapping за сваки скуп записа.

Објекат DataTableMapping је још једна колекција која садржи један или више објеката DataColumnMapping. Објекат DataColumnMapping садржи два својства: SourceColumn, који представља име колоне у коме се разликују мала и велика слова у извору података и DataSetColumn који представља име колоне у коме се не прави разлика између малих и великих слова у објекту DataSet. Постоји по један објекат DataColumnMapping за сваку колону којом рукује објекат DataAdapter.

.NET Framework подразумевано прави колекцију TableMappings (и све објекте које садржи) у којој је име својства DataSetColumn исто као и име својства SourceColumn.

4.4.4.4. Методи објекта DataAdapter

Објекат DataAdapter подржава два важна метода: *Fill* који податке из извора падатака учитава у објекат DataSet и *Update*,, који податке преноси у обрнутом смеру – учитавајући их из објекта DataSet у извор података.

Метод *Fill* учитава податке из извора података у једну или више табела објекта DataSet коришћењем команде специфициране у својству SelectCommand објекта DataAdapter. Објекат DbDataAdapter, из којег су изведени објекти OleDbDataAdapter и SqlDbDataAdapter подржава неколико различитих верзија метода *Fill*

Табела 19: Методи објекта DataAdapter[1]

Метод	Опис
Fill(DataSet)	Прави објекат DataTable са именом Table и попуњава га редовима враћеним из извора података.
Fill(DataTable)	Попуњава спецификовани објекат DataTable редовима који су враћени извора података.
Fill(DataSet,tableName)	Попуњава објекат DataTable чије име је дато у знаковном низу табле Name и који се налази у спецификованом објекту DataSet редовима враћеним из извора података.
Fill(DataTable, DataReader)	Попуњава објекат DataTable коришћењем спецификованог објекта DataReader(Пошто је објекат DataReader декларисан као IdataReader, може се користити и OleDbDataReader или SqlDataReader).

Fill(DataTable, command, CommandBehavior)	Попуњава објекат DataTable коришћењем SQL знаковног низа прослеђеног у параметру цомманд и специфікованог параметра CommandBehavior.
Fill(DataSet, startRecord, maxRecords, tableName)	Попуњава објекат DataTable чије име је дато у знаковном низу tableName почевши од записа број startRecord (у односу на нулу) и настављајући са укупно maxRecords записа или до краја скупа резултата.

Метода Update:

Објекат DataSet не задржава никакво знање о извору података које садржи и измене које се праве у објекту DateSet не преносе се аутоматски у извор података. Да би се измене пренеле у извор података користи се метод *Update* објекта DataAdapter. Метод *Update* по потреби позива команде InsertCommand, DeleteCommand и UpdateCommand објекта DataAdapter, за сваки ред у објекту DataSet који је измењен. Класа DataAdapter подржава неколико верзија метода *Update*:

Табела 20: Верзије методе Update[1]

Метод	Опис
Update(DataSet)	Ажурира извор података из објекта DataTable са називом табеле коај се налази у специфікованом објекту DataSet.
Update(dataRows)	Ажурира извор података коришћењем специфікованог низа редова података dataRows.
Update(DataTable)	Ажурира извор података коришћењем специфікованог објекта DataTable.
Update(dataRows, DataTableMapping)	Ажурира извор података коришћењем специфікованог низа редова података dataRows и специфікованог мапирања DataTableMapping.
Update (DataSet, sourceTable)	Ажурира извор података коришћењем објекта v специфікованог параматром sourceTable који се налази у специфікованом објекту DataSet.

4.4.4.5. Догађаји објекта DataAdapter

Осим догађаја који су узроковани грешкама, објекат DataAdapter подржава само два догађаја: OnRowUpdating и OnRowUpdate. Ова два догађаја се јављају на обе

стране приликом стварног ажурирања скупа података и тако обезбеђују фину контролу поступка.

Догађај OnRowUpdating

Догађај OnRowUpdating јавља се одмах после тренутка када метод *Update* постави вредност параметра команде која треба да се изврши, али пре самог извршавања команде. Руковалац догађаја за овај догађај прима аргумент чија својства обезбеђују суштинске информације о команди која треба да се изврши.

Класа аргумента догађај дефинисана је снабдевачем података, тако да ће под условом да се користи један од снабдевача података из .NET Framework-а, бити или OleDbRowUpdatingEventsArgs или SqlRowUpdatingEventsArgs.

Својства објекта RowUpdatingEventsArgs су:

Табела 21: Својства објекта RowUpdatingEventsArgs[1]

Својство	Опис
Command	Команда за податке која треба да се изврши.
Errors	Грешка коју генерише .NET снабдевач података.
Row	Објекат DataReader који треба да се ажурира.
StatementType	Врста команде која треба да се изврши. Могуће вредности су *Select, Insert, Delete* и *Update*.
Status	Својство UpdateStatus објекта Command.
tableMapping	Објекат DataTableMapping који се користи за ажурирање.

Својство Command садржи стварну референцу на стварни објекат Command који ће се користити за ажурирање извора података.

Својство StatementType аргумената догађаја дефинише акцију која ће се извршити. Ово својство је пребројиви тип који може имати вредност *Select, Insert, Delete* или *Update*. Својство StatementType може се само читати, тако да се не може користити за измену врсте акције која ће се извршити.

Својство Row садржи референцу на објекат DataRow која ће се проследити у извор података, ако се такође може само читати, док својство TableMapping садржи референцу на објекат DataTableMapping који се користи за ажурирање.

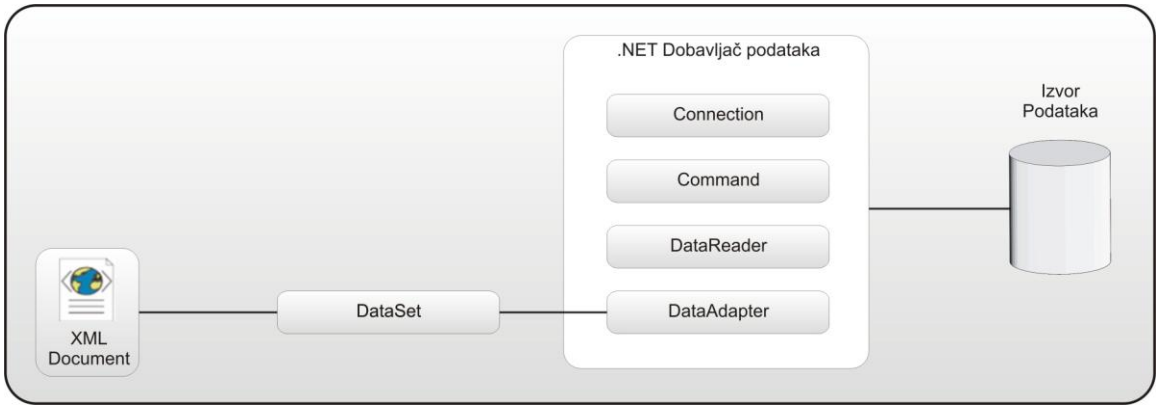
Када се руковалац догађаја први пут позове, својство Status, које је пребројиви тип UpdateStatus, дефинише статус догађаја. Ако статус једнак ErrorsOccured, својство Errors ће садржати колекцију грешака Errors.

Својство Status може се поставити у руковаоцу догађаја и тако одредити коју акцију би систем требало да предузме. Поред вредности ErrorsOccured, која проузрукује испаљивање изузетака, могуће вредности за статус изласка су *Continue*, *SkipAllRemainingRows* и *SkipCurrentRow*. Вредност *Continue*, која је подразумевана, налаже систему да настави обраду. *SkipAllRemainingRows* уствари одбацује измене које су направљене на текућем реду, као и измене направљене на свим преосталим необрађеним редовима, док *SkipCurrentRow* одбацује само обраду текућег реда.

Догађај OnRowUpdate се јавља када метод *Update* изврши одговарајућу команду на извору података. У зависности од снабдевача података, руковаоцу догађаја за овај догађај прослеђује се или аргумент *SqlRowUpdatedeventArgs* или *OleDbRowUpdatedEventArgs*. У оба случаја аргумент догађаја садржи сва својства, као и аргумент *rowUpdatingEvent*, плус једно додатно својство; аргумент RecordAffected који се може само читати и који означава број редова који су били измењени, уметнути или обрисани SQL командом која је извршена.

4.5. Објекат DataSet

Још једна у низу важних компоненти ADO.NET-а јесте DataSet (Слика 17). Поменута компонента у суштини одговара скупу записа (engl. recordset) програма ADO, али се ипак разликује на два веома битна начина. Наиме, објекат DataSet је увек искључен, а последица тога је да не проверава одакле долазе подаци. Објекат DataSet може се користити на потпуно исти начин за манипулисање подацима из традиционалног извора података или из XML документа. Да би се повозао објекат DataSet са извором података, потребно је да се употреби објекат DataAdapter као посредника између DataSet и .NET добављача података.



Слика 17: Приказ компоненти ADO.NET-а[6]

По отварању везе, постоје три поступка у процесу популарисања објекта DataSet:

- Формирање новог примерка објекта DataSet. Непосредно пре попуњавања објекта DataSet, потребно је да се спецификују информација о вези и подаци које желимо да употребимо за попуњавање објекта. Постоји много начина на које се поменути процес може обавити и један од најједноставнијих јесте да се проследи текст команде за SQL упит заједно са низом везе или отвореном везом у DataAdapter конструктор.
- Стварање новог објекта DataSet.
- Позивање DataAdapter методе Fill. Заправо, овде је потребно проследити објекат DataSet који желимо да популаришемо као параметар за поменуту методу, као и назив табеле у оквиру објекта DataSet који желимо да попунимо. Ако се позове метода Fill насупрот затворене везе, веза ће се аутоматски отворити, а затим поново затворити када се објекат DataSet попуни.

ADO.NET подржава две различите врсте објекта DataSet: Оне који су типизирани и оне који су нетипизирани. У погледу архитектуре, нетипизирани објеката DataSet је непосредан примерак објекта *System.Data.DataSet*, док је типизирани објекат DataSet класа која је изведена из класе *System.Data.DataSet*.

У погледу функционалности, типизирани објекат DataSet своје табеле и њихове колоне излаже као својства објеката. Овим се рад објекта DataSet синтаксно веома олакшава јер се табеле и колоне могу директно референцирати коришћењем њихових имена. Типизирани објекат DataSet обезбеђује једну важну предност: он омогућава да се за време превођења проверава тип за вредност података.

4.5.1. Својства објекта DataSet

Својства објекта DataSet:

Табела 22: Својства DataSet-а[1]

Својство	Вредност
CaseSensitive	Одређује да ли су поређења осетљива на величину слова
DataSetName	Име које се користи за референцирање објекта DataSet у коду.
DefaultViewMenager	Дефинише подразумевано филтрирање и редослед сортирања објекта DataSet.
EnforceConstrains	Одређује да ли се приликом измена поштују праваила за ограничења.
ExtendedProperties	Наменске корисникове информације.
HasErrors	Означава да ли неки од објеката DataRow из објекта DataSet садржи грешке.
Locale	Локалне информације које ће се користити приликом поређења знаковних низова.
Namespace	Простор имена који се користи приликом читања и писања XML документа.
Relations	Колекција објеката DatRelations која дефинише релације између објеката DataTable које се налазе у објекту DataSet.
Tables	Колекција објеката DataTable која се налази у објекту DataSet.

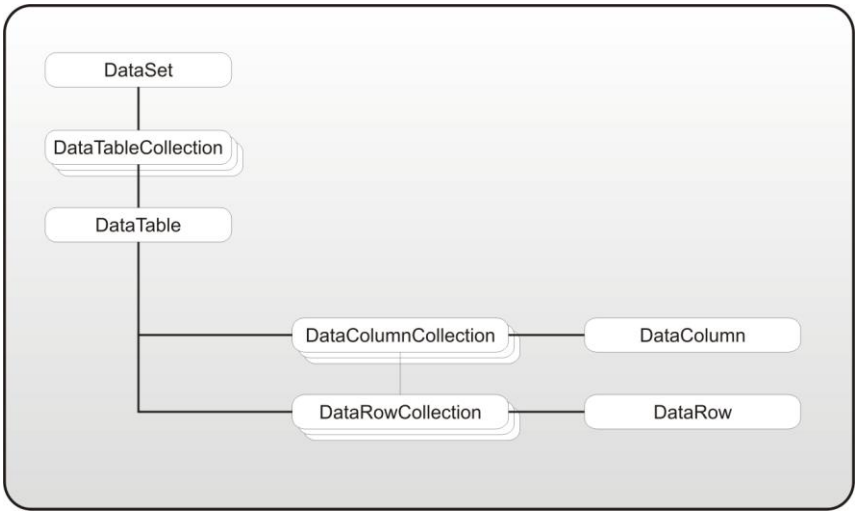
Већина својстава која подржава објекат DataSet је повезана са његовом интеракцијом са XML-ом. Од својстава која се не односе на XML најважније су колекције Tables и DataRelations, које садрже и дефинишу податке који се одржавају унутар објекта DataSet.

4.6. Објекат DataTable

Овај последњи параметар који указује на једну од најбитнијих разлика између објекта DataSet и ADO скупа записа, а то је да објекат DataSet може садржати неколико

табела података. Иако је нешто слично постојало у програму ADO у оквиру опције уређивања података, ипак је објекат DataSet учинио корак више – његове табеле могу бити узете из различитих извора података. Такође, решен је проблем везан за употребу ужасне синтаксе SHAPE. ADO.NET поседује класу DataTable која представља једну табелу у оквиру објекта DataSet. Објекат DataSet садржи својство Tables које враћа колекцију поменутих објеката (DataTableCollection).

Класа DataTable приказује податке у уобичајеном табеларном формату и поседује колекцију објеката DataColumn и DataRow које представљају сваку колону и сваки ред.



Слика 18: DataTable[6]

Иако класа DataColumn одговара ADO објекту Field, програм ADO није поседовао објекат који је приказивао један ред података, те стога се са правом може рећи да класа DataRow представља велику предност.

Ако се жели приступити подацима у класи DataTable, потребно је прво приступити одговарајућем објекту DataRow, а затим и индексирати објекат како би се добили подаци за жељену колону. Индекс може бити нумерички индекс колоне (0 за прву колону итд.) или назив колоне.

Конструктори објекта DataTable:

Табела 23: Конструктори објета DataTable[1]

Метод	Опис
New()	Прави нови објекат DataTable.
New(TableName)	Прави нови објекат DataTable са именом специфікованим у знаковном низу TableName.

New(SerializableInfo, StreamingContext)	Користи га интерно .NET Framework.

Методи за додавање табела у колекцију Tables објекта DataSet:

Табела 24: Методи колекције Tables[1]

Метод	Опис
Tabless.Add()	У објекту DataSet прави нови објекат DataTable са именом TableN, где је N секвенцијални број.
Tabless.Add (Tablename)	Прави нови објекат DataTable чије име је спецификовано у знаковном низу TableName.
Tabless.Add (DataTable)	Додаје спецификовани објекат DataTable у објекат DataSet.
Tabless.Add Range(TableArray)	У објекат DataSet додаје објекте DataTable који се налазе у низу TableArray објекта DataTable

Верзија метода Add прави објекат DataTable са именом TableN, где је N секвенцијални број. Методом Add(TableName) прави се нова табела и њено својство TableName поставља се на знаковни низ који је прослеђен као параметар.

4.6.1. Својства објекта DataTable

Својства објекта DataTable:

Табела 25: Својства DataTable[1]

Својство	Опис
CaseSensitive	Одређује начин на који ће се обавити поређење знаковних низова.
ChildRelations	Колекција објеката DataRelation у којима је овај објекат DataTable табела родитељ.
Columns	Колекција објеката DataColumn који се налазе у објекту DataTable
Constrains	Колекција ограничења која се одржавају у овом објекту DataTable.

DataSet	Објекат DataSet чији члан је овај објекат DataTable.
DisplayExpression	Израз који се користи за представљање имена табеле у корисничком интерфејсу (UI).
HasErrors	Означава да ли постоје грешке у неком од редова који припадају објекту DataTable.
ParentRelations	Колекција објеката DataRelation у којима је овај објекат DataTable табела дете.
PrimaryKey	Низ колона које функционишу као примарни кључ табеле.
Rows	Колекција редова који припадају табели.
TableName	Име објекта DataTable у објекту DataSet. Ово је име помоћу којег се објекат DataTable референцира у коду.

Ако објекат DataTable припада објекту DataSet, вредност својства CaseSensitive биће једнака вредности истог својства одговарајућег објекта DataSet. Подразумевана вредност је *False*.

Колекције ChildRelations и ParentRelations садрже референце на објекте DataRelations које табелу референцирају као дете или као родитељ. За већину независних објеката DataTable ове колекције ће бити *Null*, али је теоретски могуће додати релацију у колекцији ChildRelations и ParentRelations ако је објекат DataTable повезан са самим собом.

Својство DisplayExpression слично је својству Caption за колону, по томе што одређује како ће се име табеле приказивати кориснику у време извршавања, али за разлику од својства Caption, својство DisplayExpression користи израз за одређивање вредности у време извршавања. Једна од примена својства DisplayExpression је да се на основу садржаја табеле одреди начин на који ће се табела приказивати.

4.6.2. Колекција Columns

Колекција Columns објекта DataTable садржи нула или више објеката DataColumn који дефинише структуру табеле. Ако је објекат DataTable направљен коришћењем метода *Fill* или *FillSchema* објекта DataAdapter, колекција Columns ће бити генерисана аутоматски.

Конструктори објекта DataColumn:

Табела 26: Конструктори DataColumn[1]

Својство	Опис
New()	Прави нови објекат DataColumn без дефинисаних својстава ColumnName или Caption.
New(columnName)	Прави нови објекат DataColumn са именом специфікованим знаковним низом columnName
New(columnName, DataType)	Прави нови објекат DataColumn са именом специфікованим знаковним низом columnName и типом података који је специфікован параметром DataType.
New(columnName, DataType, Expression)	Прави нови објекат DataColumn са именом специфікованим знаковним низом columnName, и специфікованим својствима: DataType и Expression.
New columnName, DataType Expression, ColumnMapping)	Прави нови објекат DataColumn са именом специфікованим знаковним низом columnName и специфікованим својствима: DataType,Expression и ColumnMapping.

Својства објекта DataColumn:

Табела 27: Својства DataColumn[1]

Својство	Опис
AllowDbNull	Одређује да ли се колона може оставити празном.
AutoIncrement	Одређује да ли ће систем аутоматски увећати вредност колоне за један.
AutoIncrementSeed	Почетна вредност колоне AutoIncrement.
AutoIncrementStep	Прираштај за који ће се вредност колоне AutoIncrement увећавати.
Caption	Име колоне које служи за приказивање у неким контролама, као што је на пример DataGrid. Подразумевана вредност је једнака вредности својства ColumnName.
ColumnName	Име колоне у колекцији Tables објекта DataSet. Овој је име помоћу којег се колона може референцирати у коду.
DataType	Тип података колоне у .NET Framework-у.

DefaultValue	Вредност колоне коју ће обезбедити систем ако се не обезбеди ниједна друга вредност.
Expression	Израз који се користи за израчунавање вредности колоне.
MaxLength	Максимална дужина текстуалне колоне.
ReadOnly	Одређује да ли се колона може мењати пошто се ред који је садржи унесе у табелу.
Unique	Одређује да ли сви редови у табели морају имати јединствену вредност у овој колони.

4.6.3. Колекција Rows

Колекција Rows објекта DataTable садржи стварне податке који се налазе у објекту DataTable. Ти подаци су у облику једног или више објеката DataRow.

Својства објекта DataRow:

Табела 28: Својства DataRow[1]

Својство	Опис
HasErrors	Означава да ли ред садржи грешке.
Item	Вредност колоне у објекту DataRow.
ItemArray	Вредност свих колона у објекту DataRow представљена у облику низа.
RowError	Неменски опис грешке за ред.
RowState	Стање у коме се налази ред.
Table	Објекат DataTable којем припада објекат DataRow.
DataType	Тип података колоне у .NET Framework-у.

Пошто је својство Rows колекција, нови подаци се могу додати у објекат DataTable коришћењем метода *Add*, који доступан у два различита облика:

Табела 29: Облици методе Add[1]

Метод	Опис
Add(DataRow)	Додаје спецификовани објекат DataRow у табелу.
Add(dataValues())	Прави нови објекат DataRow у табели и вредности његовог свостава Item поставља коришћем низа објекта dataValues.

Својство RowState објекта DataRow осликава акције које су изведене од тренутка стварања објекта DataTable или од тренутка последњег позива метода *AcceptChanges*. Могуће вредности својства RowState су:

Табела30: Врености својства RowState[1]

Својство	Опис
Added	Објекат DataRow је додат.
Deleted	Објекат DataRow је обрисан из табеле.
Detached	Објекат DataRow још увек није додат у табелу.
Modified	Садржај објекта DataRow је измењен.

4.7. Објекат DataView

Објекат DataView обезбеђује филтриран и сортиран поглед на појединачан објекат DataTable. Иако објекат DataView обезбеђује исту функционалност као и метода *Select* објекта DataTable, он садржи многе предности. Објекти DataView могу се правити и конфигурисати и у време дизајнирања и у време извршавања, чиме се у многим случајевима олакшава њихова имплементација.

Може се направити више објеката DataView за било који објекат DataTable. Сваки објекат DataTable садржи барем један објекат DataView у свом својству DefaultDataView. Својства објекта DefaultDataView могу се подесити у време извршавања, али не и у време дизајнирања.

Редови објекта DataView, иако веома слични објектима DataRow, уствари су објекти DataRowView коиј референцирају објекте DataRow. Својства објекта DataRowView су:

Табела 31: Својства DataRowView[1]

Својство	Опис
DataView	Објекат DataView којем припада објекат DataRowView.
IsEdit	Означава да ли се објекат DataRowView тренутно мења.
IsNew	Означава да ли је објекат DataRowView потпуно нов.
Item	Вредност колоне у објекту DataRowView.

Row	Објекат DataRow који се прегледа.
RowVersion	Тренутна верзија објекта DataRowView.

Објекат DataView подржава конструктор *New* помоћу којег се омогућава стварање објекта DataView у време извршавања. Објекат DataView подржава две верзије конструктора *New*.

Табела 32: Верзије конструктора New[1]

Метод	Опис
New()	Прави нови објекат DataView.
New(DataTable)	Прави нови објекат DataView и његово својство Table поставља на спецификовани објекат DataTable.

4.7.1. Својства објекта DataView

Табела 33: Својства DataView[1]

Својство	Опис
AllowDelete	Одређује да ли се редови у објекту DataView могу брисати.
AllowEdit	Одређује да ли се редови у објекту DataView могу мењати.
AllowNew	Одређује да ли се редови могу додавати у објекту DataView.
AllowDefaultSort	Одређује да ли ће се користити подразумевани редослед сортирања, који се одређује у зависности од припадајућег извора података.
Count	Број објеката DataRowView у објекту DataView.
DataViewMenager	Објекат DataViewMenager којем припада објекат DataView.

Item(index)	Објекат DataRowView који се у објекту DataView налази на специфікованом индексу *Индех*.
RowFilter	Израз који се користи за филтрирање редова који се налазе у објекту DataView.
RowStateFilter	Променљива типа DataViewRowState која се користи за филтрирање редова који се налазе у објекту DataView.
Sort	Израз који се користи за сортирање редова који се налазе у објекту DataView.
Table	Објекат DataTable који је извор редова за објекат DataView.

Својства AllowDelete, AllowEdit, AllowNew одређује да ли се подаци из извора података који су осликани у објекту DataView могу мењати коришћењем објекта DataView.

Својство Count враћа број објеката DataRow који су осликани у објекту DataView, док својство DataViewMenager и Table служе за повезивање објеката DataView са осталим објектима у апликацији

Својства RowFilter, RowStateFilter и Sort управљају који ће објекат DataRow бити осликан у објекту DataView и како ће бити сортиран.

Изрази објекта DataColumn (*DataColumn Expresions*) користе својства RowFilter и Sort објекта DataView. Израз објекта DataColumn је знаковни низ и за његово стварање могу се користити све функције за обраду знаковних низова. Изрази објекта DataColumn подржавају и већи број Специјалних функција:

4.7.2. Својство RowStateFilter

Сваки објекат DataRow одржава свој статус у својтву RowState. Својство RowStateFilter објекта DataView може се користити за ограничавање објеката DAtaRowView који се налазе у објекту DataView на оне који су у одговарајућем стању или се може користити за враћање вредности стања. Вредности својства RowStateFilter су:

Табела 34: Својства RowStateFilter[1]

Својство	Опис
Added	Само они редови који су били додати.
CurrentRows	Све тренутне вредности редова.
Deleted	Само они редови који су били обрисани.
ModifiedCurrent	Тренутне вредности редова за редове који су били мењани.
ModifiedOriginal	Првобитне вредности редова који су били мењани.
None	Ниједан ред.
OriginalRows	Првобитне вредности свих редова.
Unchanged	Само они редови који нису били мењани.

4.7.3. Методи објекта DataView

Табела 35: Методи DataView[1]

Метод	Опис
AddNew	Додаје нови објекат DataRowView у објекат DataView.
Delete	Уклања објекат DataRowView из објекта DataView.
Find	Проналази један или више објеката DataRowView који садрже специфициране вредности примарног кључа.

Метод *Find* објекта DataView проналази један или више редова на основу вредности примарног кључа. Ако је потребно пронаћи ред на основу вредности неке друге колоне, потребно је користити својство RowFilter објекта DataView.

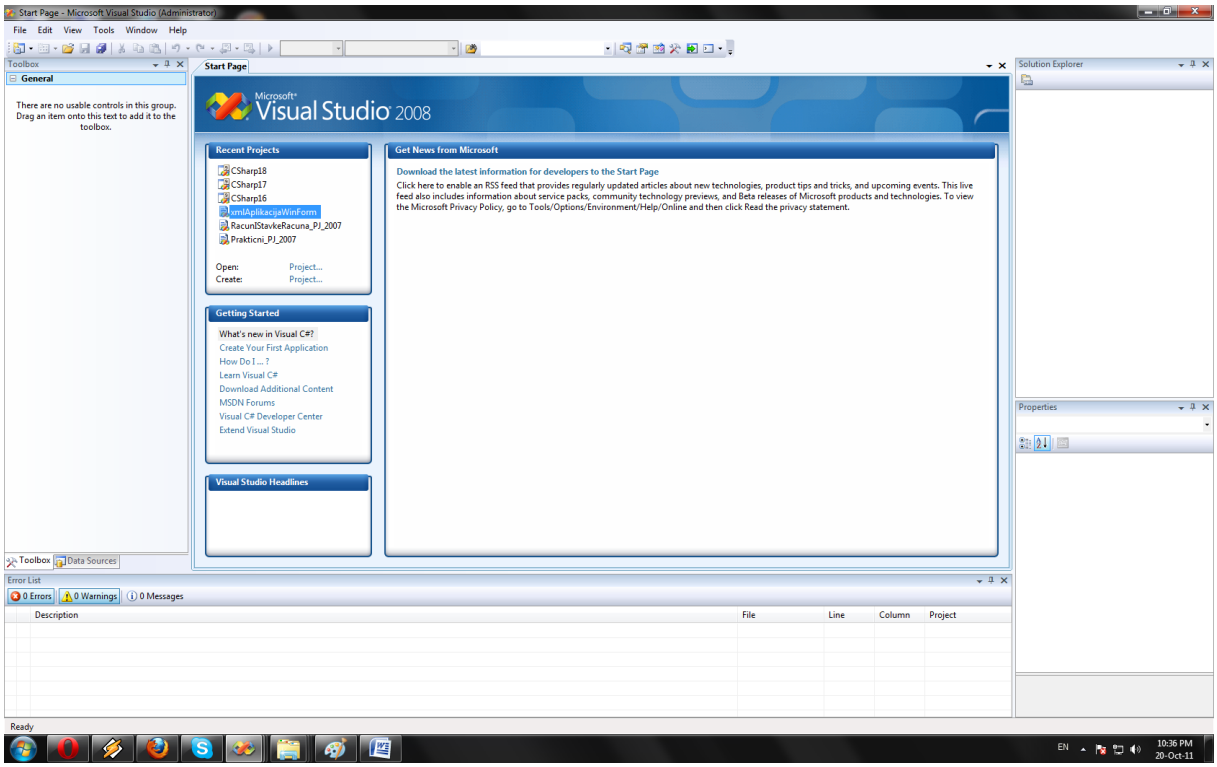
Постоје две верзије метода *Find* које омогућавају да му се проследи или појединачна вредност или низ вредности. Метод *Find* враћа индекс пронађеног реда (или низ редова у случају да му је прослеђен низ примарних кључева) или *Null* ако се вредност не пронађе у објекту DataView.

V. Водич за израду практичних примера

5.1. SQL SELECT упит

- Стартовање Visual Studio.NET –а

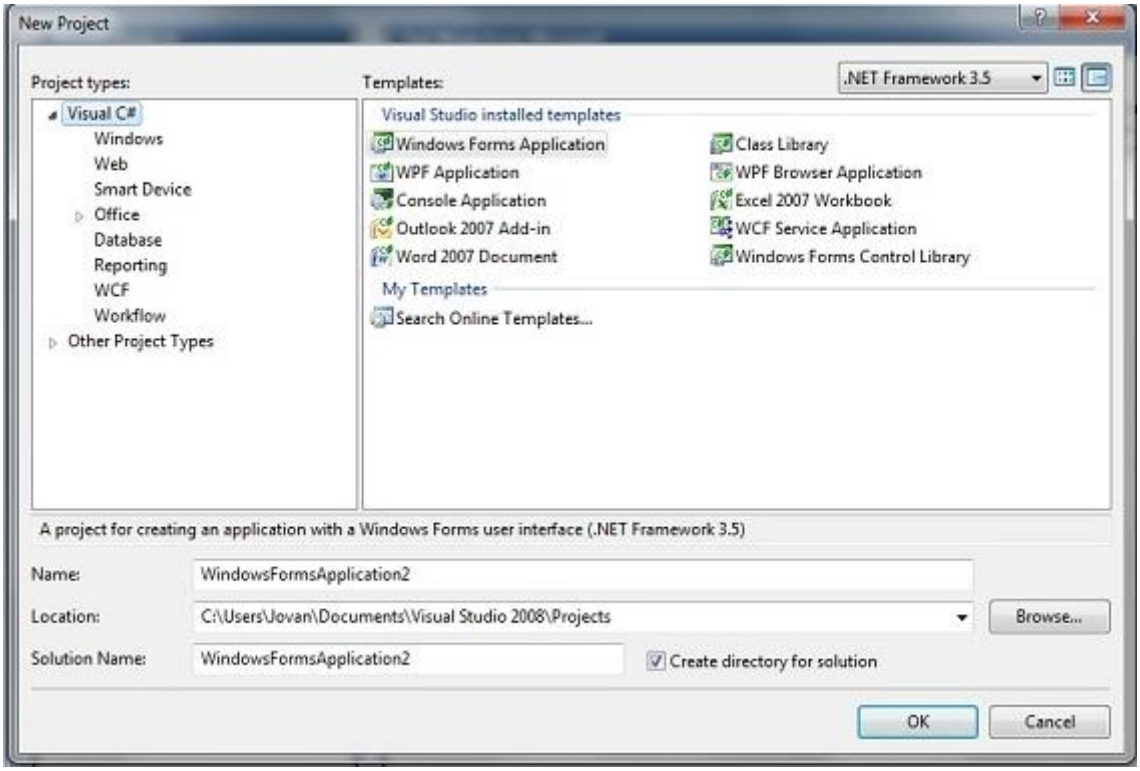
Приликом стартовања Visual Studio .NET-а појављује се уводни екран (Слика 19).



Слика 19: Уводни екран у Visual Studio .NET-у

Затим покрећемо нови пројекат кликом на File (на основном менију) па потом на New project који отвара истоимени оквир за дијалог (Слика 20).

После тога бирамо тип пројекта који ће у насем случају бити Windows Application и у пољу Name куцамо назив пројекта.



Слика 20: New project оквир за дијалог

За реализацију пројекта непходна је по једна **TextBox** и **Button** контрола као и контрола **DataGridView**. У питању је веома моћна контрола која је способна да прикаже садржај табеле у DataSet објекту.

Контроле морају имати следећа својства:

TextBox - својство **MultiLine** на **True**, како би корисник могао да откуца SQL SELECT упит у више редова. У складу са овим поставити и својство **ScrollBars** на **Vertical.** Иницијално име контроле задржати – **textBox1**

Button - својство **Text** на **Start** и такође оставити иницијално име на **button1**

DataGridView - поставити тако да заузима већи део форме. Неопходно је да контрола прати измене димензија форме што се постиже постављањем **Anchor** својства. Контролу треба привезати на све четири стране, чиме се аутоматски одржава исто растојање од ивица форме.

Кликом на дугме "START" се извршава све, од постављања конекције до везивања података на DataGridView контролу.

Урадити дупли клик на дугме и пре него што се крене са писањем кода, згодно је (не и неопходно) додати једну `using` директиву како би скратили писање. На врху прозора за писање кода уочава се неколико постојећих `using` директива, а потребно је додати још једну:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Link;
using System.Text;
using System.Windows.Forms;
using system.Data.SqlClient;
```

Вратити се на button1_Click догађај и по реду како је приказано на дијаграму писати код:

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

            SqlConnection Konekcija = new SqlConnection ();
            Konekcija.ConnectionString = @"Data
Source=.\SQLEXPRESS;AttachDbfilename=C:\Baza\HORTHWND.MDF;Integrated
Security=True;Connect Timeout=30;User Instance=True";
```

- Овде треба пажљиво унети својства *ConnectionString*. Велика и мала слова нису битна, али размак у параметру *Data Source* јесте битан.

```
            //KOMANDA

            SqlCommand Komanda = new SqlCommand();
            Komanda.Connection = Konekcija;
            Komanda.CommandText=textbox1.Text;
```

- Текст укуцан у контролу *textBox1* користимо за *CommandText* својство команде, тако да то мора бити валидан SQL SELECT упит.

```
            // DATASET

            DataSet Ds = new DataSet();


            //DATA ADAPTER

            SqlDataAdapter Da = new SqlDataAdapter ();
            Da.SelectCommand = Komanda;
            Da.Fill(Ds);

            // Prikaz sadrzaja prve tabele

            dataGridview1.DataSource=Ds.Tables[0];
        }
```

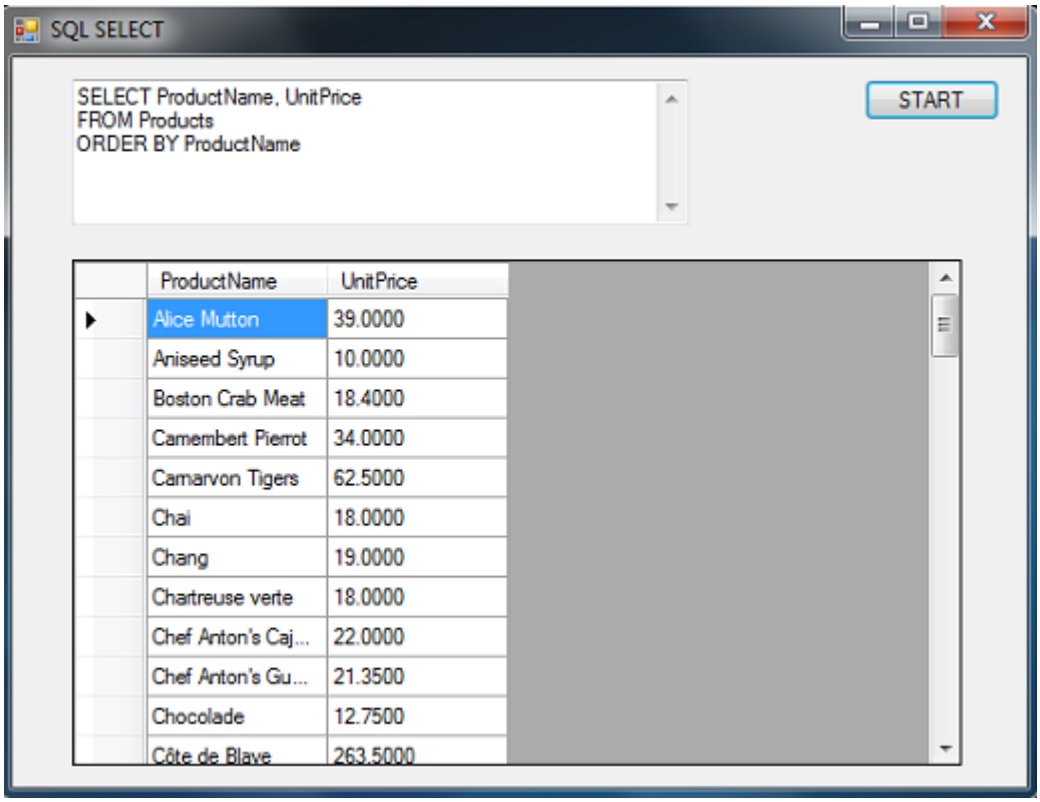
Fill метода **SqlDataAdapter** класе користи се за "пуњење" **DataSet**-а. Ова метода заправо покреће **SelectCommand** команду, која даље отвара конекцију и бази прослеђује упит. Резултат овог упита се прослеђује **DataSet**-у који прави једну или више тебела.

Својство **DataSource**, као што се може претпоставити из назива, показује који је извор података које треба приказати у контроли. У овом примеру, извор података је прва (и једина) табела у **DataSet**-у која има индекс 0 (**Ds.Tables[0]**). Сваки **Table** објекат је матрица састављена од редова и колона и чува резултат једног SQL упита.

Након завршетка писања кода покренути апликацију једноставним притискањем тастера F5 (Слика 20).

Један од упита који се може проследити гласи:

SELECT ProductName, UnitPrice
FROM Products
ORDER BY ProductName



Слика 20: Новодобијени изглед апликације

5.2. Data Binding

У претходном примеру је показано везивање са подацима (енг. Data Binding) када је постављено својство **DataSource** контроле **DataGridView**. Слична ствар се може заправо одрадити и са сваком контролом корисничког интерфејса. Везивање података значи да се у некој контроли аутоматски приказују подаци из базе, најчешће као резултат SELECT упита. Пошто SELECT упит може да обухвати податке из једне или више табела, ово представља веома елегантан начин за приказ података кориснику, уз минимум програмирања. Контроле се повезују на потребну табелу у оквиру **DataSet** објекта.

Принцип рада је идентичан као што је демонстрирано у претходном примеру, уз додатак пар линија кода које су одговорне за повезивање података. Постоје две варијанте које можемо користити и чији избор зависи од могућности саме контроле. Неке контроле су у стању да табеларно прикажу податке, као што је DataGridView. У овом случају потребно је подесити само DataSource својство и то је све. Овај начин везивања се зове **complex data binding**. Неке друге контроле као што су TextBox, Label и сличне очигледно не могу приказати редове и колоне података, већ могу да прикажу само вредност одређене колоне (поља из табеле), одређеног слога(реда у табели). Ово нас доводи до појма Current record (активни слог-запис) чије вредности колоне желимо да прикажемо у контроли.

Свака контрола ка подржава везивање података поседује коекцију **DataBindings** која може да садржи више везивања података за исту контролу, а у пракси се користи најчешће само једна. Важно је напоменути да можемо одредити на које својство контроле желимо да вежемо вредност колоне активног слога. Ако желимо да податке прикажемо у **TextBox** контроли, то чинимо преко њеног својства **Text**. Ако у подацима имамо колону која може имати вредност тачно/нетачно (енг. true/false) логично је користити **CheckBox** контролу и везивати податке на њено својство **Checked.**

Илустрација свега наведеног најбоље ће се видет кроз једноставан пример за simple и complex data bindings. Као и у прошлом примеру и у овом ћемо користити објекте:

- `SqlConnection`
- `SqlCommand`

- DataSet
- SqlDataAdapter

За приказ података користити Microsoft-ову базу Northwind.mdf и следеће колоне из тебеле Products:

- ProductName (назив производа)
- UnitPrice (цена производа)
- Discontinued (да ли је производ доступан)

Урадити дупли клик на форму како би добили Form1_Load догађај.

```
Private void Form1_Load(object sender, EventArgs e)
        {
            // Konekcija
            SqlConnection Konekcija = new SqlConnection ();
            knekcija.ConnectionString = @"Data
Source=.\SQLEXPRESS;AttachDbfilename=C:\Baza\NORTHWND.MDF;Integrated
Security=True;Connect Timeout=30;User Instance=True";

            // Komanda
            SqlCommand Komanda = New SqlCommand ();
            Komanda.Connection = Konekcija;
            Komanda.CommandText = "SELECT ProductName, UnitPrice,
Discontinued FROM Products ORDER BY ProductName";

            // DataSet
            DataSet Ds = new DataSet();

            // DataAdapter
            SqlDataAdapter Da = new SqlDataAdapter();
            Da.SelectCommand = Komanda;
            Da.fill(Ds);
```

Овим делом је практично припремљен терен за simple и complex повезивање података. На форму треба поставити елементе корисничког интерфејса следећим редоследом:

- једну **DataGridView** контролу
- две **Label** контроле
- две **TextBox** контроле
- једну **CheckBox** контролу

У наставку кода је потребно дописати само једну линију како би приказали податке у **DataGridView1** контроли која омогућава симпле дата биндинг:

```
// Simple data binding
dataGridView1.DataSource=Ds.Tables[0];
```

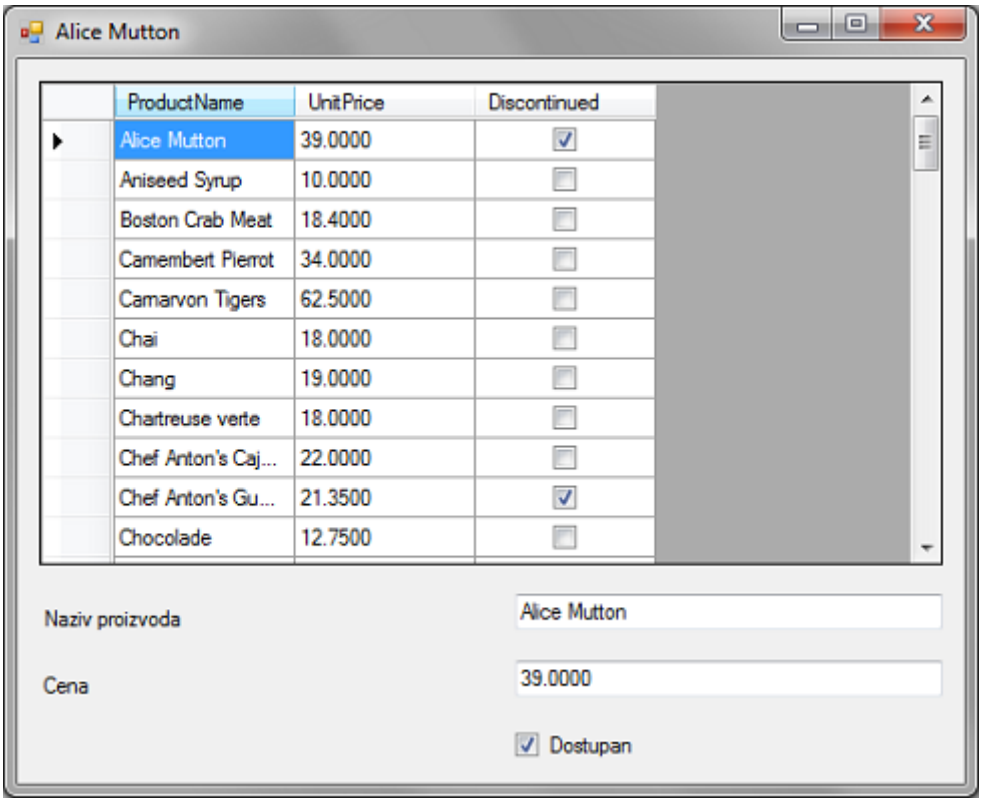
У две TextBox контроле треба приказати вредности колона "ProductName" и "UnitPrice". Ово представља complex data binding:

```
// Complex data binding
textBox1.DataBindings.Ass("Text", Ds.Tables[0],"ProductName");
textBox2.DataBindings.Адд("Text", Ds.Tables[0], "UnitPrice");
```

На крају треба приказати вредност колоне "Discounted" у контроли **CheckBox**. Вредности овде могу бити само **true** или **false**, због чега треба обратити пажњу да не везујемо својство **"Text"** што је био случај за контроле TextBox, већ за својство **"Checked"**.

```
checkBox1.DataBindings.Add("Checked", Ds.Tables[0],"Discontinued");
```

Приликом покретања пројекта добиће се изглед форме као на следећој слици (Слика 21):



Слика 21: Завршни изглед апликације

Када у табели мењамо слог, аутоматски се мењају и подаци испод. Пошто су све контроле везане за исти извор података (енг. DataSource), у овом случају на прву табелу Ds објекта, логично је да је приказ синхронизован. Такође се може приметити да подаци могу да се мењају директно у табели или у контролама испод ње. Међутим, све измене које настају су локалног типа, односе се на DataSet објекат који је смештен у меморији рачунара.

5.3. DataReader класа

Приликом коришћења DataSet класе, део података из базе (која је најчешће негде на мрежи) се преноси на рачунар на коме се налази апликација. Количина података која се преноси директно зависи од SELECT упита које се шаље бази на обраду. У вези са овим морамо водити рачуна о томе да се у SELECT упиту увек ограничи број редова и колона који су нам заиста потребни.

Избегавати упите типа: SELECT*FROM *ImeTabele*

Оваквим упитима се комплетан садржај табеле преноси до клијентског рачунара. Ако замислимо само да табела има пар стотина хиљада слогова и да апликација ради на двадесетак рачунара - јесно је да не постоји мрежа нити диск на серверу који може опслужити овакав захтев.

Међутим, у неким ситуацијама је баш потребно пренети велике количине података. Најчешћи примери су различити извештаји који типично захтевају податке из више табела истовремено. На срећу, они су потребни само периодично, покрећу се типично само на једном рачунару и само тада долази до оптерећења мрежних и серверских ресурса.

.NET нуди класу која је оптимизована за овакве потребе. Приступ подацима је секвенцијалан, од првог ка последњем. Нема потребе за напред/назад кретањем кроз слогове што резултује мањим оптерећењем сервера. Овакав приступ подацима се на енглеском зове Forward Only.

Такође, не преносе се сви подаци од сервера ка клиенту. Преноси се само слог по слог тако да у једном моменту на клијенту постоји увек само један слог што резултује минималним оптерећењем клијента и мреже.

На крају, подаци добијени на овај начин су само за читање (енг. Read Only) што додатно убрзава пренос и растерећује сервер.

Класа из .NET-а која имплементира ове финкционалности зове се SqlDataReader када радимо са Microsoft SQL Server базом података.

За илустрацију теорије користити Northwind.mdf базу података. На иницијални форму поставити једно дугме (button) и једну листу (listbox) са следећим својствима:

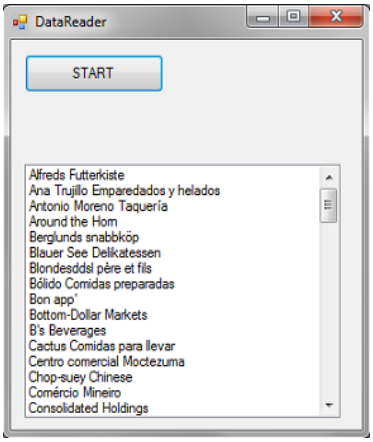
- **Button** - име дугмета је **btnStart** а текст **START**
- **Listbox** - име листе је **lstKupci**

Урадити дупли клик на дугме и унети следећи код:

```
Private void btnStart_Click(object sender, EventArgs e)
        {
            // Kreiranje objekta SqlConnection

            SqlConnection Konekcija = new SqlConnection ();
            Konekcija.ConnectionString = @"Data
Source=.\SQLEXPRESS;AttachDbfilename=C:\Baza\NORTHWND.MDF;Integrated
Security=True;Connect Timeout=30;User Instance=True";

            // Kreiranje SqlCommand objekta I prosledjivanje SELECT upita

            SqlCommand Komanda = new SqlCommand ();
            Komanda.CommandType = CommandType.Text;
            Komanda.CommandText = "SELECT CompanyName FROM Customers";
            Komanda.Connection = Konekcija;

            // Deklaracija SqlDataReader objekta

            SqlDataReader Dr;

            //Otvaranje konekcije- obavezno!!

            Konekcija.Open();

            // Izvrsavanje ExecuteReader metoda

            Dr = Komanda.ExecuteReader();

            // Brisanje sadrzaja liste (ako postoji)

            lstKupci.Items.Clear();


            while (Dr.Read() == True)
            {
                lstKupci.Items.Add(Dr["CompanyName"].ToString());
            }

            // Zatvaranje konekcija

            Konekcija.Close();
            konekcija.Dispose();
            Komanda.Dispose();
            Dr.Dispose();

        }
```

Поступак рада са SqlDataReader класом је прилично сличан ономе што је приказано до сада и може се описати кроз следеће поступке:

- Креирање и дефинисање објекта над SqlConnection класом.
  Као и увек неопходна је конекција ка бази података.
- Креирање и дефинисање објекта над SqlCommand класом.
  Команда одређује SELECT SQL упит којим се потражују подаци. Када радимо са SqlDataReader класом можемо користити искључиво SELECT упит.
- Креирање објекта над SqlDataReader класом.
  Приликом креираје не можемо користити **new()** конструктор, већ само декларишемо варијаблу SqlDataReader типа.
- Објекат који представља SqlCommand класу поседује методу **ExecuteReader** која као резултат враћа тип **SqlDataReader**. Позивом ове методе иницирамо целу процедуру. **ExecuteReader** метода неће отворити конекцију аутоматски, тако да је пре њеног позива неопходно то урадити, у супротном ће се јавити грешка у току извршавања програма (енг. Run time error).
- Следећи поступак је пролазак кроз петљу у којој се довлачи слог по слог, читају вредности потребних колона и са њима нешто ради (формирају извештаји, врши испис на екран или штампач и слично).
  Приступ одређеној колони из слога је једноставан: ***Ime SqlDataReader objekta ["Ime kolone"]*** , као што се могло видети у примеру.
  Довлачење слогова се врши методом Read() класе SqlDataReader. Метода са сервера захтева следећи слог по реду и ако га добије враћа True. У супротном враћа False, што је индикација да смо стигли до краја и да треба завршити петљу Први позив Read() методе нам враћа први слог.
- На крају, очувања ресурса ради, затварамо конекцију и уништавамо све објекте.



Слика 22: Коначни изглед апликације

Закључак

После свега горе наведеног, можемо закључити да .NET Framework технологија знатно олакшава само програмирање и да се у односу са старије, процедурално (алгоритамско) програмирање, данас може много лакше савладати начин програмирања.

За ADO.NET, може се са сигурношћу тврдити, да у многоме олакшава приступ изворима података, који представља неку врсту неизбежности која прати развој .NET окружења и појаву новог приступа подацима. Како ова компонента представља велики напредак остварен у развоју софтвера, у односу на претходне верзије, једно је сигурно да су конектовања са базом знатно растерећенија и да је уз помоћ ње капацитет конекције повећан.

У ствари, без обзира на то да ли ћете га заволети или замрзити, изгледа да се једна чињеница о .NET радном окружењу не може порећи: *правила игре се мењају*. Без обзира на то да ли сте у овој рачунарској игри толико дуго да сте већ престали да се сећате својих почетака или још увек сортирате своје хипове и стекове, учење .NET радног окружења захтеваће већа залагања, јер сада смо *сви почетници*: