



TEHNIČKO VELEUČILIŠTE U  
ZAGREBU ZAGREB UNIVERSITY  
OF APPLIED SCIENCES

# NAPREDNE TEHNIKE PROGRAMIRANJA U C++ BUILDERU

dr. sc. Željko Kovačević

Zagreb, 2022

**PRIRUČNICI TEHNIČKOG VELEUČILIŠTA U ZAGREBU**  
**MANUALIA POLYTECHNICI STUDIORUM ZAGRABIENSIS**

*Željko Kovačević*

# NAPREDNE TEHNIKE PROGRAMIRANJA U C++ BUILDERU

*Udžbenik na kolegiju 'Napredne tehnike programiranja' koji se održava u sklopu nastave na  
dodiplomskim stručnim studijima informatike i računarstva Tehničkog veleučilišta u Zagrebu*

Zagreb, 2022.



**TEHNIČKO VELEUČILIŠTE U  
ZAGREBU** ZAGREB UNIVERSITY  
OF APPLIED SCIENCES

izdavač	Tehničko veleučilište u Zagrebu Vrbik 8, Zagreb
za izdavača	mr. sc. Goran Malčić
autor	dr. sc. Željko Kovačević
recenzenti	prof. dr. sc. Danijel Radošević Hrvoje Rončević, dipl. ing.
vrsta djela	udžbenik Objavljivanje je odobrilo Stručno vijeće Tehničkog veleučilišta u Zagrebu odlukom broj: 3594-2 red. 11/20 od 17. studenog 2020
ISBN	978-953-7048-91-4



# SADRŽAJ

1. C++ Builder .....	9
1.1. Uvod .....	9
1.2. Podržane platforme i tipovi aplikacija .....	10
1.3. Aplikacije komandne linije .....	12
1.4. Uređivač programskog kôda .....	15
1.5. Otkrivanje i ispravljanje grešaka.....	16
2. Aplikacije s grafičkim korisničkim sučeljem .....	19
2.1. Windows VCL aplikacije.....	19
2.2. Dijalozi za prikaz poruka .....	21
2.3. Dijalozi za unos podataka .....	27
2.4. Svojstva, događaji i metode .....	30
2.5. VCL komponente .....	31
2.6. Komunikacija među dijalogima .....	38
2.7. Akcijske liste .....	40
2.8. Lokalizacija korisničkog sučelja.....	41
3. Formati za pohranu i razmjenu podataka .....	47
3.1. INI.....	47
3.2. Windows registar .....	49
3.3. XML .....	52
3.4. JSON.....	59
3.5. Razvoj prilagođenog formata .....	64
4. Baze podataka .....	69
4.1. Pristupi i tehnologije.....	69
4.2. Povezivanje s bazom podataka .....	71
4.3. Obrada podataka .....	74
4.4. Odnos <i>master-detail</i> .....	80
4.5. Posebni tipovi polja.....	82
4.6. Generiranje izvještaja .....	84
5. Dretve i procesi .....	89
5.1. Dretve .....	89
5.2. Sinkronizirani pristup VCL komponentama .....	94
5.3. Kritična sekcija.....	96

5.4. Mutex.....	99
5.5. Biblioteka paralelnog programiranja .....	101
5.6. Procesi.....	105
6. Mrežne aplikacije .....	109
6.1. Komunikacija klijent-poslužitelj.....	109
6.2. Indy TCP/IP .....	110
6.3. Indy UDP .....	115
6.4. Indy HTTP .....	117
6.5. Indy ICMP.....	120
6.6. Windows servisi .....	122
7. Web servisi .....	125
7.1. SOAP klijent.....	125
7.2. SOAP poslužitelj .....	127
7.3. REST klijent .....	130
7.4. WebBroker REST poslužitelj.....	132
7.5. Instalacija ISAPI DLL-a u IIS web poslužitelju .....	140
8. Kriptografija.....	145
8.1. Simetrični kriptografski algoritmi.....	145
8.2. Asimetrični kriptografski algoritmi.....	149
8.3. Funkcije sažimanja .....	152
8.4. Digitalni potpis .....	154
9. Biblioteke i komponente .....	159
9.1. Statičke biblioteke .....	159
9.2. Dinamičke biblioteke .....	161
9.3. Statičko i dinamičko DLL povezivanje .....	165
9.4. Razvoj VCL komponenti .....	169
Popis slika .....	175
Popis primjera .....	179
Popis tablica.....	183
Reference .....	185
Ključne riječi .....	189

## Uvodna riječ

Pred vama je udžbenik pisan za potrebe studenata dodiplomskih stručnih studija informatike i računarstva na Tehničkom veleučilištu u Zagrebu. Sadržaj udžbenika obuhvaća sadržaj kolegija "Napredne tehnike programiranja", a opisuje razvoj aplikacija u Embarcadero C++ Builderu korištenjem VCL (*Visual Component Library*) biblioteke.

Za što brže usvajanje sadržaja ovog udžbenika poželjno je da čitatelj već ima određena iskustva u području objektno-orijentiranog programiranja, pri čemu je svakako prednost ukoliko je riječ o programskom jeziku C++. Udžbenik se fokusira na razvoj aplikacija koje u sebi sadrže jednu ili više različitih tehnologija, uzimajući u obzir danas aktualne pristupe i metode razvoja aplikacija. Tako se obrađuju teme poput razvoja grafičkog korisničkog sučelja, korištenja raznih tipova datoteka za pohranu i razmjenu podataka, korištenja baza podataka, mrežnih protokola, web servisa, kriptografskih algoritama itd.

Uz udžbenik potrebno je koristiti razvojno okruženje Embarcadero C++ Builder, dok je njegovu besplatnu inačicu (*Community Edition*) moguće preuzeti na mrežnim stranicama Embarcadera.

Ovim putem također želim zahvaliti svima koji su svojim savjetima i primjedbama doprinijeli kvaliteti ovog udžbenika, a tu su bili mnogobrojni kolege iz struke, recenzenti te studenti Tehničkog veleučilišta u Zagrebu.

dr. sc. Željko Kovačević

Tehničko veleučilište u Zagrebu, 2022.g.



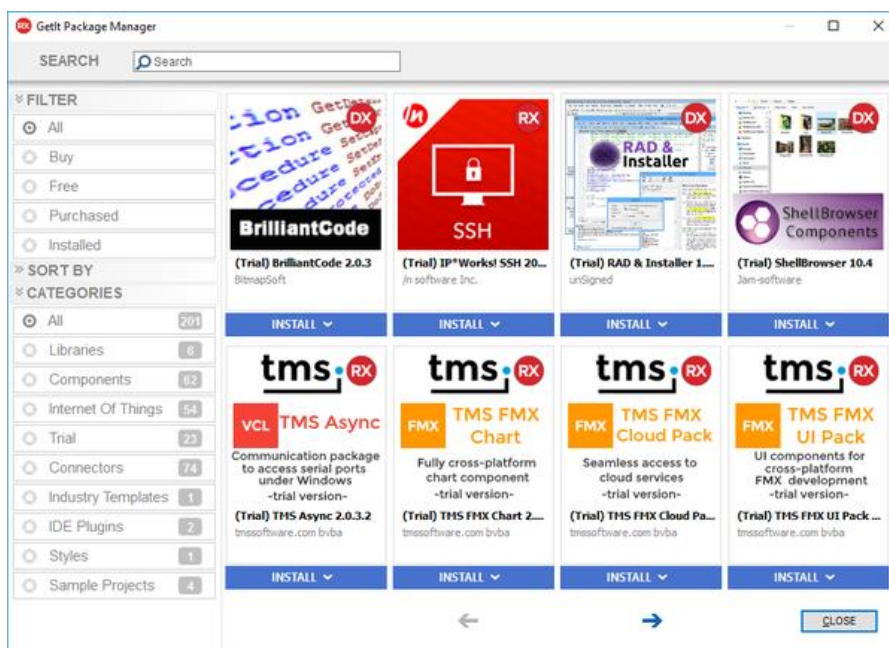


# 1. C++ Builder

## 1.1. Uvod

C++ Builder razvojno je okruženje za razvoj aplikacija koje pripada skupu RAD (eng. *Rapid Application Development*) alata tvrtke Embarcadero. C++ Builder koristi programski jezik C++, dok RAD Studio alati sadrže i razvojno okruženje Delphi koje koristi ekstenziju programskog jezika Pascal zvanu *Object Pascal*. Danas se, osim C++ Buildera i Delphija, u skupini RAD Studio alata nalaze i Embarcadero Prism te HTML5 Builder.

Jedna od najvećih prednosti RAD Studio alata brzina je razvoja aplikacija koja se postiže korištenjem velikog broja već postojećih te razvijanjem novih vizualnih i nevizualnih komponenti. Umjesto pisanja velike količine programskog kôda najčešće je tek potrebno koristiti komponente koje u svojoj implementaciji sadrže sve funkcionalnosti potrebne za izvršavanje određenih tipova zadataka. Zato RAD Studio alati inicijalno već nakon osnovne instalacije sadrže nekoliko stotina komponenti za rad s osnovnim i naprednim elementima grafičkog sučelja, bazama podataka, mrežnim protokolima, web servisima itd.



Slika 1.1.1. GetIt Package Manager

RAD Studio sadrži i repozitorij *GetIt Package Manager* (Slika 1.1.1) koji omogućuje izravno preuzimanje i instalaciju raznih paketa besplatnih i komercijalnih komponenti te alata.

Primjerice, tu su kompleti komponenti za korištenje funkcija sažimanja (eng. *hash functions*) i kriptografskih algoritama, razni generatori izvještaja, alati za poboljšanje već postojećih funkcionalnosti razvojnog okruženja itd.

## 1.2. Podržane platforme i tipovi aplikacija

RAD Studio alati počeli su se razvijati još 1995. godine kada izlazi prva inačica razvojnog okruženja Delphi. Ona je bila namijenjena samo za razvoj Windows aplikacija korištenjem biblioteke VCL (eng. *Visual Component Library*), dok se po uzoru na nju 1997. pojavljuje prva inačica C++ Buildera. Od tada Delphi i C++ Builder koriste vizualno identično razvojno okruženje (eng. *Integrated Development Environment, IDE*) te istu VCL biblioteku.

OS	Supported Versions	Target Platform (Project Manager node)	Supported Languages		Supported UI Frameworks	
			Delphi	C++	FireMonkey	VCL
Windows	Windows 8 – 10 (Including Windows 10 Anniversary Update and Windows 10 Creators Update) Windows 7 (SP1+) Windows Server 2012, 2016	32-bit Windows	✓	✓	✓	✓
		64-bit Windows	✓	✓	✓	✓
OS X	10.10 – 10.13 (Yosemite, El Capitan, Sierra, and High Sierra)	OS X	✓	✓	✓	
iOS	iOS 9, 10, and 11	iOS Device 32-bit	✓	✓	✓	
		iOS Device 64-bit	✓	✓	✓	
		iOS Simulator	✓		✓	
Android	5, 6, 7, and 8 4.1.x – 4.4.x	Android	✓	✓	✓	
Linux	Ubuntu Server (Ubuntu 16.04 LTS) RedHat Enterprise Linux (version 7)	64-bit Linux	✓		✓	

Slika 1.2.1. Podržani operacijski sustavi i platforme [1]

Kroz više od 20 godina razvoja RAD Studio alati omogućavali su razvoj aplikacija za sve više operacijskih sustava i platformi (Slika 1.2.1). Primjerice, prva inačica Delphija imala je potporu za 16-bitnu platformu Windowsa, dok danas podržava 32 i 64-bitne platforme te razvoj aplikacija za operacijske sustave OS X, iOS, Android i Linux. Slično vrijedi i za C++ Builder koji u stopu prati novitete dostupne u Delphiju. Također, C++ Builder danas omogućuje korištenje i programskog kôda napisanog u Delphiju (ali ne i obrnuto) čime se unutar C++ Buildera omogućilo korištenje komponenti inicijalno razvijanih za Delphi.

Dugo vremena su Delphi i C++ Builder bili usredotočeni isključivo na korištenje biblioteke VCL koja je bila ograničena samo na razvoj aplikacija za OS Windows. Da bi se omogućio razvoj aplikacija za nove operacijske sustave i uređaje prvo je razvijen Kylix. To razvojno okruženje vizualno je bilo identično Delphiju i C++ Builderu, ali je radilo u operacijskom sustavu Linux (distribucije Mandrake, Red Hat i SuSE) te je omogućavalo razvoj aplikacija za Linux korištenjem CLX biblioteke. [2] Međutim, nakon tri inačice Kylix se više nije razvijao zbog slabog interesa programera.

Nakon promjene vlasnika, RAD Studio alati počinju razvijati podršku za nove operacijske sustave i uređaje pomoću novorazvijene biblioteke FireMonkey. Osim za Windows, ova biblioteka podržava razvoj aplikacija za operacijske sustave OS X, iOS, Android i Linux, dok je biblioteka VCL i dalje ograničena samo na razvoj Windows aplikacija. Unatoč širokoj podršci FireMonkey biblioteke za razne platforme i uređaje još uvijek je riječ o relativno novoj biblioteci koja je tek u razvoju. Proizvođači komponenti još uvijek ne nude toliko široku podršku kao što je trenutno prisutna za VCL biblioteku, te je zato pri razvoju Windows aplikacija i dalje najčešće korištena VCL biblioteka.

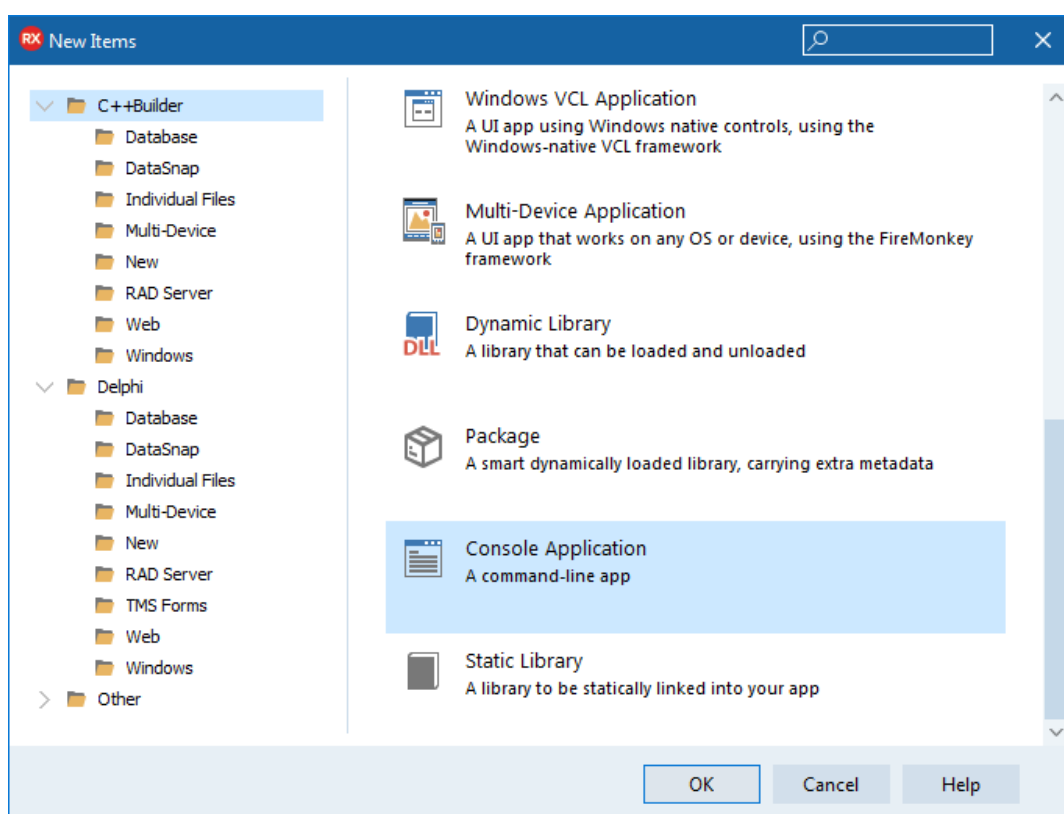
U žarištu sadržaja ove literature bit će razvijanje Windows aplikacija, servisa i biblioteka, a na tom području Delphi i C++ Builder nude širok izbor. Primjerice,

- aplikacije komandne linije
- aplikacije s grafičkim sučeljem (*VCL Forms*)
- aplikacije za rad s jednim ili više dokumenata (SDI, MDI)
- razvoj statičkih i dinamičkih biblioteka
- Windows servisi
- web servisi (klijent/server aplikacije SOAP i REST)
- mrežne klijent/server aplikacije (TCP/IP bazirani protokoli, UDP...)
- razvoj klijent aplikacija za razne baze podataka (Access, SQL Server, Oracle...)
- itd.

Aplikacije s grafičkim sučeljem danas su najčešći tipovi aplikacija, a u kombinaciji s njima za rad s podacima vrlo često razvijaju se i koriste web servisi i klijent aplikacije za obrađivanje sadržaja baza podataka.

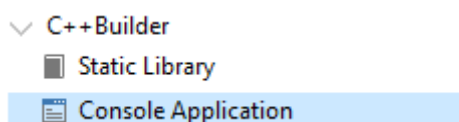
### 1.3. Aplikacije komandne linije

Projekti se unutar C++ Buildera kreiraju na vrlo sličan način, neovisno o njihovom tipu. Primjerice, aplikacije komandne linije (vrlo često zvane i „konzolne aplikacije“) koriste tekstualno korisničko sučelje, a najpogodnije su za manje projekte u kojima nema potrebe za razvijanjem grafičkog korisničkog sučelja. Da bismo kreirali projekt (aplikaciju) komandne linije u glavnom izborniku potrebno je odabrati stavku *File / New / Other...*, nakon čega se pojavljuje dijalog kojeg prikazuje Slika 1.3.1.



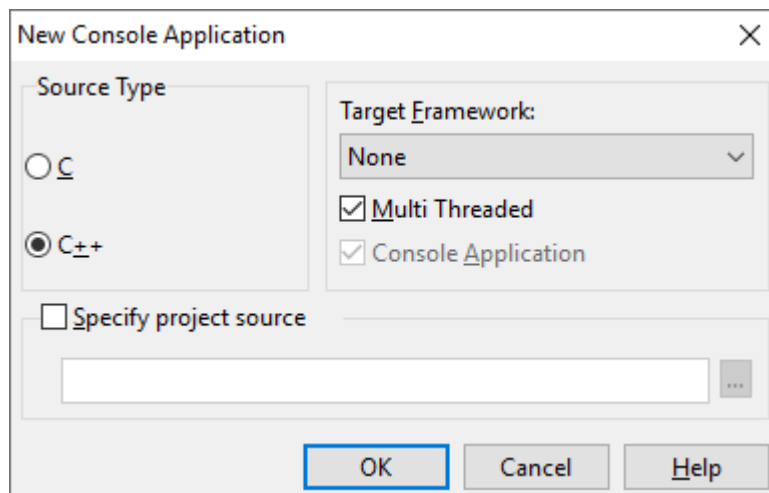
Slika 1.3.1. Kreiranje novog projekta u C++ Builderu

Na lijevoj strani prikazanog dijaloga nalaze se tipovi projekata grupirani po razvojnom okruženju i području primjene, a aplikaciju komandne linije u C++ Builderu kreiramo odabirom stavki sa slike (Slika 1.3.1).



Slika 1.3.2. Kreiranje novog projekta u C++ Builderu (glavna paleta)

Aplikaciju komandne linije moguće je alternativno kreirati i korištenjem glavne palete s alatima (Slika 1.3.2) koja je vidljiva s desne strane nakon pokretanja C++ Buildera.



*Slika 1.3.3. Postavke nove aplikacije komandne linije*

Ovisno o tipu projekta (aplikacije) nekada će pri njegovom kreiranju biti potrebno navesti dodatne postavke. Primjerice, za slučaju aplikacije komandne linije moguće je odrediti tip izvornog kôda (C ili C++), ciljani razvojni okvir (niti jedan, Visual Component Library ili FireMonkey), te podržava li aplikacija višedretvenost (Slika 1.3.3). Podrazumijevano, za novu aplikaciju komandne linije stvara se nova .c ili .cpp datoteka za pisanje izvornog kôda, no također je moguće iskoristiti i već postojeću datoteku (stavka „specify project source“).

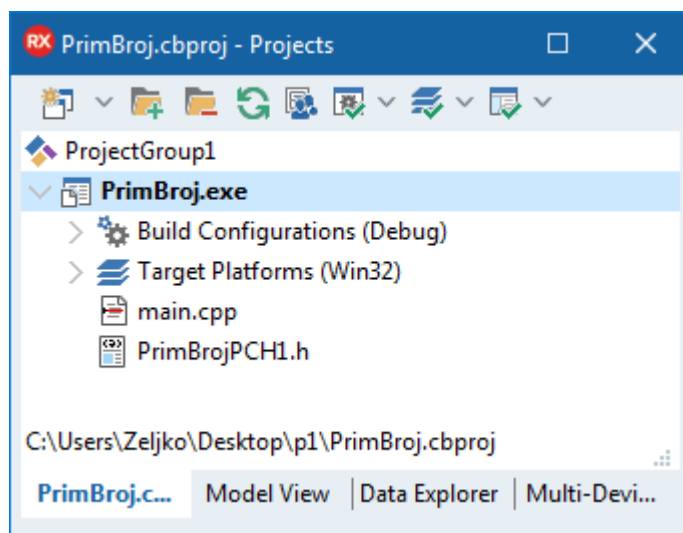
Nakon kreiranja projekta (aplikacije) komandne linije automatski se kreiraju tri datoteke:

- datoteka projekta (.cbproj)
- datoteka izvornog kôda (.c, .cpp)
- datoteka prekompiliranog zaglavlja (.h)

Datoteka projekta sadrži sve bitne informacije koje su razvojnem okruženju potrebne za učitavanje i prevođenje izvornog kôda. Primjerice, u datoteci projekta nalazi se popis svih datoteka izvornog kôda koje se koriste u projektu, popis biblioteka i komponenti potrebnih za prevođenje programskog kôda, pa čak i informacije o inačici razvojnog okruženja u kojem se projekt prvobitno kreirao.

Datoteka izvornog kôda automatski se kreira ukoliko nismo odabrali želimo li koristiti neku već postojeću .c ili .cpp datoteku. Svaki C++ Builder projekt sadrži najmanje jednu datoteku izvornog kôda u kojoj treba biti smještena funkcija *main*, a po potrebi moguće je dodati i koristiti više datoteka izvornog kôda.

Već dugo vremena C++ Builder za svaki projekt automatski izrađuje i datoteku prekompiliranog zaglavlja. Ova datoteka predstavlja poseban oblik datoteke zaglavlja koja se prevodi vrlo mali broj puta, čime doprinosi brzini prevođenja ostatka projekta. Naime, datoteka prekompiliranog zaglavlja prvi puta se prevodi prilikom prvog prevođenja novog projekta, a pri svakom sljedećem prevođenju projekta prevoditelj preskače ovu datoteku ukoliko u njoj nisu napravljene izmjene. Na ovaj način može se značajno ubrzati prevođenje većih projekata, tj. onih koji koriste veliki broj datoteka zaglavlja. [3]

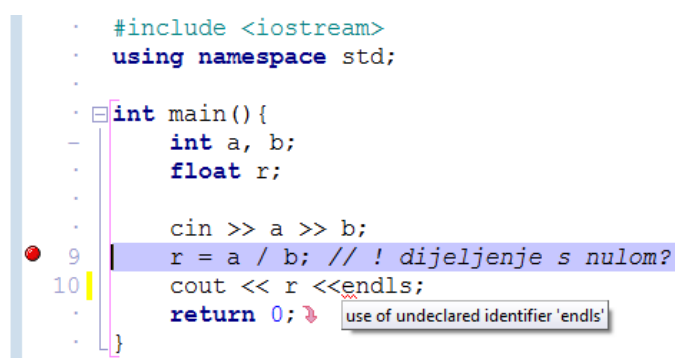


Slika 1.3.4. Pregled projektnih datoteka

Kreirane datoteke još uvijek nisu pohranjene na disku, pa je to potrebno napraviti korištenjem stavke *File / Save All*. Nakon toga korištenjem stavke *View / Tool Windows / Projects* moguće je pregledati od kojih se sve datoteka sastoji naš projekt, a po potrebi dodati nove i izbrisati postojeće datoteke iz projekta (Slika 1.3.4). Ostale stavke ovog dijaloga, poput *Data Explorer* i *Multi-Device Preview*, nisu dostupne za aplikacije komandne linije.

## 1.4. Uređivač programskog kôda

C++ Builder sadrži mnoštvo alata i značajki koje omogućavaju kvalitetno pisanje, uređivanje i analizu programskog kôda. Primjerice, ugrađeni uređivač programskog kôda (eng. *code editor*) automatski označava najvažnije dijelove kôda i ključne riječi drukčijim bojama, simbolima i linijama, a također dopušta i grupiranje programskih odsječaka po postojećim blokovima, funkcijama, klasama itd. (Slika 1.4.1). Uređivač također podržava statičku analizu programskog kôda tokom pisanja (podržano od inačice 10.3), refaktoriranje programskog kôda i još mnoge druge mogućnosti.



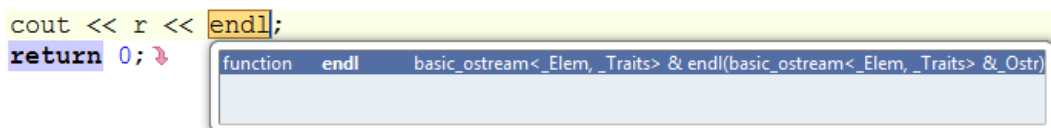
```

1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int a, b;
6      float r;
7
8      cin >> a >> b;
9      r = a / b; // ! dijeljenje s nulom?
10     cout << r << endl;
11     return 0;
12 }
```

A warning message is displayed: "use of undeclared identifier 'endl'" pointing to the `endl` in line 10.

Slika 1.4.1. Uređivač programskog kôda u C++ Builderu

Jedna od najkorisnijih značajki uređivača je IntelliSense. Koristi za brže dovršavanje pisanja programskog kôda (eng. *code completion*) prilikom čega pruža informacije o dostupnim ključnim riječima, imenima klasa i funkcija te postojećim parametrima (Slika 1.4.2).



```

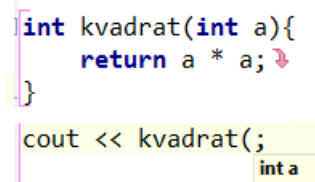
cout << r << endl;
return 0;
```

A dropdown menu is shown with the following options: `function`, `endl`, and `basic_ostream<_Elem, _Traits> & endl(basic_ostream<_Elem, _Traits> & Ostr)`.

Slika 1.4.2. IntelliSense značajka

Ova značajka podrazumijevano je vidljiva automatski prilikom pisanja određenih segmenata programskog kôda, ali ju je moguće i u bilo kojem trenutku aktivirati kombinacijom tipki CTRL + SPACE. Ipak, IntelliSense neće se pojaviti ukoliko postoje greške u trenutnoj ili prethodnim linijama programskog kôda.

Osim IntelliSense značajke, uređivač nudi još jedan oblik pomoći pri pisanju poziva funkcija korištenjem kombinacije tipki CTRL + SHIFT + SPACE. Tada se dobije opis parametara pozivajuće funkcije (Slika 1.4.3).

The screenshot shows a C++ code editor. A function definition is visible: `int kvadrat(int a){ return a * a; }`. Below it, a function call `cout << kvadrat(;` is being typed. A tooltip box appears next to the opening parenthesis, displaying `int a`, which represents the parameter of the `kvadrat` function.

Slika 1.4.3. Opis parametara funkcije

Uređivač podržava i refaktoriranje programskog kôda korištenjem stavke glavnog izbornika *Refactor / Rename*. Varijabloma je moguće automatski promijeniti imena, imajući uvid u sve lokacije tih varijabli u programskom kôdu te s mogućnošću uključivanja i isključivanja refaktoringa varijabli na pojedinim lokacijama.

Mogućnosti uređivača mogu se dodatno poboljšati i proširiti instalacijom *CnWizards* paketa alata koji je dostupan i iz repozitorija *GetIt Package Manager*. Ovaj komplet alata nudi desetke dodatnih pomagala koji su, osim u uređivaču, vrlo korisni i pri radu s dizajnerom, dokumentacijom i drugim segmentima razvojnog okruženja.

## 1.5. Otkrivanje i ispravljanje grešaka

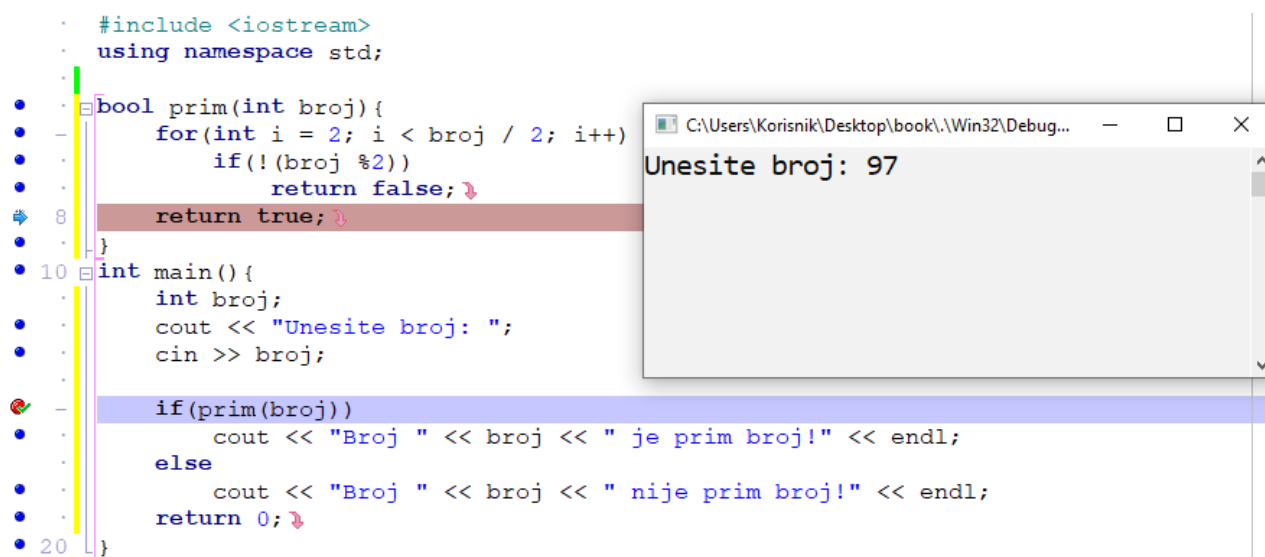
Sastavni dio razvojnog okruženja C++ Builder je i pomoćna aplikacija za otkrivanje grešaka (eng. *debugger*). Vrlo često se koristi kada je potrebno otkriti razloge rušenja razvijanih aplikacija ili kada one ne daju očekivani rezultat. Tada je pomoću aplikacije za otkrivanje grešaka problematične aplikacije moguće izvršavati korak po korak i pri tome pratiti pojedina stanja i vrijednosti varijabli te u konačnici otkriti problem.

Da bi se aplikacija za otkrivanje grešaka mogla koristiti, razvijane aplikacije potrebno je pokrenuti u načinu rada za otkrivanje grešaka (eng. *debug mode*) korištenjem stavke glavnog izbornika *Run / Run (F9)*. Prije pokretanja razvijane aplikacije na ovakav način moguće je odrediti i prekidne točke (eng. *breakpoints*), a tijekom rada izvršavati ju korak po korak korištenjem stavki poput:



- *Step over*
- *Trace into*
- *Trace to next source line*
- *Run to cursor*

Osim ovih stavki, unutar *Run* izbornika moguće je koristiti i stavke *Evaluate / Modify...*, *Add Watch...* itd., koje omogućuju praćenje specifičnih varijabli te izravnu izmjenu njihovih vrijednosti tokom rada aplikacije.



Slika 1.5.1. Korištenje aplikacije za otkrivanje grešaka

Korištenje aplikacije za provjeru grešaka moguće je demonstrirati programskim kôdom na prethodnoj slici (Slika 1.5.1). Prekidna točka nalazi se na liniji br. 15 (poziv funkcije *prim*), a prilikom pokretanja aplikacije za provjeru grešaka testna aplikacija pauzira s daljnjim izvršavanjem upravo na toj liniji. U tom trenutku se korištenjem stavke *Run / Trace Into* (*F7*) ulazi unutar tijela funkcije *prim*, gdje se naredbe u toj funkciji dalje izvršavaju jedna po jedna korištenjem stavke *Run / Step Over* (*F8*). U konačnici, vidljivo je koji tijek izvršavanja naredbi u funkciji *prim* za ulaznu vrijednost 97 rezultira povratnom vrijednosti *true*.

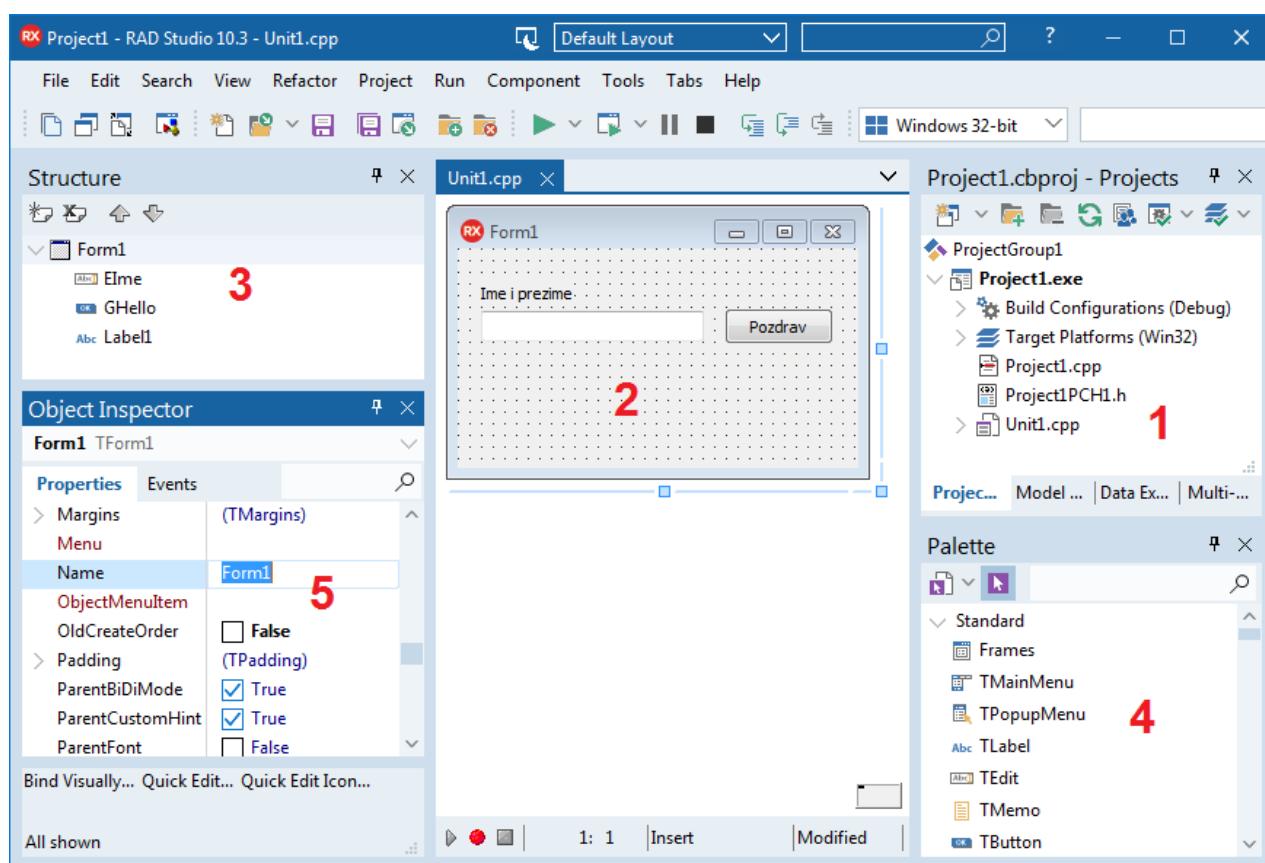
Na sličan način moguće je otkriti situacije koje uzrokuju rušenje programa (pokušaj pristupa nedostupnoj ili zaštićenoj memorijskoj lokaciji, korištenje nepodržanih instrukcija procesora itd.



## 2. Aplikacije s grafičkim korisničkim sučeljem

### 2.1. Windows VCL aplikacije

Od svojih samih početaka C++ Builder koristi VCL (*Visual Component Library*) biblioteku za razvoj aplikacija s grafičkim korisničkim sučeljem. Ova biblioteka nudi najširi spektar mogućnosti (veliki broj postojećih komponenti, veliku podršku zajednice, dostupnost biblioteka i komponenti trećih strana itd.), ali je zato ograničena samo na razvoj Windows aplikacija. Da bismo kreirali *Windows VCL* aplikaciju u glavnom izborniku potrebno je odabrati stavku *File / New / Other...*, a u narednom dijalogu (Slika 1.3.1) odabrati stavku *Windows VCL Application*.



Slika 2.1.1. Nova Windows VCL aplikacija

Kreiranjem nove *Windows VCL* aplikacije automatski se kreiraju sljedeće datoteke (Slika 2.1.1, oznaka 1);

- datoteka projekta (<ime\_projekta>.cbproj)

- glavna datoteka izvornog kôda (<ime\_projekta>.cpp)
- datoteka prekompiliranog zaglavlja (<ime\_projekta>PCH1.h)

Za svaku novu *Windows VCL* aplikaciju automatski se kreira i jedan dijalog (forma) koji predstavlja glavno sučelje aplikacije (Slika 2.1.1, oznaka 2). Po potrebi je moguće dodati i nove dijaloge korištenjem stavke *File / New / VCL Form – C++ Builder*, a za svaki od njih kreiraju se tri datoteke;

- datoteka zaglavlja (<ime\_dijaloga.h)
- implementacijska datoteka (<ime\_dijaloga.cpp)
- opisna datoteka (ime\_dijaloga.dfm)

Kreiranjem novog dijaloga kreiramo novu derivaciju klase *TForm*. Njena definicija nalazi se u datoteci zaglavlja dijaloga (.h), a implementacija metoda u .cpp datoteci. Opisna (.dfm) datoteka sadrži opise svih komponenti koji se koriste u dijalogu, a to uključuje podatke poput širine, visine, naslova itd. (Primjer 2.1.1).

#### Primjer 2.1.1. Sadržaj DFM datoteke

```

object Form1: TForm1
    Left = 0
    Top = 0
    Caption = 'Form1'
    ClientHeight = 151
    ClientWidth = 275
    Color = clBtnFace
    Font.Charset = DEFAULT_CHARSET
    Font.Color = clWindowText
    Font.Height = -11
    Font.Name = 'Tahoma'
    Font.Style = []
    OldCreateOrder = False
    PixelsPerInch = 96
    TextHeight = 13
    object Label1: TLabel
        Left = 16
        Top = 24
        Width = 63
        Height = 13
        Caption = 'Ime i prezime'
    end
    object EIme: TEdit
        Left = 16
        Top = 43
        Width = 153
        Height = 21
        TabOrder = 0
    end
    object GHello: TButton
        Left = 183
        Top = 41
        Width = 75
        Height = 25
        Caption = 'Pozdrav'
        TabOrder = 1
    end
end

```

Na osnovu sadržaja .dfm datoteke C++ Builder će prilikom učitavanja projekta za svaki dijalog i njegove komponente odmah moći inicijalizirati vrijednosti svojstava (eng. *properties*). Inače, popis komponenti je vidljiv i u dijalogu razvojnog okruženja *Structure* (Slika 2.1.1, oznaka 3), a svojstva pojedinih komponenti moguće je pregledati i inicijalizirati pomoću objektnog inspektora (Slika 2.1.1, oznaka 5).

Uspješnim prevođenjem *Windows VCL* aplikacije stvorit će se izvršna (.exe) datoteka. Međutim, njena veličina iznimno je mala (par stotina KB) jer početne postavke svake *Windows VCL* aplikacije definiraju dinamičko povezivanje s RTL (*Run-Time Library*) bibliotekama te povezivanje s vanjskim paketima komponenti. Zbog toga će dotična aplikacija na nekom drugom računalu (na kojemu nije instaliran C++ Builder) pri pokretanju tražiti dodatne .dll i .bpl biblioteke koje je tada potrebno distribuirati uz izvršnu (.exe) datoteku. U protivnom, aplikacija se neće moći uspješno pokrenuti i izvršavati na tim računalima.

Da bismo to izbjegli potrebno je u *Project / Options... / C++ Linker* isključiti stavku *Use dynamic RTL*. Također, potrebno je isključiti i stavku *Build with runtime packages* koja se nalazi u dijelu postavki projekta *Packages / Runtime Packages*. [4] Pri sljedećem prevođenju aplikacije, veličina izvršne datoteke znatno će porasti (preko nekoliko desetaka MB) jer će se sada u njoj nalaziti sav programski kôd i resursi koji su inače bili u tim drugim bibliotekama i paketima. Međutim, sada je izvršna datoteka samostalna i ne treba nikakve dodatne biblioteke za svoje pokretanje i izvršavanje te se može uspješno pokrenuti na bilo kojem računalu s OS Windows.

## 2.2. Dijalozi za prikaz poruka

Dijalozi se koriste kao sučelja za interakciju s korisnikom. Njihov izgled ovisi o vrsti interakcije (prikaz poruka, unos podataka i sl.), a VCL biblioteka nudi mogućnost kreiranja novih ili korištenja nekoliko tipova već postojećih (ugrađenih) dijaloga za prikaz poruka i unos podataka.

Ako je riječ o ugrađenim dijalogima za prikaz poruka moguće je koristiti pristupe poput:

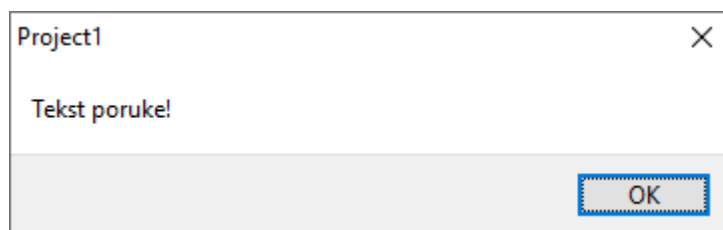
- funkcija *ShowMessage*

- funkcija *MessageBox*
- funkcija *MessageDlg*
- komponenta *TTaskDialog*

Funkcija *ShowMessage* predstavlja najjednostavniji oblik dijaloga za prikaz poruka. Ima tek jedan parametar koji određuje tekst poruke. Primjerice, izvršavanjem naredbe

```
ShowMessage("Tekst poruke!");
```

kao rezultat dobivamo sljedeći dijalog (Slika 2.2.1).



Slika 2.2.1. Korištenje funkcije *ShowMessage*

Ova funkcija nema mogućnost odabira naslova (eng. *caption*) poruke, odabira gumbi, ikona itd., te je prvenstveno namijenjena samim programerima prilikom razvijanja i ispitivanja rada aplikacije. Za prikaz poruka krajnjem korisniku prikladnije je koristiti funkciju *MessageBox*.

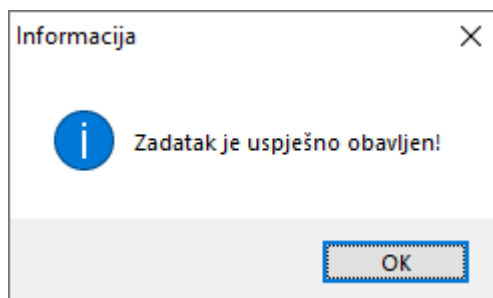
Opći oblik funkcije *MessageBox* izgleda na sljedeći način:

```
Application->MessageBox(poruka, naslov, gumbi | ikona | podrazumijevani_gumb);
```

Prilikom poziva ove funkcije tekst poruke i naslov su obavezni, dok gumb(i), ikona i podrazumijevani gumb (gumb u fokusu) ne moraju biti određeni. U slučaju da gumb(i) nisu određeni, podrazumijevano se pojavljuje samo gumb *OK*. Tako se izvršavanjem naredbe

```
Application->MessageBox(L"Zadatak je uspješno obavljen!",  
                        L"Informacija",  
                        MB_OK|MB_ICONINFORMATION);
```

pojavljuje sljedeći dijalog (Slika 2.2.2);



Slika 2.2.2. Korištenje funkcije `MessageBox` (prikaz informacije)

Gumbi, ikone i podrazumijevani gumb određuju se pomoću cjelobrojnih konstanti. Tako je za gumb `OK` potrebno koristiti konstantu `MB_OK`, dok za ikonu informacije konstantu `MB_ICONINFORMATION` (Tablica 2.2.1).

Konstanta	Značenje
<code>MB_OK</code>	Gumb OK
<code>MB_OKCANCEL</code>	Gumbi OK i Cancel
<code>MB_ABORTRETRYIGNORE</code>	Gumbi Abort, Retry i Ignore
<code>MB_RETRYCANCEL</code>	Gumbi Retry i Cancel
<code>MB_YESNO</code>	Gumbi Yes i No
<code>MB_YESNOCANCEL</code>	Gumbi Yes, No i Cancel
<code>MB_HELP</code>	Gumb Help
<code>MB_ICONINFORMATION</code>	Ikona za informaciju
<code>MB_ICONWARNING</code>	Ikona za upozorenje
<code>MB_ICONQUESTION</code>	Ikona za pitanje
<code>MB_ICONSTOP</code>	Ikona za grešku
<code>MB_DEFBUTTON1</code>	Prvi gumb je podrazumijevani
<code>MB_DEFBUTTON2</code>	Drugi gumb je podrazumijevani
<code>MB_DEFBUTTON3</code>	Treći gumb je podrazumijevani
<code>MB_DEFBUTTON4</code>	Četvrti gumb je podrazumijevani

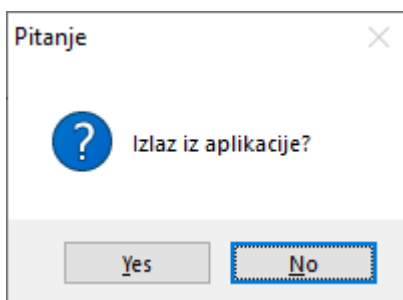
Tablica 2.2.1. Cjelobrojne konstante gumbi i ikona funkcije `MessageBox` [5]

Za neke od navedenih konstanti postoje i alternativni nazivi. Tako je umjesto konstante *MB\_ICONSTOP* moguće koristiti *MB\_ICONERROR* i *MB\_ICONHAND* konstante jer sve tri imaju istu numeričku vrijednost.

#### Primjer 2.2.1. Korištenje funkcije *MessageBox* (upit korisniku)

```
int odgovor = Application->MessageBox(L"Izlaz iz aplikacije?", L"Pitanje",
    MB_YESNO | MB_ICONQUESTION | MB_DEFBUTTON2);
if (odgovor == IDYES)
    Application->MessageBox(L"Odabrali ste: DA!", L"Informacija",
    MB_OK | MB_ICONINFORMATION);
else
    Application->MessageBox(L"Odabrali ste: NE!", L"Informacija",
    MB_OK | MB_ICONINFORMATION);
```

Funkciju *MessageBox* može se koristiti i za postavljanje upita korisnicima pri čemu se korisnikov odgovor (odabrani gumb) vraća kao povratna vrijednost funkcije (Primjer 2.2.1). Ovisno o odabranom gumbu funkcija *MessageBox* može vratiti jednu od sljedećih vrijednosti (konstanti): *IDOK*, *IDYES*, *IDNO*, *IDCANCEL*, *IDIGNORE*, *IDCONTINUE*, *IDABORT*, *IDRETRY* i *IDTRYAGAIN*.



Slika 2.2.3. Korištenje funkcije *MessageBox* (upit korisniku)

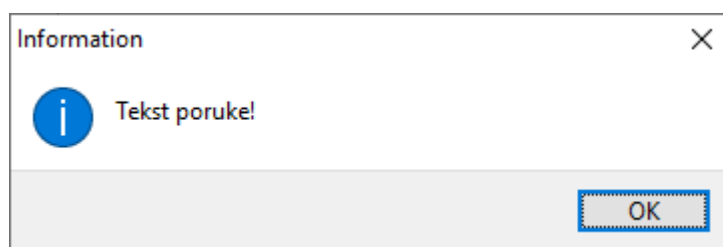
Tako Primjer 2.2.1 korisnika traži potvrda o izlasku iz aplikacije (Slika 2.2.3). Prikazani dijalog ima gumbе *YES* i *NO* (*MB\_YESNO*), ikonu upitnika (*MB\_ICONQUESTION*), te gumb *NO* kao podrazumijevani gumb (*MB\_DEFBUTTON2*). Ovisno o odabranom gumbu funkcija *MessageBox* vratit će vrijednost *IDYES* ili *IDNO*, te će se sukladno tome u nastavku programa ispisati odgovarajuća poruka.



Za razliku od funkcije *MessageBox*, koja je jedna od *WinAPI* funkcija, poruke je moguće prikazivati i korištenjem funkcije *MessageDlg*. Ova je funkcija alternativa funkciji *MessageBox* i dostupna je samo u VCL biblioteci. Tako se izvršavanjem naredbe

```
MessageDlg(L"Tekst poruke!", mtInformation, TMsgDlgButtons() << mbOK, 0);
```

pojavljuje sljedeći dijalog (Slika 2.2.4);



Slika 2.2.4. Korištenje funkcije *MessageDlg*

Funkcija *MessageDlg* kao povratnu vrijednost vraća odabrani gumb korištenjem konstanti *mrOK*, *mrCancel*, *mrYes* itd. [6]. Postoji i druga inačica ove funkcije (*MessageDlgPos*) koja dopušta i specifikaciju koordinata na kojima se dijalog treba pojaviti.

VCL biblioteka sadrži i *TTaskDialog* komponentu, koju se također može koristiti kao dijalog za prikaz podataka. Ova komponenta nudi vrlo širok izbor mogućnosti, no zbog toga ju je teže konfigurirati i koristiti (Primjer 2.2.2).

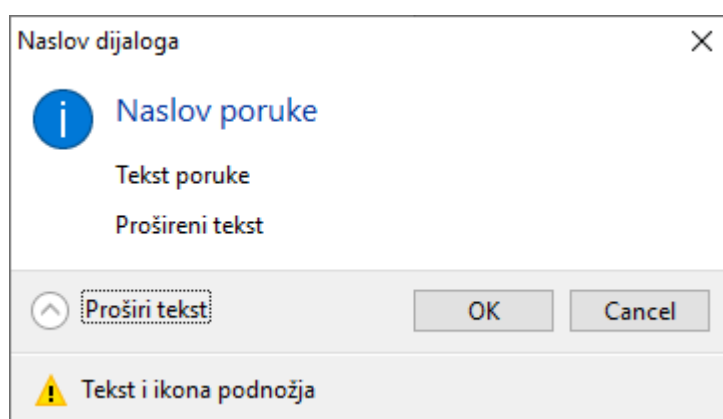
#### Primjer 2.2.2. Konfiguriranje i korištenje *TTaskDialog* komponente

```
// na dijalog (formu) prethodno je potrebno dodati TTaskDialog
// (TaskDialog1) komponentu
TaskDialog1->Caption = "Naslov dijaloga";
TaskDialog1->Title = "Naslov poruke";
TaskDialog1->Text = "Tekst poruke";
TaskDialog1->ExpandButtonCaption = "Proširi tekst";
TaskDialog1->ExpandedText = "Prošireni tekst";
TaskDialog1->FooterText = "Tekst i ikona podnožja";
TaskDialog1->FooterIcon = tdiWarning;

// dodaj gumbe OK i Cancel
TaskDialog1->CommonButtons << tcbOk << tcbCancel;
TaskDialog1->Execute();
```

```
// provjera odabranog gumba
if(TaskDialog1->ModalResult == mrOk)
    ShowMessage("Odabran je gumb 'OK'");
else
    ShowMessage("Odabran je gumb 'Otkazi'");
```

Komponenta *TTaskDialog* osim naslova i poruke nudi mogućnost specifikacije proširenog teksta, teksta podnožja, ikone podnožja itd. Sva ta svojstva (eng. *properties*) moguće je inicijalizirati i korištenjem objektnog inspektora.



Slika 2.2.5. Korištenje *TTaskDialog* komponente

Rezultat izvršavanja prikazuje Slika 2.2.5. Osim prikazanih mogućnosti, *TTaskDialog* komponenta omogućuje i dodavanje gumbi s proizvoljnim naslovima (Primjer 2.2.3), korištenje komandnih linkova, drugih komponenti itd.

#### **Primjer 2.2.3. *TTaskDialog* - Dodavanje gumba s proizvoljnim naslovom**

```
TTaskDialogBaseButtonItem* gumbOtkazi = TaskDialog1->Buttons->Add();
gumbOtkazi->Caption = "Otkazi";
gumbOtkazi->ModalResult = mrCancel;
```

Za prikaz poruka moguće je dodatno koristiti i funkcije poput *TaskMessageDlg* i *TaskMessageDlgPos*, a u konačnici kreirati i vlastite dijaloge.

## 2.3. Dijalozi za unos podataka

Najčešće se za unos podataka kreiraju i koriste zasebni dijalozi (forme) koji pomoću komponenti usmjeravaju i kontroliraju korisnikov unos. No, razvoj dodatnog dijaloga zahtjeva određeno vrijeme, a zbog dodatnih datoteka dijaloga (.h, .cpp, .dfm) produžuje se i vrijeme prevođenja projekta. Takav bi pristup trebalo izbjegavati kada god je moguće, a pogotovo kada je riječ samo o unosu teksta. U takvim slučajevima VCL biblioteka nudi mogućnost korištenja funkcija *InputBox* i *InputQuery*.

Opći oblik funkcije *InputBox* izgleda na sljedeći način:

```
UnicodeString InputBox( UnicodeString Naslov,
                        UnicodeString Poruka,
                        UnicodeString PodrazumijevanaVrijednost
                        );
```

Pozivom ove funkcije prikazuje se dijalog s odabranim naslovom i porukom, te se od korisnika očekuje unos podataka (teksta). Podrazumijevana vrijednost tog podatka nije obavezna, ali ako je određena već se nalaziti upisana u polju za unos znakova.

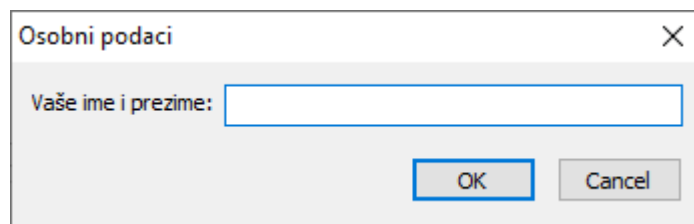
Prikazani dijalog sadržavat će gumb *OK* i *Cancel*. Prilikom odabira gumba *OK* uneseni podatak vraća se kao povratna vrijednost funkcije, a u protivnom vraća se prazan niz znakova.

### **Primjer 2.3.1. Korištenje funkcije *InputBox***

```
UnicodeString ImePrezime;
ImePrezime = InputBox("Osobni podaci", "Vaše ime i prezime:", "");
if(!ImePrezime.IsEmpty())
    ShowMessage("Ime i prezime osobe: " + ImePrezime);
else
    ShowMessage("Korisnik nije unio tražene podatke!");
```

Prikazani Primjer 2.3.1 korištenjem funkcije *InputBox* od korisnika traži unos imena i prezimena. Ukoliko je korisnik odabrao gumb *OK* (Slika 2.3.1) u varijablu *ImePrezime* pohranjuje se uneseni tekst, a u protivnom (odabran je gumb *Cancel*) ta varijabla bit će

prazna. Upravo po tome moguće je znati koji gumb je odabran, te sukladno tome izvršiti odgovarajuće akcije.



Slika 2.3.1. Dijalog *InputBox*

Također, potrebno je primijetiti da se u prikazanom primjeru uneseni tekst pohranjuje u varijablu tipa *UnicodeString*. To je poseban tip varijable za rad sa znakovnim nizovima koji podržava najširi spektar međunarodnih slova i znakova. Tako je u varijablu (objekt) tog tipa moguće pohraniti tekst koji sadrži sva slova naše abecede (uključujući i slova č,ć,ž,š,đ,Č,Ć,Ž,Š i Đ), kao i slova te posebne znakove iz drugih jezika. Dužina znakovnog niza ovog tipa ograničena je samo količinom dostupne radne memorije.

U C++ Builderu tip *UnicodeString* implementiran je kao klasa koja ima metode poput *IsEmpty*, *Trim*, *TrimLeft*, *TrimRight*, *Length*, *UpperCase*, *LowerCase*, *ToInt*, *ToDouble*, *Insert*, *Delete*, *SubString* itd. [7] U prošlosti C++ Builder najčešće je koristio alternative poput *AnsiString* i *WideString*, dok je danas *UnicodeString* općeprihvaćen kao podrazumijevani tip za rad sa znakovnim nizovima. Štoviše, u Delphiju se ovaj tip podatka koristi kao primitivni tip.

Alternativa funkciji *InputBox* je funkcija *InputQuery*. Ona također kao parametre ima naslov i poruku, ali umjesto podrazumijevane vrijednosti unesenog podatka (teksta) ima referencu na objekt u koji se treba pohraniti uneseni tekst. Opći oblik:

```
bool InputQuery( UnicodeString Naslov,
                UnicodeString Poruka,
                UnicodeString &UneseniTekst
                );
```

Razlika između tih dviju funkcija je i u povratnoj vrijednosti. Dok funkcija *InputBox* kao povratnu vrijednost vraća uneseni tekst, funkcija *InputQuery* vraća logičku vrijednost *true* (odabran je gumb *OK*) ili *false* (odabran je gumb *Cancel*; Primjer 2.3.2).

### Primjer 2.3.2. Korištenje funkcije *InputQuery*

```
UnicodeString ImePrezime;
if (InputQuery("Osobni podaci", "Vaše ime i prezime:", ImePrezime))
    ShowMessage("Ime i prezime osobe: " + ImePrezime);
else
    ShowMessage("Korisnik nije unio tražene podatke!");
```

Rezultat poziva obiju funkcija vizualno je identični dijalog (Slika 2.3.1), pa je na programeru da donese odluku koju od funkcija više preferira koristiti.

Ako nije riječ o unosu teksta, moguće je koristiti *TTaskDialog* komponentu. Iako se ova komponenta najčešće koristi za prikaz poruka njen dijalog može sadržavati i komponente poput radio gumbi i potvrdnog okvira (eng. *checkbox*). Upotrebom tih komponenti korisniku se daje mogućnost odabira jedne od više unaprijed ponuđenih stavki, što se također može iskoristiti kao jedna od mogućnosti za dobivanja podataka od krajnjeg korisnika.

### Primjer 2.3.3. Korištenje *TTaskDialog* komponente pri odabiru podataka

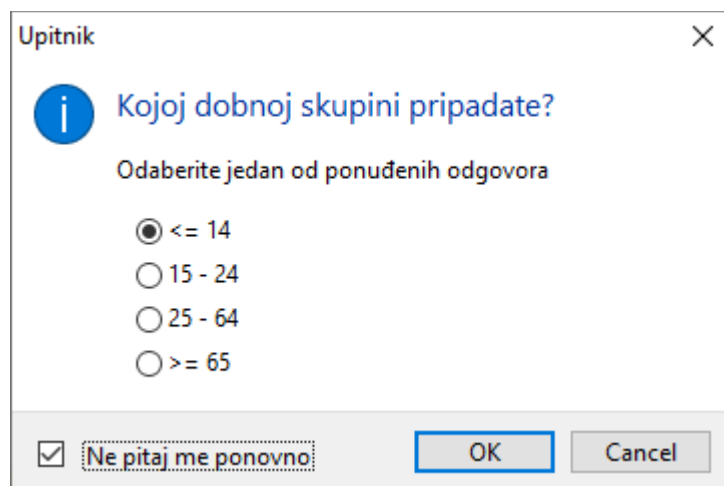
```
TaskDialog1->Caption = "Upitnik";
TaskDialog1->Title = "Kojoj dobnoj skupini pripadate?";
TaskDialog1->Text = "Odaberite jedan od ponuđenih odgovora";
TaskDialog1->CommonButtons << tcbOk;
// radio gumbi...
TTaskDialogBaseButtonItem* Izbor1 = TaskDialog1->RadioButtons->Add();
TTaskDialogBaseButtonItem* Izbor2 = TaskDialog1->RadioButtons->Add();
TTaskDialogBaseButtonItem* Izbor3 = TaskDialog1->RadioButtons->Add();
TTaskDialogBaseButtonItem* Izbor4 = TaskDialog1->RadioButtons->Add();
Izbor1->Caption = "<= 14";
Izbor2->Caption = "15 - 24";
Izbor3->Caption = "25 - 64";
Izbor4->Caption = ">= 65";
TaskDialog1->VerificationText = "Ne pitaj me ponovno";
TaskDialog1->Execute();
// odabrani radio gumb
ShowMessage("Odabrali ste izbor br. " +
```

```

        IntToStr (TaskDialog1->RadioButton->Index) );
// Provjera potvrdnog okvira (eng. checkbox)
if (TaskDialog1->Flags.Contains(tfVerificationFlagChecked))
    ShowMessage ("Nećemo Vas više kontaktirati");

```

U prikazanom primjeru komponenta *TTaskDialog* koristi se za dobivanje podatka o dobnoj skupini korisnika. Međutim, umjesto izravnog upisa starosti u polje za unos znakova korisnik ima mogućnost odabira dobne skupine korištenjem jednog od četiri radio gumba. Da bismo doznali koji je izbor (radio gumb) korisnik odabrao čitamo vrijednost podatkovnog člana *Index*, a za provjeru označenosti potvrdnog okvira provjeravamo postojanost zastavice *tfVerificationFlagChecked* (Primjer 2.3.3). Kako točno izgleda konačni dijalog prikazuje Slika 2.3.2.



Slika 2.3.2. Korištenje komponente *TTaskDialog* pri odabiru podataka

Sve demonstrirane funkcije i komponente za prikaz i unos podataka dovoljne su tek kada je riječ o zahtjevima niske razine. Za kompliciranije scenarije (npr. odabir ulaznih podataka iz liste, unos i validacija podataka po određenim pravilima itd.) programeri moraju kreirati i koristiti vlastite dijaloge.

## 2.4. Svojstva, događaji i metode

Komponente su građevni blokovi aplikacije koji se mogu višestruko upotrebljavati. Mogu biti vizualne i nevizualne, a grafička sučelja najčešće su kreirana upravo

kombiniranjem vizualnih komponenti (gumbi, liste, potvrdni okviri itd.). Ključne značajke svake komponente su njena svojstva (eng. *properties*), metode i događaji (eng. *events*).

Primjerice, ukoliko pretpostavimo da je *Mjenjač* komponenta vozila, njegovo svojstvo može biti *StupanjBrzine*. Također, ta komponenta može sadržavati metode *SetVisaBrzina* i *SetNizaBrzina* te može pratiti događaj *NaPromjenuBrzine* (dogodila se promjena brzine).

Svojstva su atributi, tj. opisni elementi komponenti poput *ime*, *visina*, *širina*, *boja* itd. Dostupni su u vrijeme dizajna (eng. *design-time*) korištenjem objektnog inspektora (Slika 2.1.1 - oznaka 5) i u vrijeme izvršavanja (eng. *runtime*) referenciranjem u programskom kôdu. Međutim, treba znati da komponente mogu sadržavati i attribute koji nisu svojstva, tj. attribute koji nisu dostupni u vrijeme dizajna. Naime, komponente mogu sadržavati veliki broj atributa, ali najčešće su samo najvažniji atributi dostupni kao svojstva kako bi se programerima omogućio pristup tim atributima i u vrijeme dizajna. Svi ostali atributi dostupni su isključivo referenciranjem u programskom kôdu.

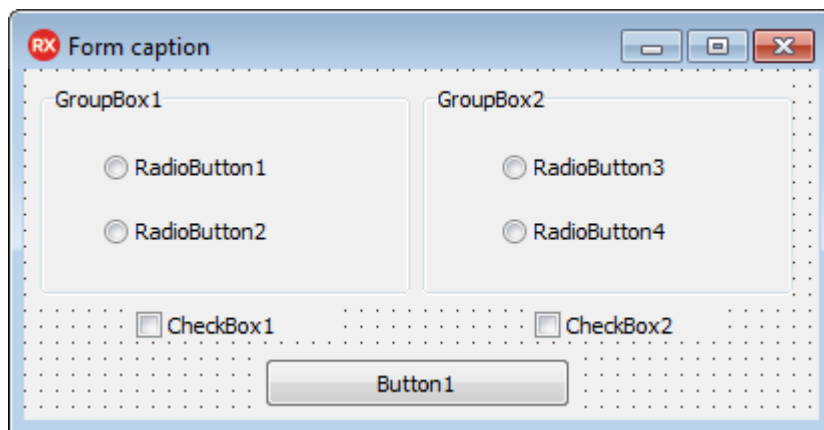
Pojedina metoda komponente predstavlja njenu funkcionalnost, a svaka funkcionalnost može ovisiti o vrijednosti pojedinih atributa. Primjerice, metoda *SetVisaBrzina* komponente *Mjenjač* može ubaciti u viši stupanj brzine tek kada zna kolika je trenutna vrijednost svojstva (atributa) *StupanjBrzine*.

Događaji predstavljaju akcije na koje aplikacija može odgovoriti izvršavanjem nekog programskog kôda. Tako je moguće prilikom svake promjene brzine (događaj *NaPromjenuBrzine*) izvršiti određenu metodu i sl.

## 2.5. VCL komponente

Razvoj grafičkog sučelja započinje praznim dijalogom, a on je derivacija klase *TForm*. Dijalozi nemaju puno ključnih svojstava, a neka od njih su *Name* (naziv objekta koji predstavlja dijalog), *Caption* (naslov), *Width* i *Height* (širina i visina) te svojstvo *Position* koje određuje inicijalnu poziciju dijaloga.

Od ključnih metoda tu su *Show* (interakcija s drugim dijalogima je dozvoljena), *ShowModal* (ne dozvoljava se interakcija s drugim dijalogima dok se ovaj ne zatvori) i *Hide*, a od najvažnijih događa koriste se *OnShow* (dijalog se prikazao) i *OnClose* (dijalog se zatvorio).



Slika 2.5.1. Dijalog, gumbi i potvrdni okviri

U datoteci zaglavlja dijaloga (.h) nalazi se definicija dijaloga, odnosno njegova derivacija *TForm* klase. Tako se za dijalog prikazan na slici (Slika 2.5.1) u njegovoj datoteci zaglavlja nalazi sljedeći sadržaj (Primjer 2.5.1):

#### Primjer 2.5.1. Sadržaj datoteke zaglavlja dijaloga

```
class TForm1 : public TForm{
__published:      // IDE-managed Components
    TGroupBox *GroupBox1;
    TGroupBox *GroupBox2;
    TRadioButton *RadioButton1;
    TRadioButton *RadioButton2;
    TRadioButton *RadioButton3;
    TRadioButton *RadioButton4;
    TCheckBox *CheckBox1;
    TCheckBox *CheckBox2;
    TButton *Button1;

    void __fastcall Button1Click(TObject *Sender);
private:      // User declarations
public:      // User declarations
    __fastcall TForm1(TComponent* Owner);
};

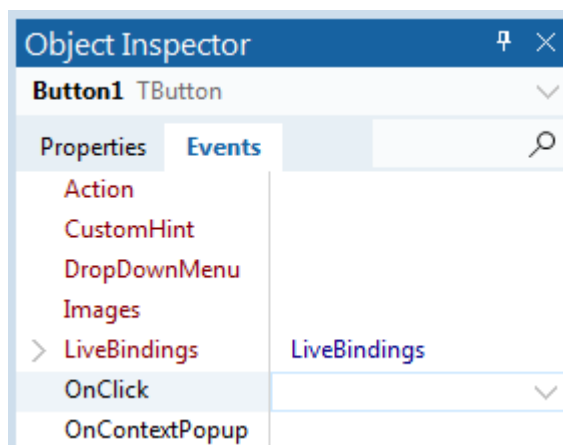
//-----
extern PACKAGE TForm1 *Form1;
```



Svaka komponenta na dijalogu predstavljena je pokazivačem odgovarajućeg tipa (*TGroupBox\**, *TRadioButton\**, *TButton\** itd.), a ime pokazivača je zapravo vrijednost svojstva *Name* pojedine komponente. Metode klase dijaloga najčešće predstavljaju dijelove kôda koji se trebaju izvršiti kao posljedica određenog događaja (npr. *Button1Click*), a po potrebi moguće je dodati i vlastite metode.

Kada se korištenjem dizajnera u dijalog dodaju nove komponente, sadržaj datoteke zaglavlja automatski se nadopunjuje odgovarajućim deklaracijama pokazivača. Slično vrijedi i prilikom praćenja novog događaja, a tada se prototip metode za obradu tog događaja automatski dodaje u datoteku zaglavlja.

Da bismo pratili neki događaj, potrebno je u dizajneru označiti željenu komponentu, a zatim u objektnom inspektoru odabrati stavku *Events* (Slika 2.5.2).



Slika 2.5.2. Objektni inspektor - praćenje događaja

Svaka komponenta ima svoj popis događaja koje je moguće pratiti, a jedan od najvažnijih događaja komponente *TButton* je *OnClick*. Dvostrukim klikom na prazno polje pokraj imena tog događaja automatski se kreira ime metode koja će se izvršavati prilikom svakog klika na taj gumb (npr. *Button1Click*). Njen prototip smješta se u datoteku zaglavlja dijaloga, a njena implementacija automatski će se otvoriti u uređivaču programskog kôda. Alternativno, umjesto dvostrukog klika na prazno polje programer može proizvoljno upisati ime metode za taj događaj ili u padajućem izborniku izabrati jednu od postojećih metoda koja se može iskoristiti za obradu tog događaja.

**Primjer 2.5.2. Gumbi i potvrdni okviri**

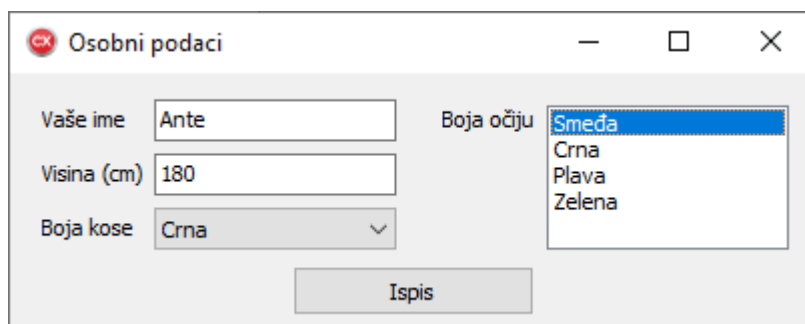
```

void __fastcall TForm1::Button1Click(TObject *Sender) {
    // je li označen CheckBox1?
    if(CheckBox1->Checked == true)
        ShowMessage("CheckBox1 je označen!");
    else
        ShowMessage("CheckBox1 nije označen!");
    // je li označen RadioButton1?
    if(RadioButton1->Checked == true)
        ShowMessage("RadioButton1 je označen!");
    else
        ShowMessage("RadioButton1 nije označen!");
}

```

Pretpostavimo da se klikom na gumb *Button1* (Slika 2.5.1) izvršava metoda *Button1Click* (Primjer 2.5.2). U tijelu te metode može se primijetiti jedno od glavnih svojstava komponenti *TCheckBox* i *TRadioButton*, odnosno svojstvo *Checked*. Čitanjem tog svojstva provjeravamo trenutno stanje tih komponenti, a u programskom kôdu moguće je vrijednost tog svojstva postaviti na vrijednost *true* ili *false* te time automatski promijeniti i stanja tih komponenti. Gumb (*TButton*) i komponente (*TCheckBox* i *TRadioButton*) podržavaju obradu događaja *OnClick*.

Komponenta *TGroupBox* prvenstveno služi za grupiranje *TRadioButton* komponenti. Naime, u isto vrijeme na dijalogu može biti označena samo jedna od *TRadioButton* komponenti. Da bismo omogućili odabir više *TRadioButton* komponenti u isto vrijeme potrebno ih je grupirati *TGroupBox* komponentama. Tada, iz svake grupe može biti označena po jedna *TRadioButton* komponenta.



Slika 2.5.3. Polja za unos znakova i liste

Za rad s tekstom najčešće se koristi *TEdit* komponenta (Slika 2.5.3). Da bismo pročitali korisnikov unos potrebno je pročitati vrijednost svojstva *Text*, a inicijalizacijom nove vrijednosti tog svojstva automatski se postavlja novi tekst u polje za unos znakova.

Navedena komponenta također ima i svojstvo *NumbersOnly* (vrijednost *true* ili *false*). Ukoliko je vrijednost ovog svojstva *true*, komponenta će u polje za unos znakova dopustiti samo unos brojčanih znakova, odnosno znamenki. Također, kada u polje za unos znakova želimo unositi skrivene lozinke moguće je koristiti svojstvo *PasswordChar*. Podrazumijevana vrijednost ovog svojstva je *#0* (uneseni tekst je vidljiv), a ukoliko tu vrijednost zamijenimo znakom *\** (zvjezdica) svi uneseni znakovi bit će skriveni zvjezdicama. Komponenta *TEdit* podržava obradu mnoštva tipova događaja, a jedan od najvažnijih događaja je *OnChange* (promijenio se sadržaj polja za unos znakova).

Alternativa komponenti *TEdit* je komponenta *TMemo*. Ona podržava višelinijski unos teksta te ima ugrađene metode *SaveToFile* i *LoadFromFile* kojima se dopušta pohrana i učitavanje sadržaja datoteka.

Liste (*TListBox*) i padajući popisi (*TComboBox*) vrlo su slični jer oboje liste popise stavki prilikom čega korisnik može odabrati jednu ili više od njih. Slika 2.5.3 prikazuje kako se komponenta *TListBox* koristi za odabir boje očiju korisnika, dok se za odabir boje kose koristi ista lista stavki, ali ovaj put pomoću komponente *TComboBox*. Da bismo odredili stavke u listi kod oba tipa komponenti koristimo svojstvo *Items*.

Komponenta *TListBox* podrazumijevano dopušta odabir samo jedne stavke iz liste, a pomoću svojstva *MultiSelect* (vrijednost *true* ili *false*) možemo omogućiti i odabir više stavki odjednom. S druge strane, komponenta *TComboBox* dopušta odabir isključivo jedne stavke. Da bismo provjerili odabranu stavku u komponenti *TListBox* možemo izvršiti sljedeći programski odsječak:

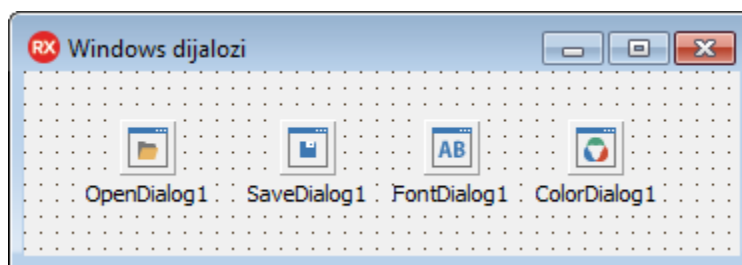
```
if(ListBox1->ItemIndex >= 0) // ukoliko je nešto odabrano...  
    ShowMessage(ListBox1->Items->Strings[ListBox1->ItemIndex]);
```

Svaka stavka u listi ima svoj indeks, a indeks odabrane stavke pohranjen je u svojstvu *ItemIndex*. Konačno, da bismo pročitali tekst označene stavke, potrebno je pročitati stavku

pod označenim indeksom (`Strings[ListBox1->ItemIndex]`). Od važnijih, komponenta *TListBox* podržava obradu događaja *OnClick*, a njega se može koristiti svaki put kada se klikne na komponentu, odnosno kada se odabere neka od stavki.

Da bismo doznali koju je stavku korisnik odabrao u komponenti *TComboBox*, možemo upotrijebiti isti pristup kao i s komponentom *TListBox*. Ipak, komponenta *TComboBox* sadrži i svojstvo *Text* čija je vrijednost upravo označena stavka. I komponenta *TComboBox* podržava obradu događaja *OnClick*, ali kod nje je prikladnije koristiti događaj *OnChange* (promijenila se odabrana stavka).

Osim prethodno spomenutih vizualnih komponenti, možemo koristiti i standardne Windows dijaloge poput *OpenFile*, *SaveFile*, *ChooseColor*, *ChooseFont* itd. Ti dijalozi dostupni su pomoću nevizualnih komponenti *TOpenDialog*, *TSaveDialog* itd. (Slika 2.5.4)



Slika 2.5.4. Windows dijalozi

Svaka od navedenih komponenti sadrži ključna svojstva poput *FileName* (odabrana datoteka u dijalogima *OpenDialog* i *SaveDialog*), *Font*, *Color* itd. Zajednička karakteristika svih ovih dijaloga jest da se pozivaju korištenjem metode *Execute*, koja vraća vrijednost *true* (korisnik je odabrao datoteku, font ili boju) ili *false* (korisnik je unutar dijaloga kliknuo gumb *Cancel*) (Primjer 2.5.3).

#### Primjer 2.5.3. Korištenje Windows dijaloga

```
void __fastcall TForm1::Button1Click(TObject *Sender){
    // OpenDialog
    OpenDialog1->DefaultExt = ".txt";
    OpenDialog1->Filter = "Tekstualne datoteke (*.txt)|*.TXT";
    // ukoliko je korisnik odabrao datoteku (nije kliknuo gumb Cancel)
    if(OpenDialog1->Execute() == true)
        ShowMessage(OpenDialog1->FileName);
}
```

```
// FontDialog
if(FontDialog1->Execute() == true){
    ShowMessage(FontDialog1->Font->Name);
    ShowMessage(FontDialog1->Font->Size);
}
// ColorDialog
if(ColorDialog1->Execute() == true)
    ShowMessage(ColorDialog1->Color);
}
```

Osim u vrijeme dizajna, instance VCL komponenti moguće je kreirati i tijekom rada aplikacije. U tom slučaju potrebno ih je dinamički alocirati korištenjem operatora *new* te dealocirati operatorom *delete* (Primjer 2.5.4). Alternativno, moguće je koristiti i pametne pokazivače poput *unique\_ptr*.

#### **Primjer 2.5.4. Dinamička alokacija VCL komponenti**

```
TButton* Gumb = new TButton(this);
Gumb->Parent = this;
Gumb->Caption = "GUMB";
Gumb->Left = 10;
Gumb->Top = 10;
Gumb->Width = 50;
Gumb->Height = 30;
ShowMessage("Gumb će sada biti dealociran!");
delete Gumb;
```

Deklaracija pokazivača trebala bi se nalaziti u prostoru sa što širim dosegom (primjerice, unutar deklaracije klase dijaloga), a nakon dinamičke alokacije potrebno je inicijalizirati vrijednost svojstva *Parent* (tko je roditelj komponente?). Primjerice, ukoliko se dvije komponente *TRadioButton* tipa nalaze unutar jedne *TGroupBox* komponente, tada je njihov roditelj upravo ta *TGroupBox* komponenta. Nadalje, roditelj te iste *TGroupBox* komponente je dijalog ili neka druga komponenta u kojoj je smještena.

Ovako alocirane instance VCL komponenti također mogu obrađivati i događaje, za što se koriste pokazivači na funkcije. Tako bi se izrazom `Gumb->OnClick = Button1Click;` definirala obrada događaja *OnClick* metodom *Button1Click*.

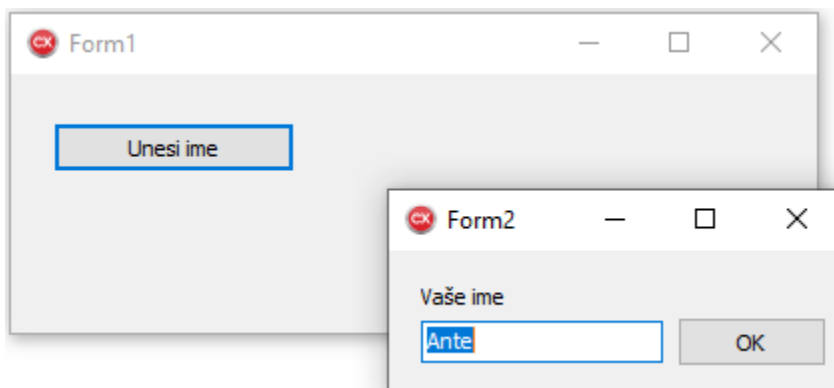
## 2.6. Komunikacija među dijalogima

Kreiranjem novog projekta Windows VCL aplikacije automatski se kreira i njen glavni dijalog (*Form1*). Da bismo u aplikaciju dodali novi dijalog (*Form2*) potrebno je odabrati stavku *File / New / VCL Form – C++ Builder*. Za svaki novi dijalog automatski se kreiraju njegove .h, .cpp i .dfm datoteke te jedino preostaje dizajnirati ga i pozvati iz nekog drugog dijaloga.

Da bi se iz dijaloga *Form1* mogao prikazati dijalog *Form2* potrebno je slijediti ove korake:

- 1) Pretprocesorskom naredbom *#include* uključiti datoteku zaglavlja dijaloga *Form2* u .h ili .cpp datoteci dijaloga *Form1*.
- 2) Prilikom klika na gumb (ili nekog drugog događaja) u dijalogu *Form1* pozvati metodu *Show* ili *ShowModal* dijaloga *Form2*.

Pretpostavimo da na glavnom dijalogu *Form1* postoji gumb čijim se klikom poziva novi dijalog *Form2* (Slika 2.6.1). U dijalogu *Form2* od korisnika se traži unos njegovog imena, a nakon zatvaranja tog dijaloga klikom na gumb "OK" (*modalResult = mrOk*) ime korisnika treba se ispisati u glavnom dijalogu *Form1* pomoću funkcije *ShowMessage*.



Slika 2.6.1. Unos podatka u drugom dijalogu

Sukladno prethodno opisanim koracima datoteka *Unit1.cpp* (implementacijska datoteka dijaloga *Form1*) treba sadržavati sljedeći programski odsječak (Primjer 2.6.1):

**Primjer 2.6.1. Sadržaj datoteke Unit1.cpp**

---

```
#include "Unit2.h" // zaglavlje dijaloga Form2
//...
void __fastcall TForm1::Button1Click(TObject *Sender){
    if (Form2->ShowModal() == mrOk) // prikaži dijalog i čekaj unos korisnika
        ShowMessage("Korisnik je unio ime " + Form2->EditIme->Text);
}
```

Prilikom klika na gumb izvršit će se metoda *Button1Click*. U njoj se pomoću metode *ShowModal* poziva (prikazuje) dijalog *Form2* na kojemu se nalazi polje za unos znakova *EditIme*. Tek nakon zatvaranja dijaloga *Form2* klikom na gumb "OK" na glavnom dijalogu ispisat će se tekst koji je unesen u komponenti *EditIme*. Da je pozvana metoda *Show* glavni dijalog ne bi čekao da korisnik unese svoje ime pa zatvori dijalog, već bi odmah nakon prikaza dijaloga *Form2* ispisao trenutni tekst koji se nalazi u *EditIme* komponenti.

U datoteci zaglavlja dijaloga *Form2* (*Unit2.h*) nalazi se i deklaracija

```
extern PACKAGE TForm2 *Form2;
```

kojom se pokazivač *Form2* proglašava globalnim. Upravo zbog ove deklaracije drugi dijalozi mogu koristiti dijalog *Form2* referencirajući se na njega spomenutim pokazivačem. Deklaracija pokazivača *Form2* nalazi se u *Unit2.cpp* datoteci, a u funkciji *main* (*Project1.cpp*) aplikacija je za njega automatski alocirala memoriju naredbom

```
Application->CreateForm(__classid(TForm2), &Form2);
```

Umjesto korištenja prethodno deklariranog i alociranog pokazivača *Form2* programer može i samostalno dinamički alocirati dijaloge (Primjer 2.6.2).

**Primjer 2.6.2. Dinamička alokacija dijaloga**

---

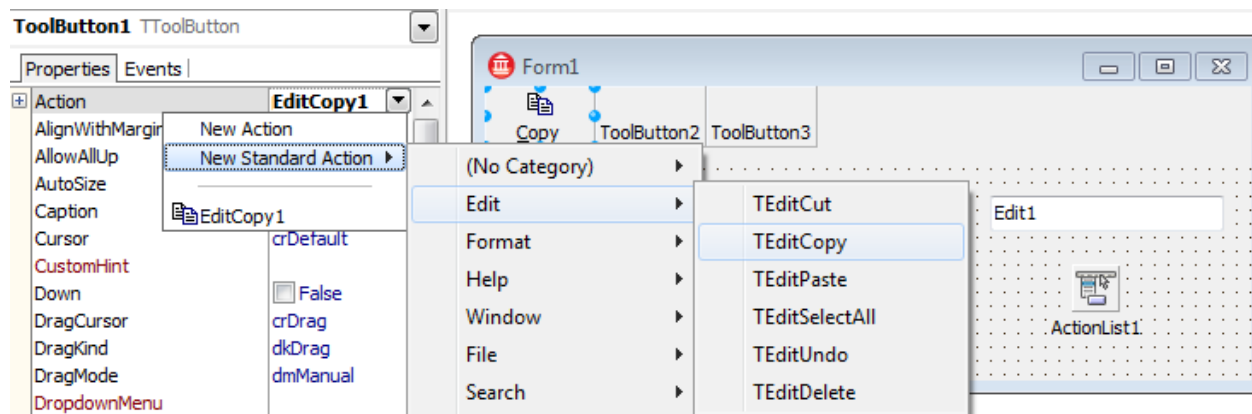
```
void __fastcall TForm1::Button1Click(TObject *Sender){
    TForm2* noviDijalog = new TForm2(this);
    noviDijalog->ShowModal();
    ShowMessage("Korisnik je unio ime: " + noviDijalog->EditIme->Text);
    delete noviDijalog;
}
```

Dijalog se alocira operatorom *new*, a zatim se prikazuje metodom *ShowModal*. Tek nakon toga dohvaćaju se podaci i dijalog se dealocira operatorom *delete*. Ovaj pristup najčešće se primjenjuje kada programer u jedno vrijeme želi koristiti više instanci istog dijaloga, pa mu tada samo jedan pokazivač nije dovoljan.

## 2.7. Akcijske liste

Akcijska lista predstavlja skupinu predefiniраниh akcija (komandi) koje se u svrhu automatizacije radnji povezuju s vizualnim komponentama. Akcije su prvenstveno predviđene za povezivanje sa stavkama izbornika i gumbima na alatnoj traci (eng. *toolbar*), no moguće ih je povezati i s drugim komponentama komandnog tipa (gumbi, radio gumbi, potvrdni okviri itd.).

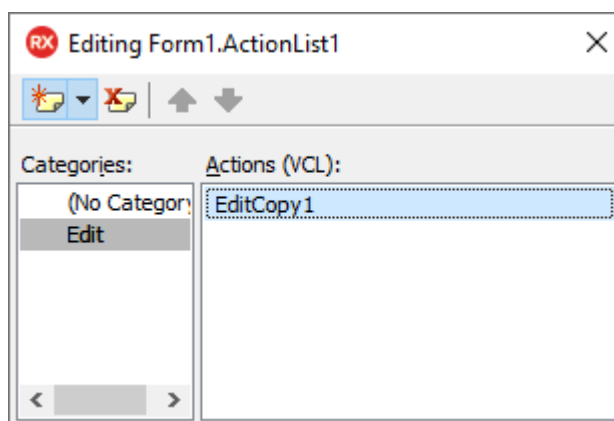
Primjerice, za komponente za obradu teksta (*TEdit*, *TMemo*, *TRichEdit* itd.) postoje predefiniрани tipovi akcija poput *TEditSelectAll*, *TEditCopy*, *TEditCut*, *TEditPaste* itd., koje će povezane komponente (primjerice, stavke izbornika) omogućiti (*Enable = true*) ili onemogućiti ovisno o tome da li je neki tekst označen ili nije.



Slika 2.7.1. Nova standardna akcija

Akcija se povezuje s komponentom korištenjem svojstva *Action*, prilikom čega je moguće odabrati jednu od već unaprijed kreiranih akcija (*EditCopy1*) ili kreirati novu akciju (Slika 2.7.1). Sve akcije pohranjuju su u komponenti *TActionList* (Slika 2.7.2), a ona može biti povezana i s komponentom *TImageList* ukoliko za pojedine akcije na povezanim komponentama želimo prikazati i ikone tih akcija.





Slika 2.7.2. Akcijska lista

Nakon dodavanja akcije *EditCopy1* gumbu na alatnoj traci (Slika 2.7.1) korisnik sada ima mogućnost kopiranja teksta pomoću tog gumba svaki puta kada je u nekom polju za unos znakova označen tekst. Kada tekst nije označen gumb *Copy* automatski je onemogućen.

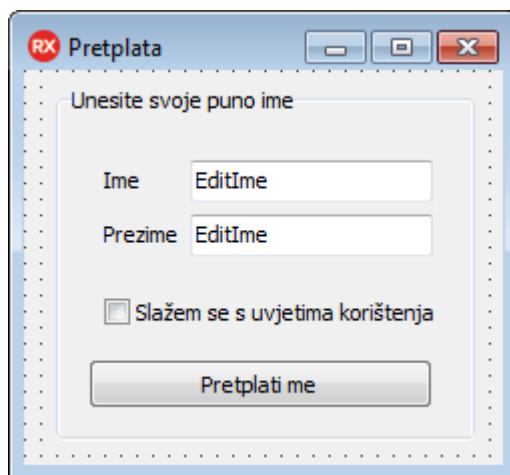
Sve akcije automatski su omogućene ili onemogućene ovisno o tome jesu li zaista izvedive nad komponentom koja je u fokusu. Tako akcije poput *TRichEditBold* i *TRichEditItalic* nije moguće koristiti prilikom rada s komponentom *TEdit*, ali ih je moguće koristiti s komponentom *TRichEdit*. Ipak, akcije poput *TEditCopy* i *TEditPaste* moguće je koristiti u obje komponente jer obje podržavaju ove funkcionalnosti.

Postoji i mnoštvo drugih tipova akcija poput akcija vezanih za rad s prozorima, standardnim dijalogima, bazama podataka itd. Tako je npr. moguće koristiti standardnu *TFileOpen* akciju te ju povezati s komponentom *TButton*. Klikom na gumb pojavit će se dijalog za odabir datoteke čije je postavke poput podrazumijevane ekstenzije, filtera i sl. moguće odrediti u svojstvu *Dialog* te akcije (*FileOpen1->Dialog*). Također, moguće je kreirati i vlastite akcije, a da bismo odredili njihovu funkcionalnost koristi se događaj *OnExecute*.

## 2.8. Lokalizacija korisničkog sučelja

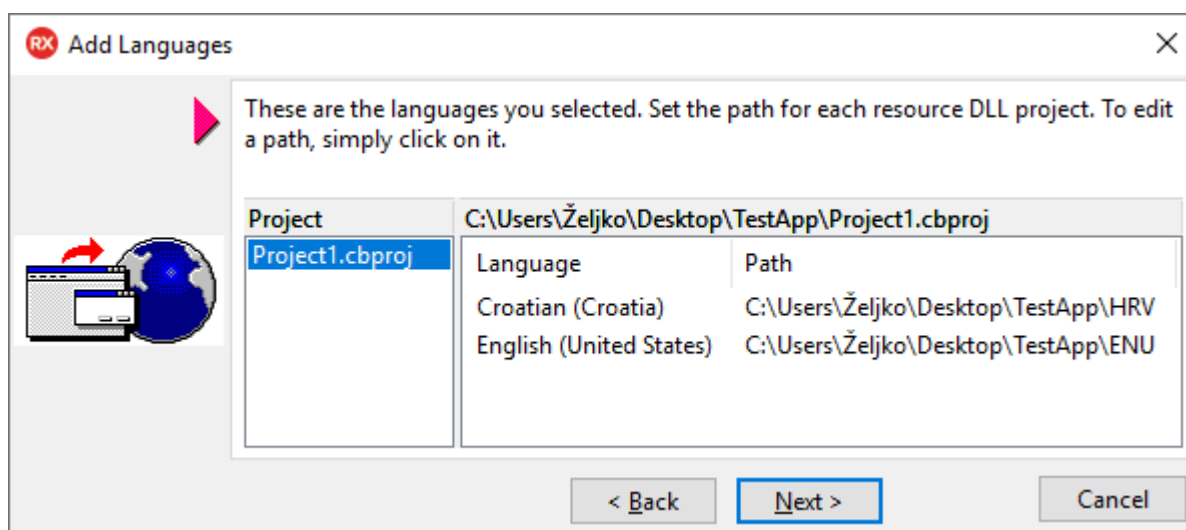
Lokalizacija je postupak prevođenja aplikacije na jedan ili više stranih jezika. Najčešće se pristupa lokalizaciji sučelja, no lokalizirati se može i funkcionalnost pojedinog dijela aplikacije (primjerice, izračun poreza koji ovisi o poreznoj stopi u pojedinoj zemlji).

C++ Builder omogućuje lokalizaciju sučelja aplikacije korištenjem alata *Translation Manager – Resource DLL Wizard*, *Translation Editor* i *Translation Repository*.



Slika 2.8.1. Dijalog za novog pretplatnika

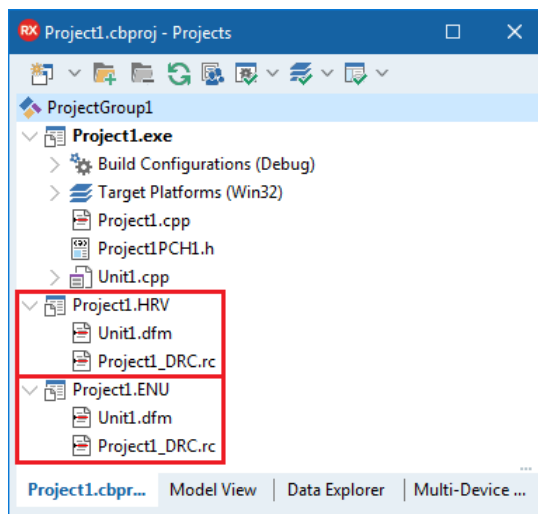
Pretpostavimo da želimo prevesti sučelje aplikacije na slici (Slika 2.8.1) tako da, osim hrvatskog, podržava i engleski jezik. Prvo je potrebno sve datoteke projekta pohraniti na disk, a zatim odabrati stavku *Project / Languages / Add...* Tada se pojavljuje dijalog alata *Resource DLL Wizard* koji nudi mogućnost odabira jednog ili više stranih jezika (Slika 2.8.2).



Slika 2.8.2. Alat Resource DLL Wizard

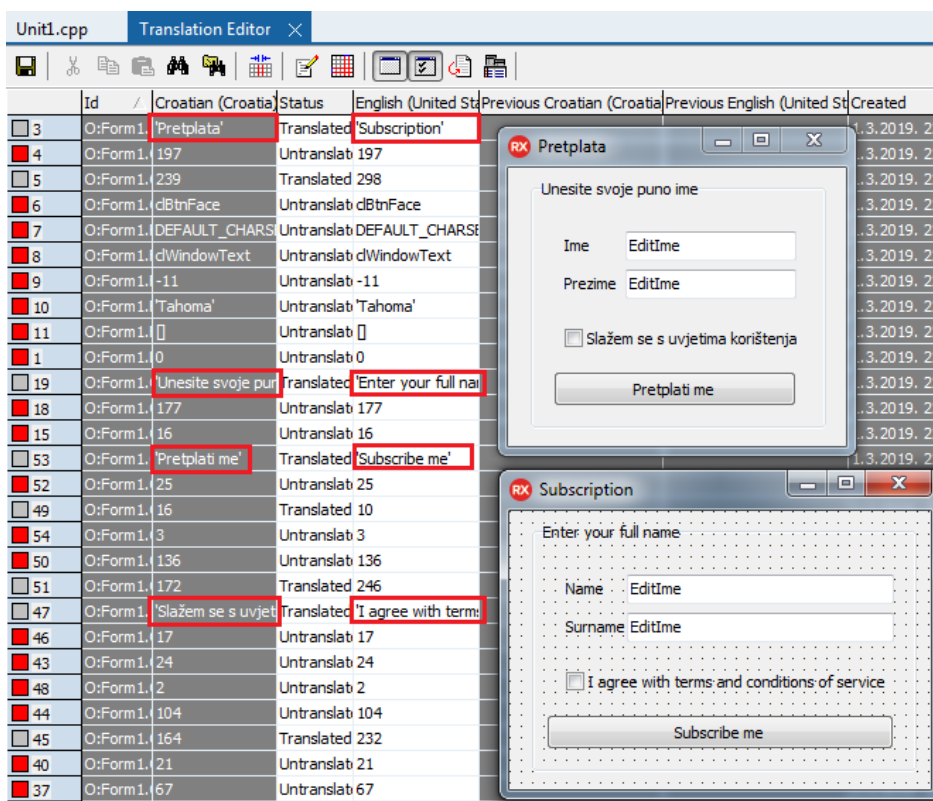
Za odabrane jezike na slici, alat *Resource DLL Wizard* generirat će zasebne resursne DLL projekte *Project1.HRV* i *Project1.ENU* (Slika 2.8.3) koji će biti pohranjeni u mapama *HRV*

i *ENU*. U svakom od tih projekata nalazit će se kopije svih dijaloga aplikacije (.dfm datoteke) te resursna (.rc) datoteka. Ona u obliku resursa sadrži sav ostali tekst korišten od strane aplikacije poput poruka o greškama, naslova standardnih gumbi itd.



Slika 2.8.3. Resursni DLL projekti

Dvostrukim klikom na .dfm i .rc datoteke (Slika 2.8.3) otvara se alat *Translation Editor* koji omogućuje prijevod sučelja i sistemskih poruka aplikacije (Slika 2.8.4).



Slika 2.8.4. Alat Translation Editor

Kao što prikazuje Slika 2.8.4, alat *Translation Editor* za svaku komponentu na dijalogu prikazao je tekst na hrvatskom i engleskom jeziku (prijevod moramo napisati sami). Štoviše, može se primijetiti da je engleska inačica dijaloga šira nego hrvatska inačica jer je engleski prijevod teksta duži na jednoj od komponenti (potvrđni okvir). Naime, za svaki od stranih jezika komponentama na dijalogu može se prevesti tekst, ali i specificirati drugačije početne koordinate, širina i visina.

Nakon što su napravljeni svi potrebni prijevodi, potrebno je zatvoriti alat *Translation Editor*, pohraniti sve datoteke te korištenjem stavke *build* generirati resursne DLL biblioteke *Project1.HRV* i *Project1.ENU*. Potom je potrebno u izborniku *Project* odabrati stavku *Languages / Set Active...*, te u prikazanom dijalogu odabrati jedan od jezika (hrvatski ili engleski) kako bi pri sljedećem pokretanju aplikacije vidjeli njeno sučelje na odabranom jeziku.

Ukoliko je došlo do naknadnih promjena na dijalogu (dodane su nove komponente ili sl.) bit će potrebno ažurirati lokalizirane inačice resursnih DLL biblioteka korištenjem stavke *Project / Languages / Update Localized Projects*. Tek nakon toga za svaki od jezika ponovno je potrebno pokrenuti alat *Translation Editor* te prevesti tekst na novim komponentama.

Pri svakom pokretanju aplikacije, ona će automatski provjeriti lokalne postavke računala u potrazi za preferiranim jezicima na računalu. Nakon toga, pokušat će učitati odgovarajuću inačicu resursne DLL biblioteke kako bi odmah prikazala sučelje na odgovarajućem jeziku. Zbog toga je resursne DLL biblioteke uvijek potrebno distribuirati uz izvršnu (.exe) datoteku. [8]

Osim automatizmom pri pokretanju, jezik je moguće odabrati i tijekom rada aplikacije (primjerice, klikom na gumb ili odabirom stavke u izborniku). Da bismo to realizirali potrebno je u projekt uključiti datoteku *reinit.pas* (*Project / Add to Project...*). Ova datoteka dolazi s instalacijom C++ Buildera (... \ *Samples\ Object Pascal\ VCL\ RichEdit*) te ju je prije dodavanja u projekt poželjno kopirati u radnu mapu tog projekta.

**Primjer 2.8.1. Promjena jezika sučelja klikom na gumb**

```
#include "reinit.hpp"
// ...
void __fastcall TForm1::Button1Click(TObject *Sender){
    const int ENGLISH = LANG_ENGLISH | (SUBLANG_ENGLISH_US << 10);
    if(LoadNewResourceModule(ENGLISH))
        ReinitializeForms();
}
```

Nakon dodavanja datoteke *reinit.pas* u projekt u C++ Builderu, potrebno je pretprocesorskom naredbom *#include* uključiti njenu datoteku zaglavlja *reinit.hpp* (Primjer 2.8.1). Naime, kada se Delphi programski kôd (.pas datoteka) prevodi u C++ Builderu za tu se datoteku automatski generira njena datoteka zaglavlja s .hpp ekstenzijom. Takva datoteka predstavlja omotač oko Delphi programskog kôda, te omogućava njegovo korištenje u C++ Builderu.

Dalje se klikom na gumb poziva funkcija *LoadNewResourceModule* koja će na osnovu predanog identifikatora za primarni jezik (LANG\_ENGLISH) i identifikatora za podjezik (SUBLANG\_ENGLISH\_US) pokušati učitati odgovarajuću inačicu resursne DLL biblioteke. Ukoliko se resursna DLL biblioteka uspješno učitala svi dijalozi u aplikaciji će se reinicijalizirati, odnosno aplikacija će početi koristiti njihove lokalizirane inačice.

Popis identifikatora za jezike i podjezike dostupan na Microsoftovim web stranicama:

<https://docs.microsoft.com/en-us/windows/desktop/intl/language-identifier-constants-and-strings>



## 3. Formati za pohranu i razmjenu podataka

### 3.1. INI

Inicijalizacijske (INI) datoteke prvobitno su bile korištene za pohranu postavki operacijskog sustava Windows. Ipak, zbog jednostavne strukture i čitljivosti (tekstualni format) njihova namjena se proširila i na pohranu postavki aplikacija. Inicijalizacijske datoteke se i danas koriste, ali su kod složenijih (višekorisničkih) aplikacija najčešće zamijenjene bazom podataka Windows Registry.

#### *Primjer 3.1.1. Sadržaj datoteke POSTAVKE.INI*

```
[GLAVNI DIJALOG]
Naslov=Glavni dijalog aplikacije
X1=100
Y1=100
X2=300
Y2=200
```

Inicijalizacijske datoteke strukturirane su na način da sadrže sekcije, svojstava i vrijednosti. Sekcije se koriste za grupiranje svojstava, a svako od svojstava sadrži svoju vrijednost. Tako Primjer 3.1.1 sadrži sekciju *GLAVNI DIJALOG*, a unutar nje prisutna su svojstva *Naslov*, *X1*, *Y1*, *X2* i *Y2* sa svojim vrijednostima.

#### *Primjer 3.1.2. Čitanje podataka iz inicijalizacijske datoteke*

```
#include <registry.hpp>
// ...

__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner){
    TIniFile *ini;
    ini = new TIniFile("postavke.ini");
    Caption = ini->ReadString("GLAVNI DIJALOG", "Naslov", "");
    Left = ini->ReadInteger("GLAVNI DIJALOG", "X1", 0);
    Top = ini->ReadInteger("GLAVNI DIJALOG", "Y1", 0);
    Width = ini->ReadInteger("GLAVNI DIJALOG", "X2", 0);
    Height = ini->ReadInteger("GLAVNI DIJALOG", "Y2", 0);
    delete ini;
}
```

Pretpostavimo da se primjer inicijalizacijske datoteke "postavke.ini" (Primjer 3.1.1) koristi prilikom pokretanja naše aplikacije kako bismo postavili naslov glavnog dijaloga i njegove koordinate ( $X1$ ,  $Y1$ ,  $X2$  i  $Y2$ ). Tada je za učitavanje njenog sadržaja moguće koristiti programski kôd koji prikazuje Primjer 3.1.2.

Biblioteka VCL sadrži klasu *TIniFile* (zaglavlje *registry.hpp*), a njenim metodama vrlo je jednostavno čitati i obrađivati sadržaj inicijalizacijskih datoteka. Metode sa prefiksom *Read* imaju tri parametra; naziv sekcije, naziv svojstva i podrazumijevana (eng. *default*) vrijednost svojstva. Naime, ukoliko iz nekog razloga nije moguće pročitati vrijednost nekog svojstva sekcije (primjerice, svojstvo ne postoji) onda će metode sa prefiksom *Read* za takvo svojstvo vratiti njegovu podrazumijevanu vrijednost.

### Primjer 3.1.3. Pohrana podataka u inicijalizacijsku datoteku

```
#include <registry.hpp>
// ...

void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action){
    TIniFile *ini;
    ini = new TIniFile("postavke.ini");
    ini->WriteString("GLAVNI DIJALOG", "Naslov", Caption);
    ini->WriteInteger("GLAVNI DIJALOG", "X1", Left);
    ini->WriteInteger("GLAVNI DIJALOG", "Y1", Top);
    ini->WriteInteger("GLAVNI DIJALOG", "X2", Width);
    ini->WriteInteger("GLAVNI DIJALOG", "Y2", Height);
    delete ini;
}
```

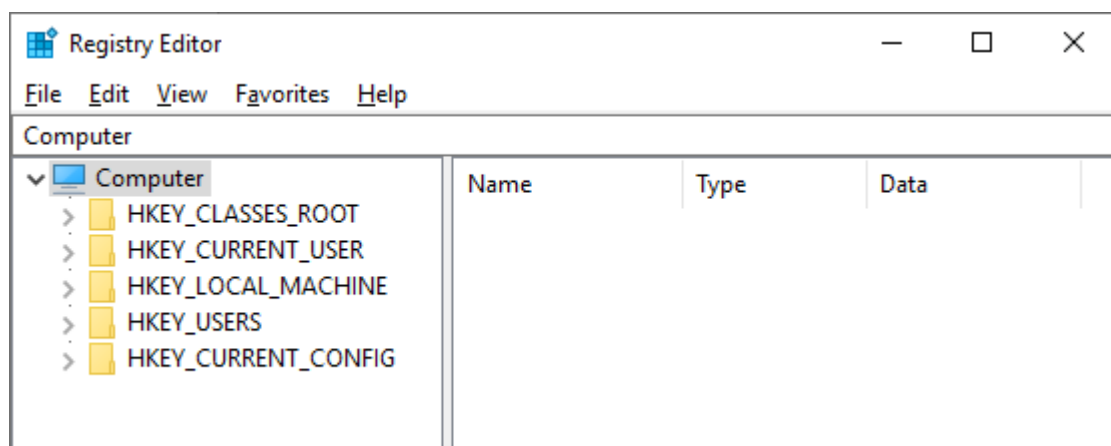
Za pohranu sadržaja inicijalizacijske datoteke koriste se metode sa prefiksom *Write*. I one također imaju tri parametra – naziv sekcije, naziv svojstva i vrijednost svojstva. Tako se prilikom zatvaranja dijaloga izvršava prikazani programski kôd (Primjer 3.1.3), odnosno u inicijalizacijsku datoteku pohranjuju se trenutne koordinate dijaloga. Štoviše, kombinirajući prethodna dva primjera naša aplikacija će svaki puta prilikom pokretanja prikazati dijalog na zadnje postavljene koordinatama.

Klasa *TIniFile* sadrži metode za čitanje i pohranu svojstava cijelih i realnih brojeva, datuma, vremena, znakovnih nizova i binarnog sadržaja. Također sadrži i metode za rad sa sekcijama (*ReadSection*, *EraseSection*, *SectionExists* itd.).



## 3.2. Windows registar

Za pohranu i čitanje postavki aplikacija danas se najčešće koristi Windows registar. To je baza podataka organizirana u obliku stabla koja sadrži informacije o računalu i njegovu radu, poput informacija o pogonskim uređajima i njihovim postavkama, postavkama operacijskog sustava, instaliranim aplikacijama itd. [9] Jedna od glavnih prednosti Windows registra u odnosu na inicijalizacijske datoteke jest mogućnost zaštite pristupa podacima te mogućnost pohrane postavki aplikacija za svaki od Windows korisničkih računa zasebno.



Slika 3.2.1. Registry Editor

Umjesto sekcija i svojstava, Windows registar sadrži ključeve i vrijednosti. Ključevi prvenstveno služe kao kontejneri za vrijednosti, ali u sebi mogu sadržavati i druge (pod)ključeve. Windows registar sastoji se od 5 korijenskih ključeva (Slika 3.2.1). Za pohranu postavki aplikacija najčešće koriste ključevi *HKEY\_CURRENT\_USER* (postavke koje vrijede samo za aktivnog korisnika Windowsa) i *HKEY\_LOCAL\_MACHINE* (postavke koje vrijede za sve Windows korisnike tog računala). Pohrana postavki za sve korisnike Windowsa također prethodno zahtjeva odgovarajuća prava i dozvole.

Kao i u primjerima s inicijalizacijskom datotekom, pretpostavimo da u Windows registar želimo pohranjivati i iz njega učitavati zadnje postavljene koordinate glavnog dijaloga aplikacije. U tu svrhu prethodno je potrebno odabrati lokaciju za pohranu postavki naše

aplikacije, odnosno odgovarajući korijenski ključ i podključeve, a zatim u ključ aplikacije pohraniti tražene vrijednosti  $X1$ ,  $Y1$ ,  $X2$  i  $Y2$ .

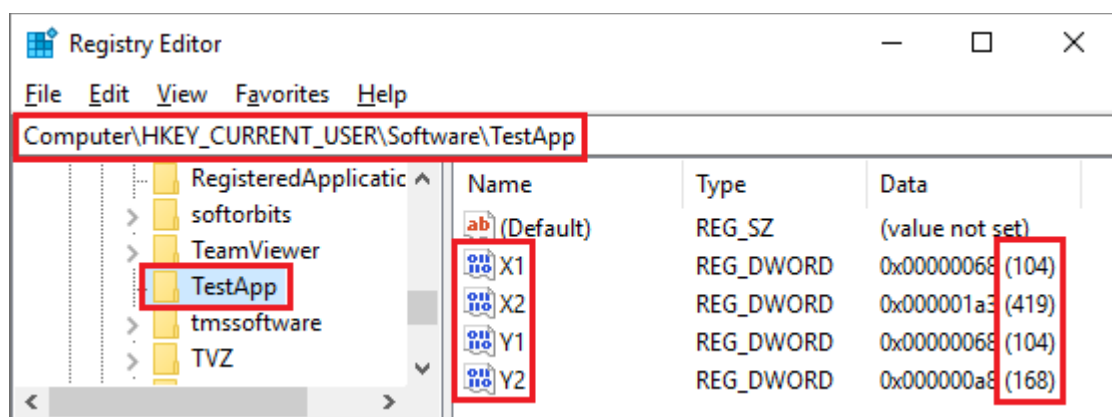
### Primjer 3.2.1. Pohrana postavki aplikacije u Windows registar

```
#include <registry.hpp>
// ...

void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action){
    TRegistry *Registry = new TRegistry;
    Registry->RootKey = HKEY_CURRENT_USER;
    UnicodeString KLJUC = "Software\\TestApp";

    if(Registry->OpenKey(KLJUC, true)){ // izradi ključ (ukoliko ne postoji)
        Registry->WriteInteger("X1", Left);
        Registry->WriteInteger("Y1", Top);
        Registry->WriteInteger("X2", Width);
        Registry->WriteInteger("Y2", Height);
        Registry->CloseKey(); // zatvori ključ
    }
    delete Registry;
}
```

Prilikom zatvaranja dijaloga izvršava se prikazani programski kôd (Primjer 3.2.1). U njemu se za pohranu postavki aplikacije koristi korijenski ključ `HKEY_CURRENT_USER`, odnosno pohranjene postavke vrijedit će samo za trenutnog Windows korisnika. Unutar tog korijenskog ključa nalazi se ključ `Software` koji je namijenjen upravo za pohranu postavki Windows aplikacija, a unutar njega kreiran je ključ `TestApp` za pohranu postavki naše aplikacije (Slika 3.2.2).



Slika 3.2.2. Postavke testne aplikacije pohranjene u Windows registru

Da bi se vrijednosti unutar nekog ključa mogli čitati ili pohranjivati, taj ključ prethodno mora biti otvoren metodom *OpenKey* klase *TRegistry*. Ova metoda pokušava otvoriti traženi ključ, a ukoliko on ne postoji može ga automatski kreirati ukoliko drugi parametar (*CanCreate*) ima vrijednost *true*. Istovremeno samo jedan ključ može biti otvoren jer se operacije čitanja i pohrane odnose samo na vrijednosti u otvorenom ključu. Sukladno tome, nakon završetka tih operacija ključ treba zatvoriti metodom *CloseKey*.

Kao i u slučaju inicijalizacijskih datoteka, metode za čitanje i pohranu vrijednosti imaju prefikse *Read* i *Write*. Primjerice,

<code>WriteBinaryData</code>	<code>WriteBool</code>	<code>WriteCurrency</code>	<code>WriteDate</code>
<code>WriteDateTime</code>	<code>WriteExpandString</code>	<code>WriteFloat</code>	<code>WriteInteger</code>
<code>WriteString</code>	<code>WriteTime</code>		

Metode za pohranu (prefiks *Write*) imaju dva parametra – simbolički naziv vrijednosti i samu vrijednost, dok metode za čitanje (prefiks *Read*) imaju samo jedan parametar, a to je naziv vrijednosti koju se želi pročitati. Metode za čitanje vraćaju pročitanu vrijednost preko svoje povratne vrijednosti (Primjer 3.2.2).

#### *Primjer 3.2.2. Čitanje postavki aplikacije iz Windows registra*

```
#include <registry.hpp>
// ...

__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner) {
    TRegistry *Registry = new TRegistry;
    Registry->RootKey = HKEY_CURRENT_USER;
    UnicodeString KLJUC = "Software\\TestApp";

    // ako ključ postoji i ako ga se može otvorit...
    if(Registry->OpenKey(KLJUC, false)) {
        Left = Registry->ReadInteger("X1");
        Top = Registry->ReadInteger("Y1");
        Width = Registry->ReadInteger("X2");
        Height = Registry->ReadInteger("Y2");
        Registry->CloseKey();
    }
    delete Registry;
}
```

I prilikom čitanja vrijednosti ključ se otvara metodom *OpenKey*. Ipak, ovaj puta njen drugi parametar (*CanCreate*) ima vrijednost *false* kako bismo spriječili stvaranje tog ključa u slučaju da on prethodno ne postoji. Naime, ukoliko bismo sada stvorili novi ključ aplikacija bi generirala greške pri pokušaju čitanja vrijednosti *X1*, *X2*, *Y1* i *Y2* jer te vrijednosti inicijalno ne bi postojale unutar novokreiranog ključa.

#### *Primjer 3.2.3. Brisanje ključa iz Windows registra*

```
#include <registry.hpp>
// ...
void __fastcall TForm1::BrisiKljucClick(TObject *Sender) {
    TRegistry *Registry = new TRegistry;
    Registry->RootKey = HKEY_CURRENT_USER;
    UnicodeString KLJUC = "Software\\TestApp";

    if(Registry->KeyExists(KLJUC)) // ako ključ postoji
        Registry->DeleteKey(KLJUC); // izbriši ga
    delete Registry;
}
```

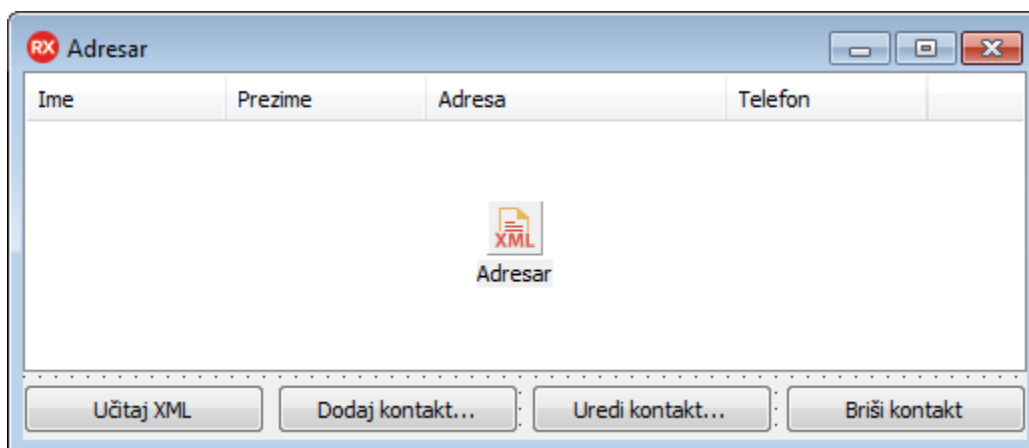
Ukoliko je potrebno izbrisati ključ iz Windows registra koristi se metoda *DeleteKey*, a tom prilikom iz njega se brišu sve njegove vrijednosti i podključevi. Prvo je poželjno provjeriti postoji li ključ (metoda *KeyExists*), pa ga tek onda izbrisati metodom *DeleteKey* (Primjer 3.2.3). Ključ nije prethodno potrebno otvarati metodom *OpenKey*, kao što je to bio slučaj pri upisu i pohrani podataka.

### 3.3. XML

XML (*EXtensible Markup Language*) standardizirani je jezik za označavanje podataka. Izrađen je s namjerom da bude razumljiv i aplikacijama i ljudima te je neovisan o platformi na kojoj se koristi. XML sadrži polustrukturirani oblik podataka (postoje oznake ali ne i striktni podatkovni model), a najčešće se koristi u svrhu komunikacije među aplikacijama, prilikom opisivanja drugih struktura i dokumenata te općenito za pohranu podataka. Podaci se najvećim dijelom nalaze unutar XML elemenata i atributa, dok se umjesto atributa vrlo često se koriste i podelementi.

Biblioteka VCL podržava obradu sadržaja XML datoteka pomoću komponente *TXMLDocument* i *XML Data Binding Wizard* alata. Navedena komponenta predstavlja sadržaj XML datoteke i svojim metodama omogućava njegovo učitavanje i pohranu, dok se korištenjem *XML Data Binding Wizard* alata generira programski kôd za rad s pojedinim XML elementima i atributima.

Pretpostavimo da razvijamo aplikaciju za uređivanje adresara koji je pohranjen u XML datoteci (Slika 3.3.1). Na primjeru ove aplikacije možemo demonstrirati nekoliko operacija s XML datotekom poput učitavanja svih postojećih zapisa te dodavanja, uređivanja i brisanja pojedinih kontakata, odnosno elemenata unutar XML dokumenta.



Slika 3.3.1. Aplikacija za uređivanje adresara

Za početak, poželjno je izraditi uzorak XML datoteke koja sadrži primjere nekoliko već pohranjenih kontakata (Primjer 3.3.1). Umjesto uzorka takve XML datoteke moguće je koristiti i XML shemu (.xsd), datoteku *Document Type Definition* (.dtd) ili *Reduced XML Data* (.xdr). [10]

#### Primjer 3.3.1. Inicijalni sadržaj datoteke adresar.xml

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<adresar>
  <kontakt>
    <ime>Ante</ime>
    <prezime>Antić</prezime>
    <adresa>Prva ulica 1, 10000 Zagreb</adresa>
    <telefon>+38511234567</telefon>
  </kontakt>
</adresar>
```

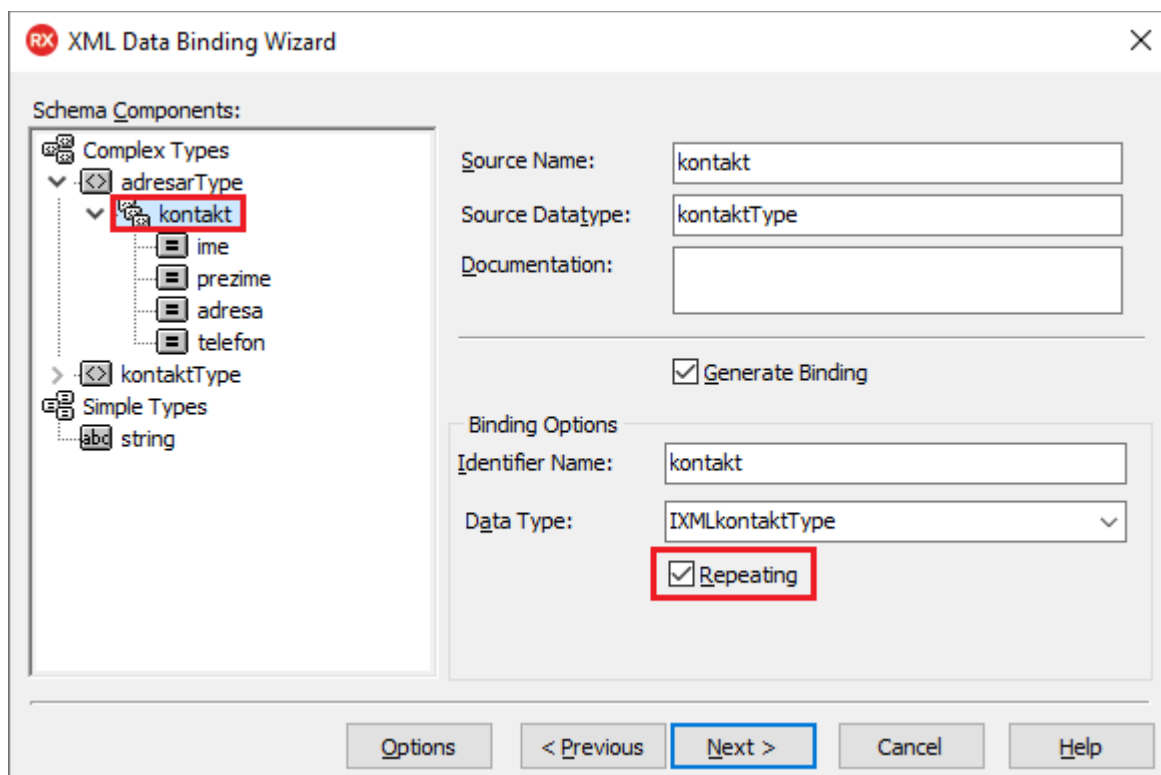
```

</kontakt>
<kontakt>
    <ime>Pero</ime>
    <prezime>Perić</prezime>
    <adresa>Prva ulica 2, 10000 Zagreb</adresa>
    <telefon>+38512345678</telefon>
</kontakt>
</adresar>

```

Atribut *encoding* koji se nalazi u zaglavlju XML dokumenta obično ima vrijednost UTF-8, dok se za podršku HR znakova koristi shema ISO-8859-2.

Komponenta *TXMLDocument* sadrži svojstvo *FileName* čija vrijednost treba biti naziv jedne od navedenih tipova datoteka (npr. *adresar.xml*). Tek nakon toga se dvostrukim klikom na komponentu *TXMLDocument* može pokrenuti *XML Data Binding Wizard* alat.



Slika 3.3.2. XML Data Binding Wizard alat - struktura XML datoteke

Ukoliko komponenta *TXMLDocument* koristi datoteku *adresar.xml*, alat *XML Data Binding Wizard* prepoznat će njenu strukturu i prikazati ju (Slika 3.3.2). Potrebno je primijetiti da je prepoznata struktura dokumenta *adresar.xml* točna, odnosno da se XML dokument sastoji

od dvaju tipova elemenata (*adresar* i *kontakt*), te da se element *kontakt* ponavlja unutar elementa *adresar* (adresar može imati više kontakata).

Unutar elementa *kontakt* nalaze se podelementi s podacima (*ime*, *prezime*, *adresa* i *telefon*) čije je tipove i prava pristupa na ovom dijalogu također moguće odrediti. Podrazumijevano će alat *XML Data Binding Wizard* automatski pokušati odrediti tipove za svaki od elemenata s podacima, ali rezultat tog procesa nije uvijek točan.

Klikom na gumb *Next* generira se sučelje (eng. *interface*) za rad s XML elementima *adresar* i *kontakt*.

### Primjer 3.3.2. Sučelje elementa "adresar"

```
// IXMLadresarType
__interface INTERFACE_UUID("{951DE878-A29F-46F5-8FAD-46C7835B0747}")
IXMLadresarType : public Xml::Xmlintf::IXMLNodeCollection{
public:
    // Property Accessors
    virtual __di_IXMLkontaktType __fastcall Get_kontakt(int Index) = 0;
    // Methods & Properties
    virtual __di_IXMLkontaktType __fastcall Add() = 0;
    virtual __di_IXMLkontaktType __fastcall Insert(const int Index) = 0;
    __property __di_IXMLkontaktType kontakt[int Index] = { read=Get_kontakt };
};
```

S obzirom da element *adresar* može sadržavati više elemenata *kontakt*, sučelje iz prethodnog primjera sadrži metode za dohvaćanje kontakta po indeksu te metode za dodavanje novog kontakta.

### Primjer 3.3.3. Sučelje elementa "kontakt"

```
// IXMLkontaktType
__interface INTERFACE_UUID("{EA71AD93-198C-4344-B5C6-0232DF6209BD}")
IXMLkontaktType : public Xml::Xmlintf::IXMLNode{
public:
    // Property Accessors
    virtual System::UnicodeString __fastcall Get_ime() = 0;
    virtual System::UnicodeString __fastcall Get_prezime() = 0;
    virtual System::UnicodeString __fastcall Get_adresa() = 0;
    virtual System::UnicodeString __fastcall Get_telefon() = 0;
```

```

virtual void __fastcall Set_ime(System::UnicodeString Value) = 0;
virtual void __fastcall Set_prezime(System::UnicodeString Value) = 0;
virtual void __fastcall Set_adresa(System::UnicodeString Value) = 0;
virtual void __fastcall Set_telefon(System::UnicodeString Value) = 0;
// Methods & Properties
__property System::UnicodeString ime = { read=Get_ime, write=Set_ime };
__property System::UnicodeString prezime = { read=Get_prezime,
write=Set_prezime };
__property System::UnicodeString adresa = { read=Get_adresa, write=Set_adresa };
__property System::UnicodeString telefon = { read=Get_telefon,
write=Set_telefon };
};

```

Za svaki od elemenata s podacima (*ime*, *prezime*, *adresa* i *telefon*) u sučelju *IXMLkontaktType* generirana su istoimena svojstva (eng. *properties*). Tako će se sadržaj tih elemenata unutar XML dokumenta moći izravno obrađivati preko imena njihovih svojstava ili korištenjem njihovih metoda *get* i *set* (Primjer 3.3.3).

Sva sučelja elemenata (Primjer 3.3.2 i Primjer 3.3.3) bit će pohranjena u datoteci zaglavlja *adresar.h*, dok će se implementacije metoda iz oba sučelja nalaziti u datoteci *adresar.cpp*. Dijalog u kojemu želimo obrađivati sadržaj datoteke *adresar.xml* mora uključiti datoteku zaglavlja *adresar.h* kako bi mogao koristiti programski kôd generiran od strane alata *XML Data Binding Wizard*.

#### Primjer 3.3.4. Čitanje svih kontakata iz XML dokumenta

```

#include "adresar.h"
// ...
void __fastcall TForm1::GUcitajXMLClick(TObject *Sender){
    // učitaj adresar korištenjem komponente TXMLDocument komponente "Adresar"
    _di_IXMLadresarType adresar = Getadresar(Adresar);

    ListView1->Items->Clear();
    for(int i = 0; i < adresar->Count; i++){
        ListView1->Items->Add();
        ListView1->Items->Item[i]->Caption = adresar->kontakt[i]->ime;
        ListView1->Items->Item[i]->SubItems->Add(adresar->kontakt[i]->prezime);
        ListView1->Items->Item[i]->SubItems->Add(adresar->kontakt[i]->adresa);
        ListView1->Items->Item[i]->SubItems->Add(adresar->kontakt[i]->telefon);
    }
}

```



```
}
```

Primjer 3.3.4 prikazuje kako učitati sve kontakte iz XML datoteke *adresar.xml*. Prvo je potrebno uključiti datoteku zaglavlja *adresar.h* u kojoj je potrebno primijetiti sljedeće deklaracije:

```
typedef System::DelphiInterface<IXMLadresarType> _di_IXMLadresarType;  
typedef System::DelphiInterface<IXMLkontaktType> _di_IXMLkontaktType;
```

Ovim deklaracijama Delphi sučelja *IXMLadresarType* i *IXMLkontaktType* moguće je koristiti kao da su C++ klase *\_di\_IXMLadresarType* i *\_di\_IXMLkontaktType*. Tako je u prethodnom primjeru kreiran objekt *adresar* tipa *\_di\_IXMLadresarType* koji je inicijaliziran globalnom funkcijom *Getadresar*. Prototip ove funkcije također se nalazi u datoteci zaglavlja *adresar.h*.

Da bismo pročitali sve kontakte u XML dokumentu prethodno je potrebno znati koliko ih zaista ima. Ta vrijednost zapisana je u svojstvu *Count* instance *adresar* (*adresar->Count*), dok se pojedini kontakt elementi i njihovi podelementi s podacima dohvaćaju preko svojstva *kontakt* (npr. *adresar->kontakt[i]->ime*).

Za prikaz kontakata korištena je komponenta *TListView* (Slika 3.3.1). Ona inicijalno ne prikazuje podatke u obliku izvještaja, pa je njenom svojstvu *ViewStyle* pomoću objektnog inspektora potrebno dodijeliti vrijednost *vsReport*. Tada je dvostrukim klikom na komponentu moguće dodavati i uređivati stupce za prikaz podataka.

#### **Primjer 3.3.5. Dodavanje novog kontakta u XML datoteku**

```
#include "adresar.h"  
// ...  
void __fastcall TForm1::GDodajKontaktClick(TObject *Sender) {  
    _di_IXMLadresarType adresar = Getadresar(Adresar);  
    _di_IXMLkontaktType kontakt = adresar->Add();  
  
    kontakt->ime = "Ime";  
    kontakt->prezime = "Prezime";  
    kontakt->adresa = "Adresa";  
    kontakt->telefon = "Telefon";  
}
```

```
Adresar->SaveToFile(Adresar->FileName);  
}
```

Za dodavanje novog kontakta u XML datoteku potrebno je pozvati metodu *Add* instance *adresar*. Ova metoda dodat će novu (praznu) instancu elementa *kontakt* u XML datoteku, a njeni podelementi s podacima inicijaliziraju se u nastavku programskog kôda, kao što prikazuje Primjer 3.3.5.

Zadnjom naredbom, odnosno pozivom metode *SaveToFile*, novi kontakt pohranjuje se u XML datoteku. To je moguće i automatizirati dodavanjem vrijednosti *true* svojstvu *Options / doAutoSave* u postavkama komponente *TXMLDocument*. Također, prilikom pohranjivanja novog kontakta svi njegovi podelementi bit će podrazumijevano pohranjeni u istom retku. Da bismo svaki element pohranili u njegovom retku, te pri tome za određen broj razmaka uvukli njegove podelemente, potrebno je svojstvu *Options / doNodeAutoIndent* dodijeliti vrijednost *true*. Broj razmaka određen je vrijednošću svojstva *NodeIndentStr*.

#### **Primjer 3.3.6. Uređivanje postojećeg kontakta u XML datoteci**

```
#include "adresar.h"  
// ...  
void __fastcall TForm1::GUrediKontaktClick(TObject *Sender){  
    // ukoliko niti jedan kontakt u komponenti ListView1 nije označen...  
    if(ListView1->ItemIndex == -1)  
        return; // kraj  
    _di_IXMLadresarType adresar = Getadresar(Adresar);  
    // dohvati označeni kontakt  
    _di_IXMLkontaktType kontakt = adresar->kontakt[ListView1->ItemIndex];  
  
    kontakt->ime = "Novo ime";  
    kontakt->prezime = "Novo prezime";  
    kontakt->adresa = "Nova adresa";  
    kontakt->telefon = "Novi telefon";  
    Adresar->SaveToFile(Adresar->FileName);  
}
```

Da bismo uredili postojeći kontakt u XML dokumentu potrebno je dohvatiti kontakt koji se želi urediti te zatim izmijeniti i pohraniti nove vrijednosti njegovih podelemenata (Primjer

3.3.6). U prikazanom primjeru uredit će se označeni kontakt u komponentu *ListView1*, a dohvaća ga se pomoću indeksa (`ListView1->ItemIndex`).

#### **Primjer 3.3.7. Brisanje kontakta iz XML datoteke**

---

```
#include "adresar.h"
// ...
void __fastcall TForm1::GBrisiKontaktClick(TObject *Sender){
    // ukoliko niti jedan kontakt u komponenti ListView1 nije označen...
    if(ListView1->ItemIndex == -1)
        return; // kraj
    _di_IXMLadresarType adresar = Getadresar(Adresar);
    // izbriši označeni kontakt
    adresar->Delete(ListView1->ItemIndex);
    Adresar->SaveToFile(Adresar->FileName);
}
```

Za brisanje postojećeg kontakta iz XML datoteke potrebno je tek pozvati metodu *Delete* instance *adresar* i nakon toga pohraniti izmjene (Primjer 3.3.7). Metoda *Delete* prima indeks elementa kojeg treba izbrisati, što je u ovom slučaju indeks označenog kontakta u komponenti *ListView1*.

## **3.4. JSON**

JSON (*JavaScript Object Notation*) predstavlja tekstualni format polustrukturiranog sadržaja koji se koristi za razmjenu podataka. Mnogi ga danas koriste kao alternativu XML-u, a svoju popularnost stekao je kroz popularizirani razvoj REST web servisa. JSON je izvorno trebao biti samo podskup skriptnog jezika JavaScript, ali danas je potpuno neovisan o jeziku. Štoviše, podrška za čitanje i pohranu podataka u JSON formatu sada je dostupna u mnogim programskim jezicima, pa tako i u C++u.

JSON dokument sačinjen je od dvije vrste struktura:

- ključ / vrijednost
- niz vrijednosti

JSON vrijednost može biti broj, niz znakova (*string*) ili niz nekog drugog tipa, logička vrijednost *true* ili *false*, null ili objekt. Objekt u JSON dokumentu opisuje se kao neuređeni niz ključeva i vrijednosti odvojenih zarezom, dok nizovi predstavljaju uređene kolekcije vrijednosti odvojene zarezom. [11]

#### *Primjer 3.4.1. Adresar u JSON formatu*

---

```
{
  "adresar":
  [
    {
      "ime" : "Ante",
      "prezime" : "Antić",
      "ulica" : "Prva ulica",
      "broj" : 42,
      "mjesto" : "Zagreb",
      "telefon" : "+38512345678"
    },
    {
      "ime" : "Pero",
      "prezime" : "Perić",
      "ulica" : "Prva ulica",
      "broj" : 43,
      "mjesto" : "Zagreb",
      "telefon" : "+38522345678"
    }
  ]
}
```

Adresar (popis kontakata) možemo prikazati i u JSON formatu (Primjer 3.4.1). Adresar je opisan kao niz objekata (kontakata) unutar uglatih zagrada "[" i "]" koji su međusobno omeđeni vitičastim zagradaama "{" i "}" i odvojeni zarezom. Unutar svakog od tih objekata nalazi se komplet ključeva i vrijednosti kojima se opisuje kontakt, odnosno podaci koje želimo moći koristiti i unutar naše aplikacije.

#### *Primjer 3.4.2. Čitanje sadržaja JSON datoteke (pristup DOM)*

---

```
#include <System.JSON.hpp>
#include <memory>
// ...
```

```
void __fastcall TForm1::GUcitajJSONClick(TObject *Sender){
    // učitaj sadržaj datoteke adresar.json u memoriju
    std::unique_ptr<TStringStream> jsonStream(new TStringStream);
    jsonStream->LoadFromFile("adresar.json");
    UnicodeString jsonDoc = jsonStream->DataString;
    // dohvati cijeli JSON
    TJSONObject* Adresar = (TJSONObject*)TJSONObject::ParseJSONValue(jsonDoc);
    // dohvati niz "adresar" unutar JSON dokumenta
    TJSONArray* Kontakti = (TJSONArray*)TJSONObject::ParseJSONValue(
        Adresar->GetValue("adresar")->ToString());
    // pročitaj i ispiši pojedine kontakte
    UnicodeString Kontakt;
    for(int i = 0; i < Kontakti->Count; i++){
        Kontakt = Kontakti->Items[i]->GetValue<UnicodeString>("ime") + " "
            + Kontakti->Items[i]->GetValue<UnicodeString>("prezime")+"\n"
            + Kontakti->Items[i]->GetValue<UnicodeString>("ulica") + ", "
            + Kontakti->Items[i]->GetValue<int>("broj") + ", "
            + Kontakti->Items[i]->GetValue<UnicodeString>("mjesto")+"\n"
            + Kontakti->Items[i]->GetValue<UnicodeString>("telefon");
        ShowMessage(Kontakt);
    }
}
```

Sadržaj adresara pohranjenog u JSON formatu (Primjer 3.4.1) pročitao je korištenjem pristupa DOM (*Document Object Model*) gdje se pojedini čvorovi dokumenta predstavljaju zasebnim objektima u memoriji (Primjer 3.4.2). Prvo je kompletan JSON dokument učitao iz datoteke i pohranjen u objekt *Adresar*, nakon čega se iz objekta *Adresar* učitao niz kontakata u objekt *Kontakti*. Pojedini čvorovi dohvaćaju se korištenjem metode *ParseJSONValue* klase *TJSONObject*, dok se vrijednosti unutar niza dohvaćaju predloškom metode *GetValue*.

Ovisno o veličini i kompleksnosti JSON dokumenta, pristup DOM najčešće zahtjeva korištenje velikog broja objekata unutar memorije. Obrada sadržaja tada je sporija, odnosno zahtjeva više procesorskog vremena, pa se umjesto pristupa DOM vrlo često preporučuje korištenje JSON čitača (Primjer 3.4.3).

#### **Primjer 3.4.3. Čitanje sadržaja JSON datoteke (pristup čitačem)**

---

```
#include <System.JSON.readers.hpp>
```

```

#include <memory>
// ...
void __fastcall TForm1::GUcitajJSONClick(TObject *Sender){
    // učitaj sadržaj datoteke adresar.json
    std::unique_ptr<TStringStream> jsonStream(new TStringStream);
    jsonStream->LoadFromFile("adresar.json");
    UnicodeString jsonDoc = jsonStream->DataString;

    std::unique_ptr<TStringReader> stringReader(new TStringReader(jsonDoc));
    std::unique_ptr<TJsonTextReader> jsonCitac(
        new TJsonTextReader(stringReader.get()));

    UnicodeString Kontakt = "";
    while(jsonCitac->Read()){
        // pročitaj token
        UnicodeString token = jsonCitac->Value.ToString();
        // ukoliko token predstavlja podatak o kontaktu...
        if(token == "ime" || token == "prezime" || token == "ulica" ||
            token == "broj" || token == "mjesto" || token == "telefon"){
            jsonCitac->Read(); // pročitaj podatak pokraj tokena
            Kontakt += jsonCitac->Value.ToString() + " ";
        }
        // nakon što su pročitani svi podaci unutar kontakta...
        if(jsonCitac->TokenType == TJsonToken::EndObject && !Kontakt.IsEmpty()){
            ShowMessage(Kontakt);
            Kontakt = "";
        }
    }
}

```

JSON čitač (instanca klase *TJsonTextReader*) pozivima metode *Read* čita sadržaj JSON dokumenta. Sadržaj se čita token po token, pri čemu je moguće pročitati vrijednost i tip tokena (predstavlja li token svojstvo, početak ili kraj objekta ili niza itd.). Tako prikazani Primjer 3.4.3 prilikom čitanja tokena izolira samo one koji sadrže kontaktne podatke (*ime*, *prezime*, *ulica* itd.), nakon čega se svi zajedno (spojeno) ispisuju za svaki pojedini kontakt.

#### Primjer 3.4.4. Generiranje sadržaja JSON dokumenta

```

#include <System.JSON.builders.hpp>
#include <memory>
// ...
void __fastcall TForm1::GGenerirajJSONClick(TObject *Sender){

```

```
std::unique_ptr<TStringBuilder>    stringBuilder(new TStringBuilder);
std::unique_ptr<TStringWriter>    stringWriter(
                                    new TStringWriter(stringBuilder.get()));
std::unique_ptr<TJsonTextWriter> stringWriter(
                                    new TJsonTextWriter(stringWriter.get()));
stringWriter->Formatting = TJsonFormatting::Indented;
std::unique_ptr<TJSONObjectBuilder> Builder(
                                    new TJSONObjectBuilder(stringWriter.get()));

Builder->BeginObject()
    ->BeginArray("adresar")
        ->BeginObject()
            ->Add("ime", String("Ante"))
            ->Add("prezime", String("Antić"))
            ->Add("ulica", String("Prva ulica"))
            ->Add("broj", 42)
            ->Add("mjesto", String("Zagreb"))
            ->Add("telefon", String("+38512345678"))
        ->EndObject()
        ->BeginObject()
            ->Add("ime", String("Pero"))
            ->Add("prezime", String("Perić"))
            ->Add("ulica", String("Prva ulica"))
            ->Add("broj", 43)
            ->Add("mjesto", String("Zagreb"))
            ->Add("telefon", String("+38522345678"))
        ->EndObject()
    ->EndArray()
->EndObject();
// ispis JSON dokumenta
ShowMessage(stringBuilder->ToString());
}
```

Za generiranje sadržaja u JSON formatu moguće je koristiti klasu *TJSONObjectBuilder* (Primjer 3.4.4). Njena najveća prednost upravo je način generiranja sadržaja jer je izravno iz programskog kôda vidljiva struktura i izgled JSON dokumenta. Alternativno, sadržaj JSON dokumenta uvijek je moguće generirati kao da je riječ o generiranju sadržaja obične tekstualne datoteke.

### 3.5. Razvoj prilagođenog formata

Unatoč svim pogodnostima korištenja prethodno spomenutih formata za pohranu i razmjenu podataka, ponekada je potrebno razviti vlastiti (prilagođeni) format. Novi format najčešće se realizira pohranom podataka u binarnom obliku fiksne duljine. Na početku sadržaja može se nalaziti zaglavlje s osnovnim podacima o formatu (naziv, inačica itd.), dok se u nastavku nalaze zapisi koji predstavljaju glavni sadržaj.

Pretpostavimo da za čitanje i pohranu kontakata iz adresara želimo razviti vlastiti format. Neka njegova struktura izgleda na sljedeći način (Slika 3.5.1):

Naziv formata Inačica	Zaglavlje
ime prezime ulica broj grad telefon	Zapis 1
ime prezime ulica broj grad telefon	Zapis 2
...	

Slika 3.5.1. Struktura novog formata za pohranu kontakata

Pomoću zaglavlja bit će moguće provjeriti je li sadržaj u odgovarajućem formatu. Zaglavlje je fiksne duljine, dok se u nastavku pohranjuju kontakti, odnosno zapisi koji su također fiksne duljine. Strukturu zaglavlja i pojedinog zapisa (kontakta) moguće je definirati pomoću C++ klase ili strukture (Primjer 3.5.1).

#### Primjer 3.5.1. Struktura novog formata

```
// zaglavlje formata...
class MojFormat{
public:
    wchar_t _naziv[10];
    float _inacica;
```



```
MojFormat() {  
    wcsncpy(_naziv, L"MojFormat", 10);  
    _inacica = 1.0;  
}  
};  
  
// kontakt u adresaru...  
  
class Kontakt{  
public:  
    wchar_t _ime[25], _prezime[25], _ulica[50], _grad[50], _telefon[25];  
    int _kucniBroj;  
    Kontakt() {}  
    Kontakt(wchar_t* ime, wchar_t* prezime, wchar_t* ulica,  
            int kucniBroj, wchar_t* grad, wchar_t* telefon) {  
        wcsncpy(_ime, ime, 25);  
        wcsncpy(_prezime, prezime, 25);  
        wcsncpy(_ulica, ulica, 50);  
        _kucniBroj = kucniBroj;  
        wcsncpy(_grad, grad, 50);  
        wcsncpy(_telefon, telefon, 25);  
    }  
};
```

Da bi zaglavlje i zapisi imali fiksnu duljinu korišteni su statički deklarirani znakovni nizovi tipa *wchar\_t*. Ovaj tip predstavlja "široki znak" (eng. *wide character*) koji, za razliku od običnog tipa *char*, za svaki znak koristi 2 bajta. To omogućava korištenje šireg raspona međunarodnih znakova, pa tako i podršku za sve znakove naše abecede.

#### **Primjer 3.5.2. Pohranjivanje zapisa u datoteku novog formata**

```
#include <memory>  
  
// ...  
  
void __fastcall TForm1::GPohraniKontakteClick(TObject *Sender) {  
    Kontakt kontakti[2] = {  
        Kontakt(L"Ante", L"Antić", L"Ulica A", 1, L"Zagreb", L"+38510340"),  
        Kontakt(L"Pero", L"Perić", L"Ulica B", 2, L"Zagreb", L"+38522352")  
    };  
  
    // pohrani zaglavlje formata u memorijski tok  
    MojFormat Zaglavlje;  
    std::unique_ptr<TMemoryStream> kontaktStream(new TMemoryStream);  
    kontaktStream->Write(&Zaglavlje, sizeof(MojFormat));  
  
    // pohrani kontakte u memorijski tok
```

```
    for(int i = 0; i < 2; i++)
        kontaktStream->Write(&kontakti[i], sizeof(Kontakt));
    // pohrani sadržaj memorijskog toka u mft (MojFormat) datoteku
    kontaktStream->SaveToFile("kontakti.mft");
}
```

Za čitanje i pohranu sadržaja u binarnom obliku biblioteka VCL nudi mogućnost korištenja raznih tipova tokova. Tok (eng. *stream*) moguće je zamisliti kao "rijeku podataka" koja ima svoj početak i kraj, dok naš položaj unutar toka ovisi o već obavljenim operacijama čitanja i pisanja. Klasa *TStream* predstavlja baznu klasu za sve tipove tokova, a ovisno o obrađivanom sadržaju i njenoj lokaciji postoje i derivacije te klase poput *TMemoryStream*, *TFileStream*, *TStringStream*, *TResourceStream* itd. [12]

Sukladno strukturi novog formata (Slika 3.5.1) prethodni Primjer 3.5.2 koristi instancu klase *TMemoryStream* kako bismo u radnu memoriju pohranili zaglavlje novog formata, a zatim i dva kontakta iz adresara. Iako se svi podaci inicijalno nalaze u radnoj memoriji, na disk se pohranjuju u binarnom obliku pozivom metode *SaveToFile*. Alternativno, moguće je koristiti i instancu klase *TFileStream*, pri čemu se sve operacije čitanja i pisanja odvijaju izravno u navedenoj datoteci na disku računala. Ipak, ovaj pristup zahtjeva prethodno postojanje datoteke.

#### **Primjer 3.5.3. Čitanje podataka iz datoteke novog formata**

```
void __fastcall TForm1::GUcitajKontakteClick(TObject *Sender){
    // učitaj sadržaj iz datoteke u memorijski tok
    std::unique_ptr<TMemoryStream> kontaktStream(new TMemoryStream);
    kontaktStream->LoadFromFile("kontakti.mft");
    // pročitaj zaglavlje formata
    MojFormat mojFormat;
    kontaktStream->Read(&mojFormat, sizeof(MojFormat));
    // je li ispravan format?
    if(UnicodeString(mojFormat._naziv) != "MojFormat" || mojFormat._inacica != 1.0){
        ShowMessage("Pogrešan format!");
        return;
    }
    // pročitaj sve kontakte
    Kontakt pom;
    int brKontakta = (kontaktStream->Size - sizeof(MojFormat)) / sizeof(Kontakt);
    for(int i = 0; i < brKontakta; i++){
```

```
String kontakt;  
kontaktStream->Read(&pom, sizeof(Kontakt));  
kontakt += String(pom._ime) + " ";  
kontakt += String(pom._prezime) + " ";  
kontakt += String(pom._ulica) + " ";  
kontakt += String(pom._kucniBroj) + " ";  
kontakt += String(pom._grad) + " ";  
kontakt += String(pom._telefon);  
ShowMessage(kontakt);  
}  
}
```

Proces čitanja podataka iz datoteke novog formata najčešće je upravo suprotan procesu njegove pohrane. To demonstrira i prikazani Primjer 3.5.3 u kojemu se sadržaj datoteke na disku prvo učitava u memorijski tok (instancu klase *TMemoryStream*), nakon čega se iz njega čita zaglavlje, a zatim i pojedini zapisi (kontakti iz adresara). Kao i u slučaju pohrane, tako se i prilikom čitanja podataka koristi operator *sizeof* kako bi se uvijek obrađivali blokovi podataka iste veličine.



## 4. Baze podataka

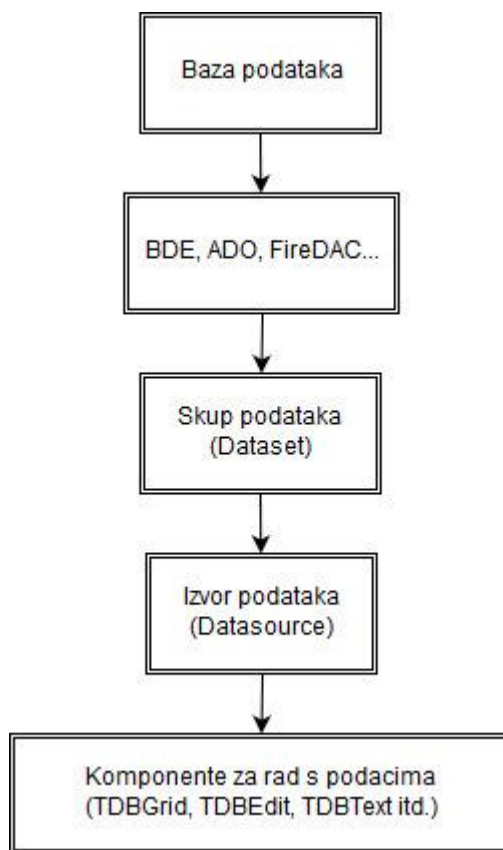
### 4.1. Pristupi i tehnologije

Baza podataka najčešće ima ulogu središnjeg dijela informacijskog sustava kojeg koriste klijenti (ljudi, aplikacije, servisi itd.) za pohranu i obradu podataka. Podaci se nalaze u tablicama, a njihova obrada omogućena je korištenjem jezika SQL. Ovisno o tipu baze podataka, one podržavaju višeklijentski rad, transakcije, izrade izvještaja itd. Poželjno ih je koristiti umjesto običnih datoteka i prilikom rada s velikim količinama podataka, kada im je potrebno ograničiti pristup (administracija) itd.

Biblioteka VCL nudi mogućnost komunikacije s bazama podataka korištenjem više različitih pristupa i tehnologija. Tako je u počecima korištena tehnologija BDE (*Borland Database Engine*), koja se postupno zamijenila programskim okvirom ADO (*Microsoft ActiveX Data Objects*). Ipak, ADO je ograničen samo na Windows platformu, pa je po uzoru na njega razvijena biblioteka *FireDAC* koja koristi vrlo sličan pristup, ali ju je moguće koristiti i u FMX (*FireMonkey*) aplikacijama na platformama Windows, Mac OS X, iOS i Android. Za komunikaciju s bazom podataka biblioteka VCL podržavala je i arhitekturu *dbExpress* koja se danas smatra zastarjelom. [13]

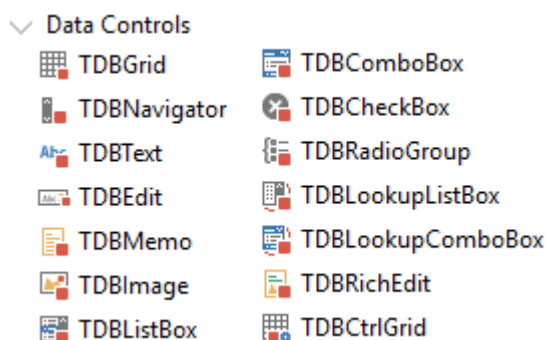
Iako nudi više mogućnosti, *FireDAC* biblioteka ima ograničenja u *Community* i *Professional* izdanjima alata RAD Studio. U tim izdanjima *FireDAC* trenutno dopušta komunikaciju samo s lokalnim bazama podataka (MS Access, SQLite, MySQL Embedded, itd.), dok putem mreže nije moguće komunicirati s bazama podataka poput Oracle, SQL Server itd. Upravo iz ovih razloga veliki broj programera i dalje koristi ADO (*dbGo*) komponente koje u VCL aplikacijama nemaju nikakvih ograničenja i dopuštaju komunikaciju sa svim tipovima baza podataka.

Osim spomenutih, moguće je koristiti i rješenja trećih strana koja najčešće nisu besplatna. Tako je dodatno moguće kupiti UniDAC (*Universal Data Access Components*), *DAC for MySQL*, *PostgresDAC* itd.



Slika 4.1.1. Komunikacija s bazom podataka

Postupak komunikacije s bazom podataka prikazuje Slika 4.1.1. Aplikacija se prvo spaja na bazu podataka korištenjem jedne od tehnologije poput BDE, ADO, FireDAC itd., nakon čega se iz nje učitava jedan ili više skupova podataka. Skup podataka (eng. *dataset*) može biti sadržaj tablice, rezultat izvršavanja SQL upita ili rezultat izvršavanja uskladištene procedure (eng. *stored procedure*).



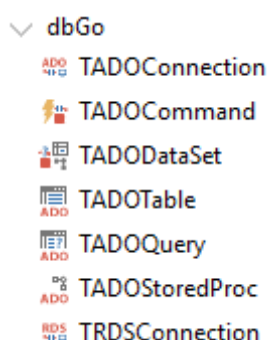
Slika 4.1.2. Data Controls komponente

Da bi se skup podataka mogao obrađivati koriste se *Data Controls* komponente (Slika 4.1.2). Sve komponente iz ove skupine imaju svojstvo *DataSource* (instanca komponente

*TDataSource*), a ono predstavlja posrednika između odabranog skupa podataka i *Data Controls* komponenti (Slika 4.1.1). Kao rezultat, sve *Data Controls* komponente automatski će prikazivati sadržaj koji se nalazi u skupu podataka, a svaka izmjena sadržaja u tim komponentama automatski se pohranjuje nazad u taj skup, odnosno nazad u samu bazu podataka.

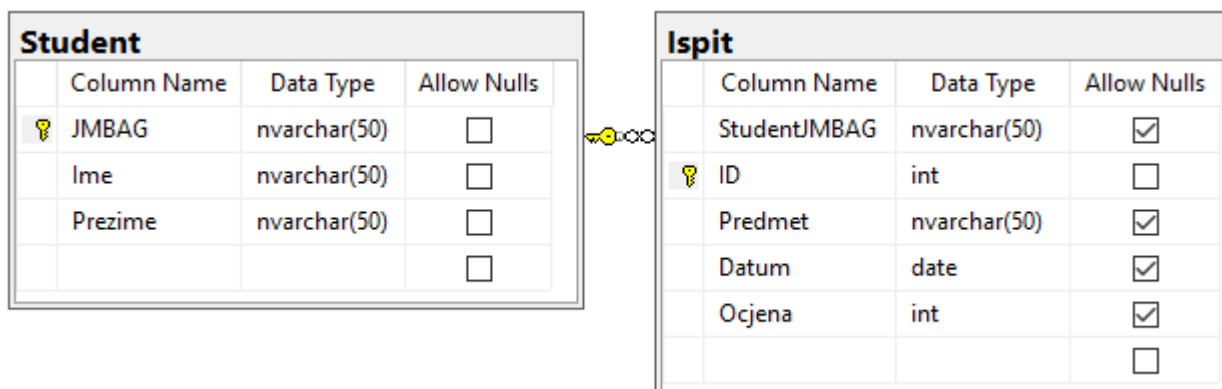
## 4.2. Povezivanje s bazom podataka

Kada je riječ o komunikaciji s bazama podataka, programski okvir ADO i dalje predstavlja jedan od primarnih izbora Delphi i C++ Builder programera. ADO omogućuje komunikaciju s velikim brojem različitih tipova baza podataka, te za razliku od FireDAC biblioteke nije ograničen na komunikaciju samo s lokalnim bazama podataka u *Community* i *Professional* izdanjima RAD Studio alata. Ipak, ADO je ograničen samo na Windows platformu, pa se koristi upravo prilikom razvoja Windows VCL aplikacija.



Slika 4.2.1. ADO komponente

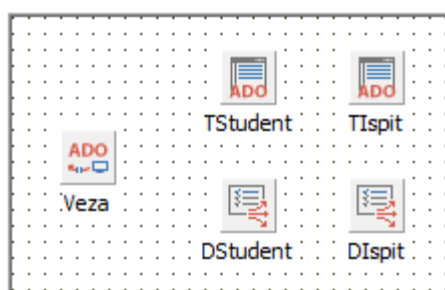
Programski okvir ADO moguće je koristiti pomoću ADO (*dbGo*) komponenti (Slika 4.2.1). Komponenta *TADOConnection* koristi se za povezivanje s bazom podataka, odnosno sadrži sve podatke potrebne za uspostavu veze. Za izvršavanje SQL upita koriste se komponente poput *TADOCommand* (SQL upiti koji ne vraćaju novi set podataka) i *TADOQuery* (ovisno o upitu može vratiti novi set podataka). Komponenta *TADOTable* predstavlja set podataka, odnosno sadržaj odabrane tablice, dok komponenta *TADOStoredProc* služi za izvršavanje uskladištenih procedura. Komponenta *TADOStoredProc* također može sadržavati set podataka ukoliko se on vraća nakon poziva uskladištene procedure.



Slika 4.2.2. SQL Server baza podataka "Studij"

Za primjer, pretpostavimo da postoji SQL Server baza podataka "Studij" koja za pohranu podataka o ocjenama studenata koristi dvije tablice, "Student" i "Ispit" (Slika 4.2.2). Da bismo iz Windows VCL aplikacije omogućili pristup ovoj bazi podataka i njenim podacima koristit ćemo komponente *TADOConnection* i *TADOTable*.

Pristup bazi podataka najčešće je potreban iz više različitih dijaloga (formi). Zbog toga je poželjno da se komponente za komunikaciju s bazom podataka (ADO) nalaze na jednom, centraliziranom mjestu koje je dostupno svim dijalogima u Windows VCL aplikaciji. U tu svrhu prvo je poželjno kreirati podatkovni modul (eng. *data module*) te u njega smjestiti sve potrebne (ADO) komponente.

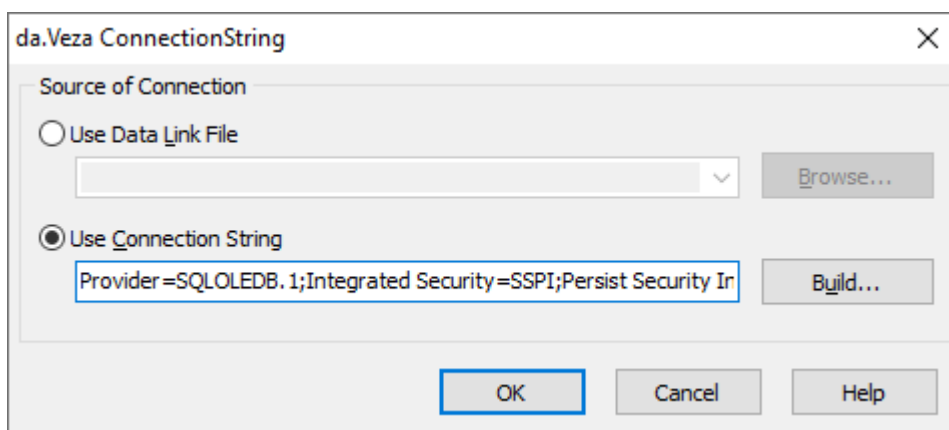


Slika 4.2.3. Podatkovni modul

Podatkovni modul kreira se odabirom stavke *File / New / Other...*, nakon čega je u prikazanom dijalogu pod dijelom *C++ Builder / Individual files* potrebno odabrati stavku *Data Module*. Podatkovni modul predstavlja kontejner za nevizualne komponente, pa je u njega moguće smjestiti komponente *TADOConnection* (*Veza*), *TADOTable* (*TStudent* i *TIspit*) i *TDataSource* (*DStudent* i *DIspit*) (Slika 4.2.3).



U komponenti *TADOConnection* (Veza) potrebno je odrediti vrijednost svojstva *ConnectionString*. To je niz znakova u kojemu su sadržani podaci potrebni za povezivanje s bazom podataka. To su, primjerice, IP adresa i port poslužitelja (lokacija baze podataka), naziv baze podataka, korisničko ime i lozinka itd. (Slika 4.2.4)



Slika 4.2.4. Svojstvo „ConnectionString“

Vrijednost svojstva *ConnectionString* može biti učitana iz datoteke s UDL ekstenzijom (stavka *Use Data Link File*) ili ju je moguće odrediti pomoću čarobnjaka (gumb *Build...*). Korištenje UDL datoteke poželjno je kada postoji mogućnost naknadnog mijenjanja pristupnih podataka (promjena lokacije baze podataka, korisničkog imena, lozinke itd.). Tada pristupne podatke ne želimo pohranjivati u izvršnoj (.exe) datoteci, već ih pri spajanju na bazu podataka uvijek učitavati iz UDL datoteke.

Ukoliko se koristi čarobnjak, za pristup SQL Server bazi podataka potrebno je u kartici *Provider* odabrati *Microsoft OLE DB Provider for SQL Server*, a zatim u kartici *Connection* upisati tražene pristupne podatke i odabrati željenu bazu podataka. Klikom na gumb "Test Connection" odmah je moguće provjeriti jesu li uneseni podaci točni, odnosno je li moguće uspostaviti vezu s bazom podataka.

Ukoliko su korisničko ime i lozinka već navedeni u svojstvu *ConnectionString*, onda je u komponenti *TADOConnection* moguće postaviti svojstvo *LoginPrompt* na vrijednost *false*. Time će se spriječiti nepotrebno pojavljivanje prijavnog (eng. *login*) dijaloga prilikom spajanja na bazu podataka. Konačno, postavljanjem svojstva *Connected* na vrijednost *true* uspostavlja se veza s bazom podataka.

Konfigurirana komponenta *TADOConnection* (Veza) zatim se koristi u komponentama *TADOTable TStudent* i *TIsplit*. Naime, svaka od ovih komponenti ima svojstvo *Connection*, čija vrijednost treba biti ime postojeće *TADOConnection* komponente. Alternativno, umjesto korištenja postojeće *TADOConnection* komponente u svakoj od *TADOTable* komponenti moguće je zasebno odrediti vrijednost svojstva *ConnectionString*.

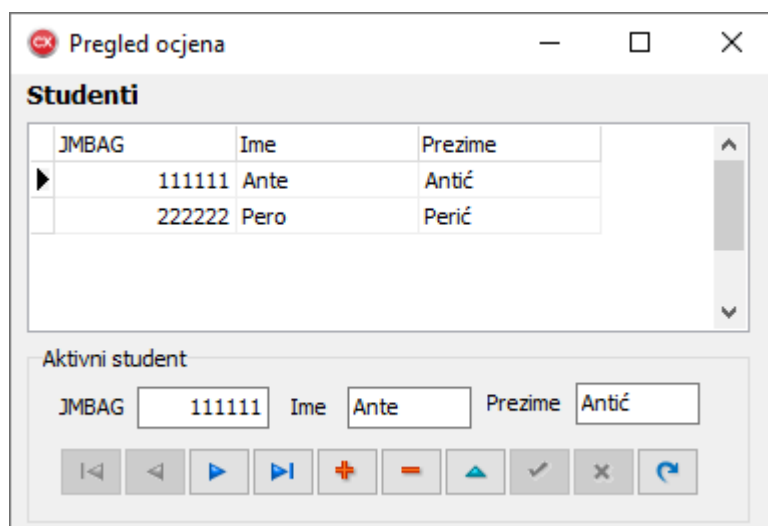
Nakon odabira veze, u svakoj od *TADOTable* komponenti potrebno je odabrati ime tablice iz baze podataka (svojstvo *TableName*) te njen sadržaj (podatkovni skup) učitati u memoriju. To je moguće tako da se svojstvo *Active* postavi na vrijednost *true* ili pozivom metode *Open*.

### 4.3. Obrada podataka

Podatke je moguće obrađivati na dva načina – korištenjem *Data Controls* komponenti i pisanjem programskog kôda. U oba slučaja, u dijalogu (formi) prvo je potrebno uključiti datoteku zaglavlja podatkovnog modula.

```
#include "dm.h" // datoteka zaglavlja podatkovnog modula DM1
```

Sada je iz odabranog dijaloga moguće pristupiti svim komponentama smještenima u podatkovnom modulu *DM1* (Slika 4.2.3).



Slika 4.3.1. Obrada podataka iz tablice "Student"

Za prikaz podataka u obliku rešetke koristi se komponenta *TDBGrid*. Kao i sve ostale *Data Controls* komponente, ona sadrži svojstvo *DataSource* pomoću kojeg se određuje skup podataka koji se prikazuje i obrađuje u komponenti. Specifičnost komponente *TDBGrid* je što u pojedinim recima prikazuje sadržaj cijelog podatkovnog seta, dok ostale *Data Controls* komponente poput *TDBEdit*, *TDBText* itd. prikazuju sadržaj samo jednog stupca aktivnog zapisa, gdje se stupac određuje svojstvom *DataField*. Aktivni zapis označen je u komponenti *TDBGrid* korištenjem strelice koja se nalazi na početku retka (Slika 4.3.1).

Kada unutar komponente *TDBGrid* želimo promijeniti vrijednost nekog podatka, potrebno je dvostrukim klikom lijeve tipke miša otvoriti ćeliju s podatkom te u ugrađenom uređivaču teksta (editoru) unijeti novu vrijednost. Unesena vrijednost automatski će biti pohranjena u podatkovni skup, odnosno bazu podataka, nakon što se promijenit aktivni zapis ili pozove metoda *Post*. Općenito, prilikom obrade podataka korištenjem *Data Controls* komponenti dobro je koristiti komponentu *TDBNavigator* koja automatski nudi mogućnost dodavanja novih zapisa, uređivanja, brisanja i pohrane promjena postojećih zapisa.

#### **Primjer 4.3.1. Sekvencijalno čitanje svih zapisa u podatkovnom setu**

```
// pozicioniraj kursor na početak (prvi zapis)
DM1->TStudent->First();
for(int i = 0; i < DM1->TStudent->RecordCount; i++){
    // pročitaj i ispiši pojedine stupce aktivnog zapisa
    String jmbag = DM1->TStudent->FieldByName("JMBAG")->AsString;
    String ime = DM1->TStudent->FieldByName("Ime")->AsString;
    String prezime = DM1->TStudent->FieldByName("Prezime")->AsString;
    ShowMessage(jmbag + " " + ime + " " + prezime);
    // postavi kursor na sljedeći zapis
    DM1->TStudent->Next();
}
```

Osim pomoću *Data Controls* komponenti, podacima je moguće pristupiti i obrađivati ih korištenjem sljedećih metoda:

- *First* – pozicioniraj kursor na početak (prvi zapis je aktivni zapis)
- *Last* – pozicioniraj kursor na kraj (zadnji zapis je aktivni zapis)
- *Next* – pozicioniraj kursor na sljedeći zapis u podatkovnom setu
- *FieldByName* – dohvati podatak iz određenog stupca

- `Locate` – pronađi zapis u podatkovnom setu
- `Append` – dodaj novi zapis
- `Edit` – uredi aktivni zapis
- `Post` – pohrani promjene nad aktivnim zapisom
- `Delete` – briši aktivni zapis
- itd.

Korištenje nekih od navedenih metoda demonstrira Primjer 4.3.1 pri radu s komponentom *TADOTable*. Za sekvencijalno čitanje svih zapisa iz tablice *Student* (podatkovni set *TStudent*) korištene su metode *First* i *Next*, dok se za dohvat podataka iz pojedinih stupaca koristi metoda *FieldByName*.

#### **Primjer 4.3.2. Dodavanje novog zapisa u podatkovni set**

```
DM1->TStudent->Append(); // dodaj novi zapis
DM1->TStudent->FieldByName("JMBAG")->AsString = "333333";
DM1->TStudent->FieldByName("Ime")->AsString = "Marko";
DM1->TStudent->FieldByName("Prezime")->AsString = "Marković";
DM1->TStudent->Post(); // pohrani zapis
```

Za dodavanje novog zapisa u bazu podataka koristi se metoda *Append* (Primjer 4.3.2). U podatkovnom setu stvara se dodatni redak za novi zapis, a vrijednosti njegovih stupaca moguće je odrediti metodom *FieldByName*. Konačno, novi zapis potrebno je pohraniti pozivom metode *Post*.

Ažuriranje postojećeg zapisa provodi se na identičan način kao u prethodnom primjeru. Razlika je tek što se umjesto metode *Append* poziva metoda *Edit*. Ova će metoda aktivni zapis postaviti u status uređivanja, nakon čega će postojeće vrijednosti biti moguće prepisati novima, a izmjene pohraniti pozivom metode *Post*. Aktivni zapis moguće je i izbrisati iz baze podataka pozivom metode *Delete*.

#### **Primjer 4.3.3. Lociranje i brisanje zapisa iz podatkovnog seta**

```
TLocateOptions Opts;
Opts.Clear();
Opts << loCaseInsensitive; // zanemari velika i mala slova

// Postoji li tablici Student student s imenom "Ante" ili "ante"?
```

```
if(DM1->TStudent->Locate("Ime", "ante", Opts) == true){  
    // izbriši studenta iz tablice  
    DM1->TStudent->Delete();  
    ShowMessage("Student je pronađen i izbrisan iz baze podataka!");  
}  
else  
    ShowMessage("Student nije pronađen!");
```

Operacije ažuriranja i brisanja provode se nad aktivnim zapisom, pa je vrlo često prije tih operacija prvo potrebno pronaći (locirati) željeni zapis. To je omogućeno metodom *Locate* (Primjer 4.3.3). Njen prvi parametar ime je jednog ili više stupaca koje je potrebno pretraživati, dok drugi parametar određuje traženu vrijednost. Treći parametar predstavlja instancu *TLocateOptions* objekta kojim je moguće odrediti dodatne postavke pretrage (primjerice, ignorirati razliku između velikih i malih slova).

Metoda *Locate* vratit će vrijednost *true* ukoliko uspije pronaći traženi zapis. U tom slučaju pronađeni zapis postat će aktivni zapis, pa je nad njime moguće izravno pozvati metode *Edit* i *Delete*. Ukoliko u tablici postoji više zapisa koji zadovoljavaju kriterije pretrage, aktivni zapis postat će prvi pronađeni zapis.

Zapise u podatkovnom setu moguće je i proizvoljno sortirati tako da se svojstvo *IndexFieldNames* inicijalizira na odgovarajuću vrijednost. Primjerice, nakon naredbe

```
DM1->TStudent->IndexFieldNames = "Ime;Prezime";
```

zapisi u podatkovnom setu bit će sortirani prvo po imenu pa zatim po prezimenu studenata. Vrijednosti se podrazumijevano sortiraju od najmanje prema najvećoj. Kada je potreban obrnuti redoslijed, nakon naziva stupca treba dodati i riječ *DESC*. Primjerice, "Ime DESC;Prezime".

Podatkovni set moguće je i filtrirati. Dva su načina implementacije – pomoću svojstva *Filter* i događaja *OnFilterRecord*.

#### **Primjer 4.3.4. Filtriranje podatkovnog seta (svojstvo *Filter*)**

```
DM1->TStudent->Filtered = false;  
DM1->TStudent->Filter = "JMBAG LIKE " + QuotedStr("1%");
```

```
DM1->TStudent->Filtered = true;
```

Primjer 4.3.4 prikazuje podatkovni set koji se filtrira tako da se u njemu prikazuju samo zapisi studenta čiji JMBAG počinje brojem 1. Osim operatora *LIKE*, u svojstvu *Filter* moguće je koristiti i operatore IS, NOT, IN, OR, =, <, >, <> itd. Prikazani pristup pogodan je i kada uvjet filtriranja treba određivati dinamički, odnosno tijekom rada aplikacije, dok se za kompliciranije filtere koristi događaj *OnFilterRecord*.

#### Primjer 4.3.5. Filtriranje podatkovnog seta (događaj *OnFilterRecord*)

```
void __fastcall TDM1::TStudentFilterRecord(TDataSet *DataSet, bool &Accept) {
    String ime = DataSet->FieldByName("Ime")->AsString;
    // ako je početno slovo imena malo slovo
    if(ime[1] >= 'a' && ime[1] <= 'z')
        Accept = true;
    else
        Accept = false;
}
```

Primjer 4.3.5 demonstrira filter koji korištenjem događaja *OnFilterRecord* prikazuje zapise samo studenata čije ime počinje malim slovom. Izrazom *Accept* svaki zapis u podatkovnom setu bit će prikazan (*Accept = true*) ili skriven, ovisno o uvjetu filtriranja. I kod ovog pristupa filtriranja podatkovnog seta inicira se postavljanjem svojstva *Filtered* na vrijednost *true*.

Komponenta *TADOTable* uvijek učitava sve zapise iz odabrane tablice, dok se operacije pretrage, sortiranja i filtriranja implementiraju korištenjem prethodno opisanih svojstava, metoda i događaja. Zbog toga je ponekad bolje koristiti komponentu *TADOQuery* koja uz pomoć jezika SQL može sve te operacije implementirati puno jednostavnije (Primjer 4.3.6).

#### Primjer 4.3.6. Izvršavanje SQL upita (komponenta *TADOQuery*)

```
ADOQuery1->Close();
ADOQuery1->SQL->Text = "SELECT * FROM Student "
                      "WHERE JMBAG LIKE '1%' "
                      "ORDER BY Ime, Prezime";
ADOQuery1->Open();
```

Kada se kao rezultat izvršavanja SQL upita očekuje novi podatkovni set (jedan ili više zapisa) onda se poziva metoda *Open* ili se svojstvo *Active* postavlja na vrijednost *true*. Za

izvršavanje ostalih tipova upita koji kao rezultat ne vraćaju novi podatkovni set (primjerice, INSERT, DELETE, UPDATE itd.) poziva se metoda *ExecSQL*.

SQL upiti mogu biti pohranjeni i unutar uskladištene procedure (eng. *stored procedure*), a za njeno izvršavanje koristi se komponenta *TADOStoredProc*.

#### **Primjer 4.3.7. Uskladištena procedura "DohvatiStudente"**

```
CREATE OR ALTER PROCEDURE DohvatiStudente
    @n int = 0
AS
BEGIN
    IF @n = 0
        SELECT * FROM Student;
    ELSE
        SELECT TOP (@n) * FROM Student;
END
```

Uskladištena procedura iz prikazanog primjera (Primjer 4.3.7) ima parametar *@n* s podrazumijevanom vrijednosti 0. To nam omogućuje da ju pomoću komponente *TADOStoredProc* pozovemo na sljedeći način (Primjer 4.3.8):

#### **Primjer 4.3.8. Poziv uskladištene procedure**

```
ADOSToredProc1->Close();
ADOSToredProc1->ProcedureName = L"DohvatiStudente";
ADOSToredProc1->Open();
```

Kao i u slučaju s komponentom *TADOQuery*, metoda *Open* koristi se kada za rezultat izvršavanja upita (uskladištene procedure) očekujemo novi podatkovni set. U protivnom, poziva se metoda *ExecProc*.

#### **Primjer 4.3.9. Pozivanje uskladištene procedure s parametrom**

```
ADOSToredProc1->Close();
ADOSToredProc1->ProcedureName = L"DohvatiStudente";

// dodavanje parametra uskladištene procedure
TParameter* param = ADOSToredProc1->Parameters->AddParameter();
param->Name = L"@n";
param->DataType = ftInteger;
ADOSToredProc1->Parameters->Refresh();
```

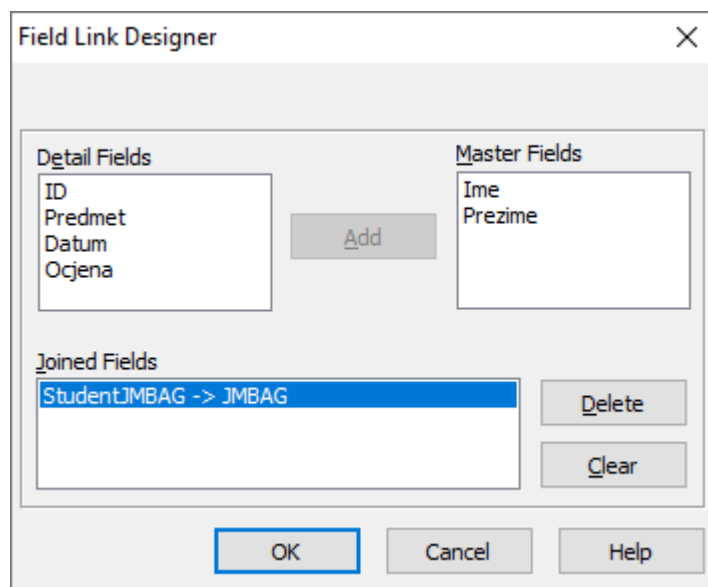
```
// postavljanje vrijednosti parametra @n
ADOSToredProc1->Parameters->ParamByName("@n")->Value = 2;
ADOSToredProc1->Open();
```

Predaja parametara uskladištenoj proceduri zahtjeva prethodno kreiranje tih parametara u komponenti *TADOSToredProc*. To je moguće napraviti pomoću objektnog inspektora (svojstvo *Parameters*) ili pomoću programskog kôda (Primjer 4.3.9). Tek nakon toga inicijalizira se vrijednost parametara i poziva uskladištena procedura.

## 4.4. Odnos *master-detail*

Ukoliko za jedan zapis iz tablice roditelja može postojati više zapisa u tablici djeteta, tada govorimo o *master-detail* odnosu između tih dviju tablica. Za primjer, tablice *Student* i *Ispit* (Slika 4.2.2) su u odnosu *master-detail* jer za svakog studenta može postojati više zapisa o njegovim položenim ispitima.

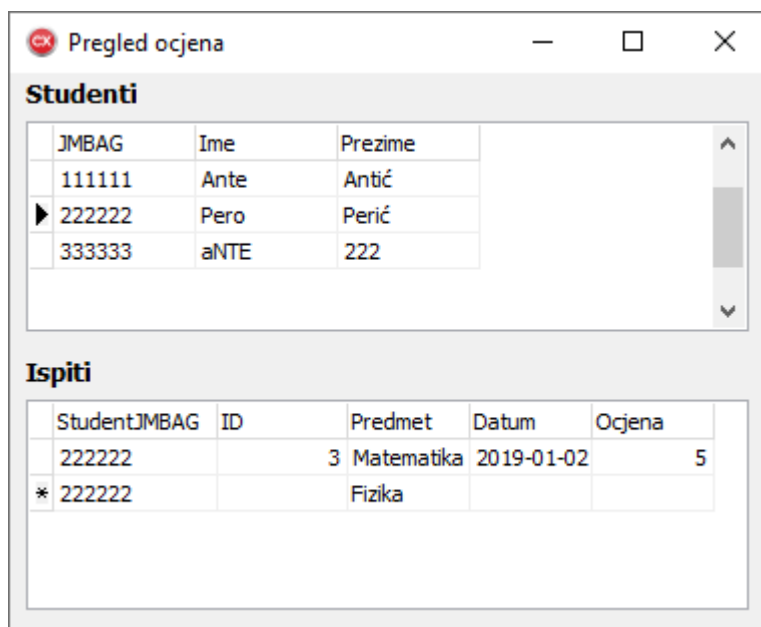
Implementacija odnosa *master-detail* nije obavezna unutar aplikacije, ali nudi određene pogodnosti. Primjerice, podatkovni set tablice djeteta automatski će se filtrirati ovisno o aktivnom zapisu tablice roditelja, a prilikom dodavanja novog zapisa u tablicu djeteta automatski će se inicijalizirati vrijednost njegovog stranog ključa.



Slika 4.4.1. Povezivanje primarnog i stranog ključa



ADO komponente omogućuju implementaciju *master-detail* odnosa na vrlo jednostavan način. U komponenti *TIsplit* potrebno je svojstvo *MasterSource* postaviti na vrijednost *DStudent* (Slika 4.2.3). Ovime se za tablicu djeteta (*Ispit*) određuje njegov roditelj (tablica *Student*). Konačno, potrebno je povezati primarni ključ tablice roditelja i strani ključ tablice djeteta korištenjem svojstva *MasterFields* (Slika 4.4.1).



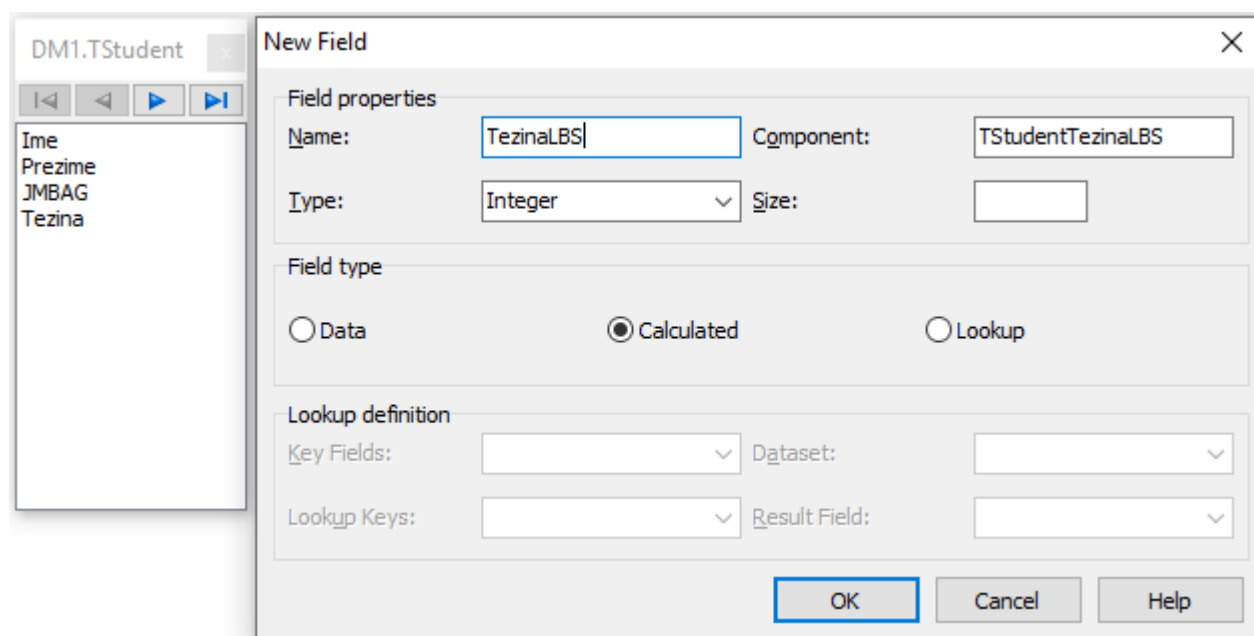
Slika 4.4.2. Unos novog zapisa u tablicu djeteta

Učinak povezivanja primarnih i stranih ključeva prikazuje Slika 4.4.2. U tablici djeteta (*Ispit*) vidljivi su samo zapisi čija vrijednost stranog ključa *StudentJMBAG* odgovara vrijednosti primarnog ključa *JMBAG* aktivnog zapisa tablice roditelja. Tako su sada za studenta s JMBAG-om 222222 prikazani samo njegovi ispiti, odnosno samo ispiti čiji strani ključ ima vrijednost 222222.

Kada bi se aktivni zapis tablice roditelja promijenio, tada bi se u tablici djeteta automatski prikazali (filtrirali) samo oni zapisi koji odgovaraju novoodabranom studentu. Štoviše, prilikom dodavanja novog zapisa u tablicu djeteta, vrijednost njegovog stranog ključa automatski bi bila inicijalizirana vrijednošću primarnog ključa aktivnog zapisa tablice roditelja.

## 4.5. Posebni tipovi polja

Ponekad je u podatkovni set potrebno dodati nova pomoćna polja koja ne postoje u bazi podataka. To mogu biti nova podatkovna (eng. *data*), izračunata (eng. *calculated*) ili *lookup* polja. Podatkovna polja predstavljaju tek dodatne stupce kojih u bazi podataka inače nema, no u aplikaciji se mogu koristiti za privremenu pohranu i obradu dodatnih podataka. Za razliku od njih, izračunata polja namijenjena su samo za čitanje (*read-only*), a njihova vrijednost automatski se izračunava po unaprijed određenom pravilu (formuli).



Slika 4.5.1. Kreiranje novog izračunatog polja

Novo polje u podatkovnom setu najjednostavnije je kreirati tijekom dizajniranja. Dvostrukim klikom na komponentu *TStudent* pojavit će se dijalog u kojemu je desnim klikom moguće dodati sva već postojeća polja tablice *Student* te kreirati nova polja.

Pretpostavimo da tablica *Student* već ima polje *Tezina* koje predstavlja težinu studenta u kilogramima. Ukoliko težinu u kilogramima želimo u dodatnom stupcu prikazati i kao težinu u funtama (LBS) možemo kreirati izračunato polje *TezinaLBS* (Slika 4.5.1).

### Primjer 4.5.1. Implementacija izračunatog polja

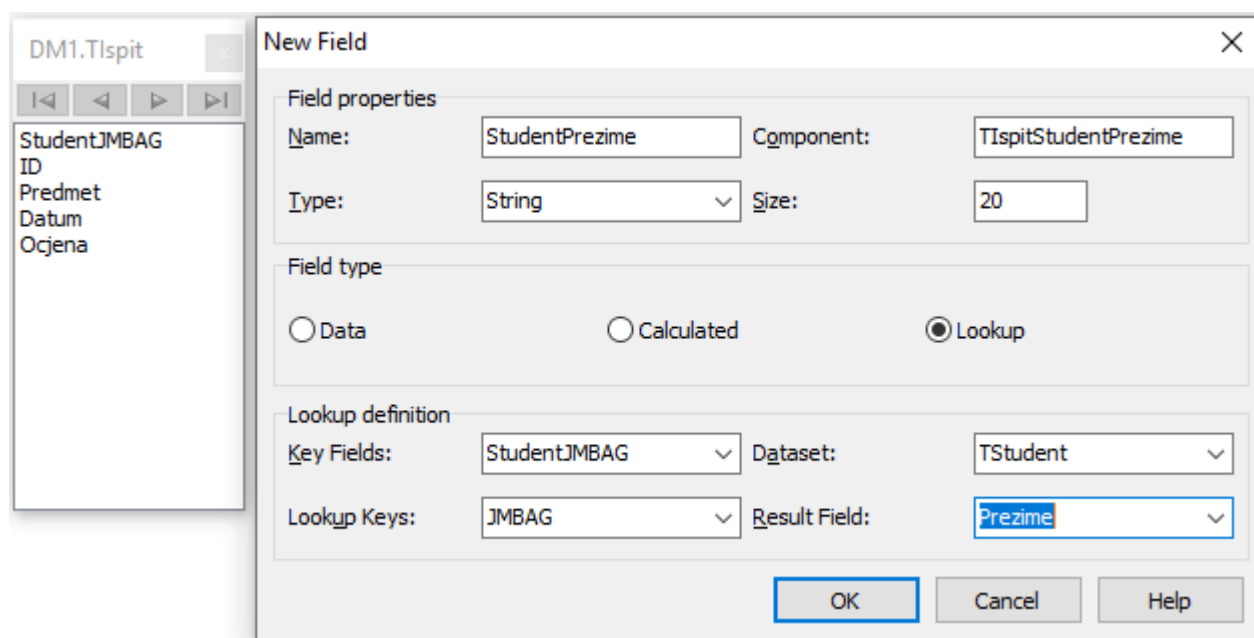
```
void __fastcall TDM1::TStudentCalcFields(TDataSet *DataSet) {
    int kg = DataSet->FieldByName("Tezina")->AsInteger;
    DataSet->FieldByName("TezinaLBS")->AsInteger = kg * 2.20462262185;
```

```
}

```

Da bi odredili kako se računa vrijednost izračunatog polja *TezinaLBS* potrebno je napisati implementaciju događaja *OnCalcFields* nad komponentom *TStudent* (Primjer 4.5.1). Prikazana implementacija tj. izračun izvršit će se za svaki zapis u podatkovnom setu. Zato je poželjno izbjegavati komplicirane izračune koji dugo traju kako se pri radu s većim podatkovnim setovima ne bi usporio rad aplikacije.

Osim podatkovnih i izračunatih, moguće je koristiti i polja tipa *lookup* čija vrijednost se dohvaća iz tablice roditelja. Primjerice, ukoliko bi u popisu ispita uz JMBAG studenta (strani ključ *StudentJMBAG*) htjeli prikazati i stupac s prezimenom tog studenta onda bi u komponenti *TIsplit* mogli kreirati novo polje tipa *lookup*.



Slika 4.5.2. Kreiranje novog polja tipa *lookup*

Novo *lookup* polje *StudentPrezime* (Slika 4.5.2) svoju vrijednost će dohvatiti tako da će se u podatkovnom setu tablice roditelja *TStudent*, u njegovom polju *JMBAG* tražiti vrijednost *StudentJMBAG*. Kada se pronađe odgovarajući zapis, vrijednost polja *Prezime* pronađenog zapisa bit će pohranjena u polje *StudentPrezime*.

I polja tipa *lookup* mogu drastično usporiti rad aplikacije, a pogotovo kada se radi s podatkovnim setovima koji sadrže veliki broj zapisa. Naime, polja tipa *lookup* sadrže

svojstvo *LookupCache* koje podrazumijevano ima vrijednost *false*. Zbog toga će se prilikom svake promjene aktivnog zapisa ponovno dohvatiti vrijednosti svih njegovih *lookup* polja. Ukoliko pak znamo da se podaci iz tablice roditelja neće mijenjati (samo su za čitanje) onda se vrijednost svojstva *LookupCache* može postaviti na *true*, te time uvelike poboljšati performanse aplikacije. [14]

Biblioteka VCL sadrži i dvije *lookup* komponente. Komponentama *TDBLookupListBox* i *TDBLookupComboBox* zapise odabrane tablice možemo koristiti kao popise stavki. U tu svrhu, obje komponente sadrže sljedeća zajednička svojstva;

- *ListSource* – izvor podataka
- *ListField* – polje iz kojeg se čita popis stavki
- *KeyField* – ključno polje
- *DataSource* – odredišni izvor podataka
- *DataField* – odredišno polje

#### **Primjer 4.5.2. Konfiguracija komponente *TDBLookupComboBox***

---

```
// izvor podataka
DBLookupComboBox1->ListSource = DM1->DStudent;
DBLookupComboBox1->ListField = "Prezime";
DBLookupComboBox1->KeyField = "JMBAG";
// odredište
DBLookupComboBox1->DataSource = DM1->DISpit;
DBLookupComboBox1->DataField = "StudentJMBAG";
```

Primjer 4.5.2 demonstrira kako možemo konfigurirati komponentu *TDBLookupComboBox*. Kao popis stavki bit će ponuđen popis svih prezimena iz tablice *Student*, dok će se *JMBAG* odabranog studenta pohraniti u polje *StudentJMBAG* tablice *Ispit*.

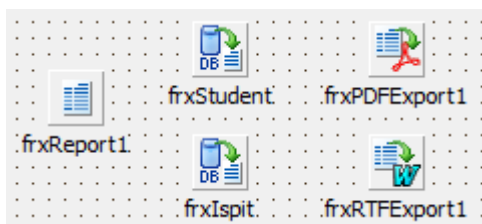
## **4.6. Generiranje izvještaja**

Kroz povijest, RAD Studio alati nudili su više različitih generatora izvještaja. U najranijim inačicama podrazumijevano je instaliran i korišten Quickreport, pa zatim Rave Reports, dok danas programeri mogu izabrati između više različitih rješenja. Tako su tu i rješenja poput Fast Report, FortesReport, ReportBuilder itd. U svrhu demonstracije u

nastavku će biti korišten Fast Report (VCL) kojeg je prethodno potrebno preuzeti i instalirati iz *GetIt Package Manager* repozitorija (Slika 1.1.1).

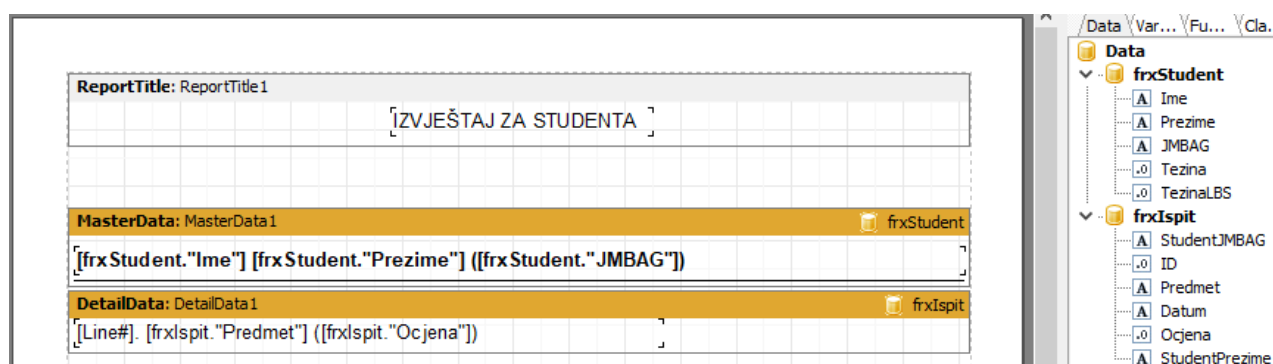
Nakon instalacije, Fast Report komponente bit će dostupne u paleti s alatima. Grupirane su u nekoliko skupina, a najvažnije od njih su:

- *TfrxReport* – dizajn izvještaja
- *TfrxDBDataset* – pristup podatkovnom setu (podacima iz baze podataka)
- *Tfrx\*Export* – izvoz izvještaja u PDF, RTF, HTML, BMP, JPEG, TIFF i sl. formatima



Slika 4.6.1. Fast Report komponente

Pretpostavimo da želimo generirati izvještaj o pojedinom studentu i njegovim položenim ispitima. Prvo nam je potrebna komponenta *TfrxReport* (Slika 4.6.1, *frxReport1*) te dvije *TfrxDBDataset* komponente (*frxStudent* i *frxIspit*) za pristup podacima iz baze podataka. Također, da bismo omogućili pohranu (izvoz) podataka u PDF i RTF formatima potrebne su i komponente *TfrxPDFExport* i *TfrxRTFExport*.



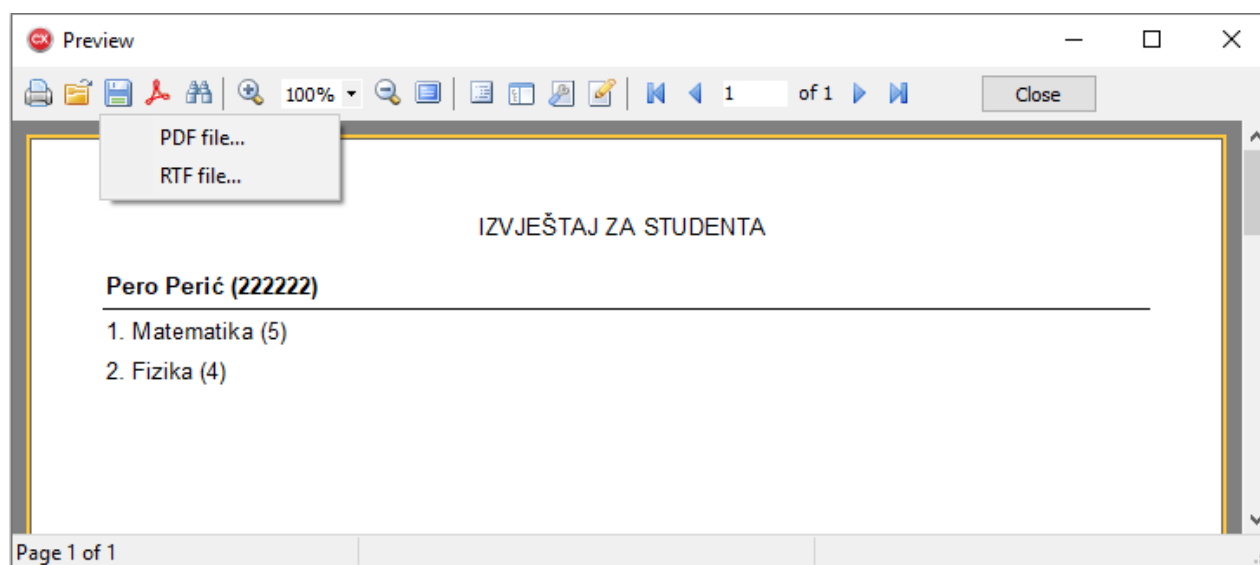
Slika 4.6.2. Dizajner izvještaja

Dizajner izvještaja pojavljuje se nakon dvostrukog klika na komponentu *TfrxReport* (Slika 4.6.2). Unutar dizajnera prvo je potrebno u izborniku *Report / Data...* uključiti prethodna dva podatkovna seta *frxStudent* i *frxIspit*. Tada se s desne strane dizajnera pojavljuju polja iz oba podatkovna seta koja je sada moguće odvući na trake (eng. *bands*) u dizajneru i

tako dizajnirati izvještaj. Dvostrukim klikom na pojedino polje u traci moguće je to polje dodatno urediti i spojiti ga s nekim drugim poljima.

Izvještaj sa slike ima traku s naslovom (*ReportTitle*), traku s podacima o studentu (*MasterData*) te traku s podacima o ispitima studenta (*DetailData*). Traka s naslovom pojavljuje se samo jednom, na početku izvještaja. Ovisno o tome želimo li prikazati izvještaj samo za jednog (aktivnog) studenta ili za sve njih, traka *MasterData* pojavit će se jednom ili više puta na izvještaju, dok se traka *DetailData* ponavlja onoliko puta koliko postoji zapisa (ispita) za pojedinog studenta.

Da bismo odredili generira li se izvještaj samo za jednog ili sve studente, potrebno je svojstva *RangeBegin* i *RangeEnd* (komponenta *frxStudent*) postaviti na vrijednosti *rbCurrent* i *reCurrent* ili na vrijednosti *rbFirst* i *reLast*.



Slika 4.6.3. Pregled izvještaja

Pregled izvještaja moguć je tijekom dizajna (izbornik *File / Review*) ili tijekom rada aplikacije pozivom metode *ShowReport*. Kako izgleda prethodno dizajnirani izvještaj prikazuje Slika 4.6.3. Također je potrebno primijetiti da je zbog komponenti *TfrxPDFExport* i *TfrxRTFExport* moguća pohrana (izvoz) generiranog izvještaja u PDF i RTF formatima.

U vrijeme dizajna izvještaja moguće je odrediti većinu njegovog sadržaja. Međutim, izvještaj i njegove dijelove moguće je uređivati i korištenjem programskog kôda.

**Primjer 4.6.1. Uređivanje sadržaja izvještaja korištenjem programskog kôda**

```
TfrxMemoView* Naslov;  
Naslov = dynamic_cast<TfrxMemoView*>(frxReport1->FindObject("Memo1"));  
Naslov->Text = L"Naslov izvještaja";
```

Kada je potrebno programski urediti sadržaj izvještaja, pristupa se objektima (poljima) koji predstavljaju pojedine dijelove izvještaja. Primjerice, ukoliko je naslov prethodnog izvještaja pohranjen u komponenti *TfrxMemoView* imena *Memo1*, sadržaj tog naslova može se izmijeniti pomoću programskog kôda na način koji prikazuje Primjer 4.6.1.

U prethodnom tekstu prikazane su tek osnovne značajke Fast Report generatora izvještaja koji u punoj (plaćenj) inačici nudi mnogo više mogućnosti. Ipak, i u ovoj ograničenoj (besplatnoj) inačici nudi sasvim dovoljno za izradu većine potrebnih tipova izvještaja.

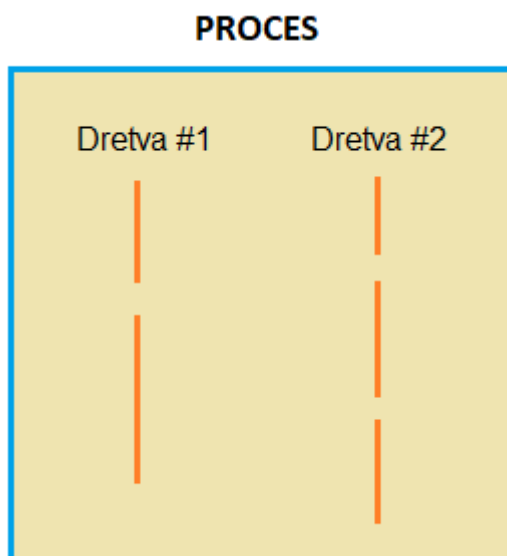




## 5. Dretve i procesi

### 5.1. Dretve

Dretva ili nit (eng. *thread*) predstavlja niz instrukcija u procesu koji se može izvršavati neovisno o ostalom programskom kôdu. Svaki proces (instancija pokrenutog programa) sastoji se od barem jedne dretve, a po potrebi može ih kreirati i više. Na taj način jedan proces može paralelno izvršavati više nizova instrukcija te tako brže rješavati određene programske zadatke (Slika 5.1.1).



Slika 5.1.1. Proces sa dvije dretve

Bez obzira na broj izrađenih dretvi, maksimalni broj zadataka koji se može izvršavati u isto vrijeme ovisi o procesoru. Općenito, procesor s više jezgri podržavat će rad s minimalno isto toliko dretvi. Ukoliko procesor podržava tehnologiju *HyperThreading* (HT) ili *Simultaneous MultiThreading* (SMT), tada on po svakoj jezgri podržava i dvije dretve. [15]

Processes	Threads	Handles
195	3514	109211

Slika 5.1.2. Broj procesa i dretvi u operacijskom sustavu

Operacijski sustav može izvršavati po nekoliko tisuća dretvi. Međutim, ukoliko naš procesor podržava istovremeni rad samo 16 dretvi, onda će se njih 3514 (Slika 5.1.2)

izvršavati tako da će se svakoj dretvi naizmjenično davati mali dio procesorskog vremena. Na taj način prividno se postiže paralelizam u radu, dok se u stvarnosti u svakom trenutku izvršava tek 16 dretvi.

Za demonstraciju pretpostavimo da u klasi glavnog dijaloga (podrazumijevano *TForm1*) postoji vektor cijelih brojeva `vector<int> brojevi`. Neka taj vektor sadrži 1 milijun slučajno generiranih brojeva, pri čemu je svaki generirani broj u intervalu [1, 1.000.000] (Primjer 5.1.1).

#### Primjer 5.1.1. Inicijalizacija elemenata vektora slučajno generiranim brojevima

```
__fastcall TForm1::TForm1(TComponent* Owner): TForm(Owner) {
    brojevi = vector<int>(1000000);
    srand((unsigned)time(NULL));
    for (int i = 0; i < brojevi.size(); i++)
        brojevi[i] = rand() % brojevi.size() + 1;
}
```

Zadatak je programa ispisati koliko od tih slučajno generiranih brojeva su prim brojevi (brojevi koji su djeljivi samo brojem 1 i samim sobom).

#### Primjer 5.1.2. Pretraživanje prim brojeva u glavnoj dretvi

```
#include <chrono>
// ...
using namespace std::chrono;
// funkcija koja vraća je li predani broj prim broj
bool IsPrim(int broj) {
    if (broj <= 1) return false;
    for (int i = 2; i <= floor(pow(broj, 0.5)); i++)
        if (broj % i == 0)
            return false;
    return true;
}
// provjera svih brojeva u vektoru korištenjem samo jedne (glavne) dretve
void __fastcall TForm1::SingleThreadedClick(TObject *Sender) {
    auto start = high_resolution_clock::now();
    int primKol = 0;
    for (int i = 0; i < brojevi.size(); i++)
        if (IsPrim(brojevi[i]))
```

```
        primKol++;  
    auto end = high_resolution_clock::now();  
    auto duration = duration_cast<milliseconds>(end - start).count();  
    ShowMessage("Prim brojeva: " + IntToStr(primKol) +  
        ", dretve: 1, vrijeme: " + IntToStr(duration) + " ms.");  
}
```

Prvo i implementacijski najjednostavnije rješenje jest da pomoću *for* petlje u glavnoj dretvi sekvencijalno provjerimo sve brojeve u vektoru (Primjer 5.1.2). Ovisno o računalu, zbog velike količine brojeva, taj postupak traje oko 8 sekundi. Glavna dretva je tijekom tog vremena zauzeta (aplikacija je "zamrznuta") te korisnik nije u mogućnosti koristiti ostale funkcionalnosti aplikacije.

Umjesto da jedna (glavna) dretva provjerava sve brojeve, taj posao možemo raspodijeliti na više dretvi. Primjerice, ukoliko bi postojale 4 dretve, svaka od njih bi mogla paralelno provjeravati svoj dio podataka. Primjerice,

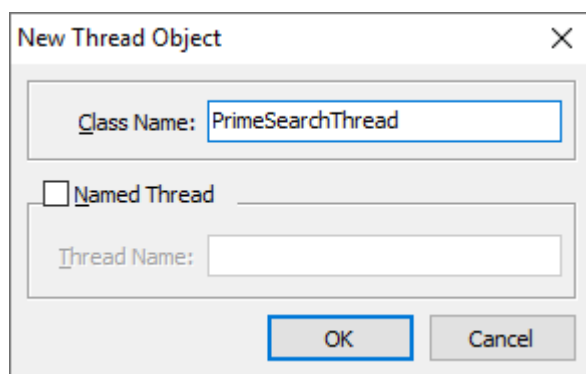
Dretva #1 - (elementi vektora 0 – 249999)

Dretva #2 - (elementi vektora 250000 – 499999)

Dretva #3 - (elementi vektora 500000 – 749999)

Dretva #4 - (elementi vektora 750000 – 999999)

Za kreiranje dretvi u C++ Builderu aplikacijama moguće je koristiti apstraktnu klasu *TThread*. Njenim nasljeđivanjem izrađuje se nova klasa u kojoj se implementira ponašanje dretve, dok objekti te klase predstavljati pojedine dretve.



Slika 5.1.3. Izrada nove dretve

Novu dretvu moguće je izraditi odabirom stavke *File / New / Other...* U prikazanom dijalogu (Slika 1.3.1) potrebno je odabrati stavku *C++ Builder / Individual Files / Thread Object*. Tada se pojavljuje dijalog kojeg prikazuje Slika 5.1.3. Stavka *Class Name* predstavlja ime nove derivacije *TThread* klase, dok stavkom *Named Thread* dretvi možemo dati ime po kojem će ju biti jednostavnije pronaći u dijalogu *Thread Status (Debugger)*.

#### **Primjer 5.1.3. PrimeSearchThread - derivacija klase TThread**

```
#include <vector>
// ...
class PrimeSearchThread: public TThread{
private:
protected:
    void __fastcall Execute(); // definira kod koji se izvršava u dretvi
public:
    std::vector<int>& niz; // referenca na predani vektor
    int start, kraj, primKol; // područje niza i količina prim brojeva
    __fastcall PrimeSearchThread(std::vector<int>& niz,
                                int start, int kraj,
                                bool CreateSuspended);
};
```

Klikom na gumb *OK* generirat će se klasa *PrimeSearchThread*, derivacija klase *TThread* (Primjer 5.1.3). Ta klasa podrazumijevano ima samo konstruktor i metodu *Execute*. Konstruktor određuje je li dretva nakon kreiranja suspendirana ili se počinje odmah izvršavati (*bool CreateSuspended*), a moguće ga je proširiti dodatnim parametrima te tako u dretvu prenijeti ulazne podatke (Primjer 5.1.4).

#### **Primjer 5.1.4. Konstruktor klase PrimeSearchThread**

```
__fastcall PrimeSearchThread::PrimeSearchThread(std::vector<int>& niz,
                                                int start, int kraj,
                                                bool CreateSuspended)
: niz(niz), start(start), kraj(kraj), TThread(CreateSuspended){}
```

Polazna ideja je da svaka dretva provjerava određeni dio vektora. Zbog toga konstruktor dretve osim parametra *CreateSuspended* sadrži referencu vektora iz glavne dretve (da se izbjegne kopiranje vektora) te područje pretrage (od kojeg do kojeg elementa).

**Primjer 5.1.5. Metoda Execute**

```

void __fastcall PrimeSearchThread::Execute() {
    // traženje prim brojeva u zadanom dijelu vektora...
    primKol = 0;
    for (int i = start; i <= kraj; i++)
        if (IsPrim(niz[i]))
            primKol++;
}

```

Metoda *Execute* (Primjer 5.1.5) sadrži programski kôd koji se izvršava prilikom pokretanja dretve. Upravo u ovoj metodi u zadanom dijelu vektora (od *start* do *kraj*) vršimo pretragu za prim brojevima te njihovu količinu pohranjujemo u podatkovni član *primKol*.

**Primjer 5.1.6. Kreiranje i pokretanje dretvi**

```

#include <chrono>
#include <System.Math.hpp>
#include "PrimeSearchThread.h"
// ...
using namespace std::chrono;
void __fastcall TForm1::MultiThreadedClick(TObject *Sender) {
    int brojDretvi = 4; // broj kreiranih dretvi
    std::vector<PrimeSearchThread*> t(brojDretvi); // pokazivači na dretve...
    int start, dodijeljeniElementi = 0, brElemenata = 0;
    if (brojevi.size() < brojDretvi)
        brojDretvi = brojevi.size();
    auto start_t = high_resolution_clock::now();
    for (int i = 0; i < brojDretvi; i++) {
        // izračunaj broj elemenata vektora po dretvi
        dodijeljeniElementi += brElemenata;
        start = dodijeljeniElementi;
        brElemenata = brojevi.size() / brojDretvi + (i < brojevi.size() %
        brojDretvi);
        int kraj = start + brElemenata - 1;
        // svaka dretva pretražuje svoj dio vektora: [start, kraj]
        t[i] = new PrimeSearchThread(brojevi, start, kraj, false);
    }
    // čekaј kraj rada svih dretvi i sumiraj pronađene prim brojeve
    int ukPrim = 0;
    for (int i = 0; i < brojDretvi; i++) {
        t[i]->WaitFor();
    }
}

```

```

        ukPrim += t[i]->primKol;
        delete t[i];
    }
    auto end_t = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end_t - start_t).count();
    ShowMessage("Prim brojeva: " + IntToStr(ukPrim) + ", dretve: " +
        IntToStr(brojDretvi) + ", vrijeme: " + IntToStr(duration) + " ms.");
}

```

Kao i sve komponente VCL, tako i svaka dretva mora biti kreirana korištenjem operatora *new*. Primjer 5.1.6 demonstrira kako se za pretragu prim brojeva koriste 4 dretve, prilikom čega svaka od njih pretražuje svoj dio vektora. Metodom *WaitFor* čeka se završetak rada pojedine dretve, nakon čega se iz podatkovnog člana *primKol* čita te zbraja količina pronađenih prim brojeva.

Broj dretvi	Trajanje (ms)
1	8009
2	3996
4	1974
8	1040
16	880
32	491

Tablica 5.1.1. Rezultati testiranja

Kako broj dretvi utječe na performanse pretraživanja prikazuje Tablica 5.1.1. Za pretraživanje prim brojeva u vektoru od 1 milijun elemenata jednoj (glavnoj) dretvi bilo je potrebno malo više oko 8 sekundi (8009 ms), dok su 32 dretve taj isti posao odradile za manje od pola sekunde (491 ms). Prilikom testiranja korišten je procesor sa 16 jezgri i podrškom za 32 dretve.

## 5.2. Sinkronizirani pristup VCL komponentama

Svaka dretva ima pristup memorijskom prostoru svog procesa. U takvim situacijama moguća je pojava konkurentnosti (dvije ili više dretvi u isto vrijeme pokušava obrađivati iste

resurse). Krajnji rezultat takve obrade je nepredvidiv ukoliko se dretve ne sinkroniziraju prilikom pristupa takvim resursima. Dretve se mogu sinkronizirati metodama *Synchronize* i *Queue* klase *TThread*, korištenjem objekata kritične sekcije, te *Mutex* i *Semaphore* objektima.

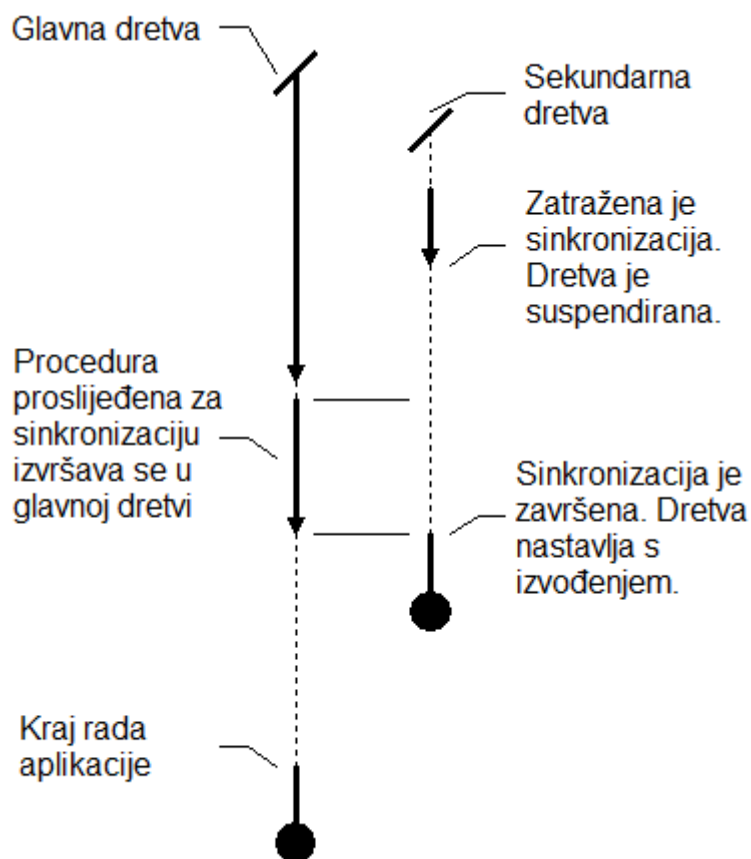
Pretpostavimo da metodu *Execute* (Primjer 5.1.5) želimo preurediti tako da svaka dretva nakon svog završetka u komponentu *Form1->ListBox1* doda poruku o broju pronađenih prim brojeva te vremenskom trajanju izvođenja. Kada se iz dretve želi modificirati grafičko sučelje aplikacije (VCL komponente), takve je prilagodbe potrebno izvršiti pozivom metode *Synchronize* ili *Queue*. U protivnom, mogući su nepredvidivi rezultati jer se izravna modifikacija VCL komponenti unutar dretve ne smatra sigurnom (nije "thread-safe").

#### Primjer 5.2.1. Sinkronizacija dretve pozivom metode *Synchronize*

```
void __fastcall PrimeSearchThread::Execute() {
    auto start_t = high_resolution_clock::now();
    primKol = 0;
    for (int i = start; i <= kraj; i++)
        if (IsPrim(niz[i]))
            primKol++;
    auto end_t = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end_t - start_t).count();
    // sinkronizacija s glavnom dretvom pomoću anonimne lambda funkcije
    Synchronize([&]() {
        Form1->ListBox1->Items->Add("Dretva gotova! "
                                     "Prim brojeva: " + IntToStr(primKol) +
                                     ", Vrijeme (ms): " + duration + ".");
    });
}
```

Primjer 5.2.1 demonstrira kako je za sinkronizirani pristup VCL komponentama moguće koristiti metodu *Synchronize*. Ona kao argument prima proceduru (njeno ime ili lambda implementaciju) koja sadrži sve željene promjene nad VCL komponentama.

Kada se upotrebom metode *Synchronize* iz dretve zatraži sinkronizacija, glavnoj se dretvi signalizira da izvrši predanu proceduru. Na taj način glavna dretva tražene promjene (predane procedure) izvršava jednu po jednu, pri čemu se ne može dogoditi konkurentni pristup VCL komponentama.



Slika 5.2.1. Sinkronizacija dretve metodom *Synchronize* [16]

Metoda *Synchronize* suspendirat će dretvu u kojoj je pozvana sve dok se predana procedura ne izvrši u glavnoj dretvi (Slika 5.2.1). Tek nakon završetka sinkronizacije ta dretva će moći dalje nastaviti sa svojim radom. Alternativno, umjesto metode *Synchronize* moguće je koristiti metodu *Queue*. Nakon signaliziranja glavnoj dretvi, ova metoda neće suspendirati rad dretve u kojoj je pozvana čekajući da se obavi sinkronizacija, već će joj odmah dopustiti daljnji rad.

### 5.3. Kritična sekcija

Dok se za sinkronizirani pristup VCL komponentama koriste metode *Synchronize* i *Queue*, za sinkronizirani pristup ostalim tipovima dijeljenih resursa moguće je koristiti objekte kritične sekcije. Kritična sekcija predstavlja dio programskog kôda dretve u kojemu se pristupa dijeljenim resursima. Kôd omeđen kritičnom sekcijom u isto vrijeme dostupan je samo jednoj dretvi, a ostale dretve su na čekanju dok ju prethodna ne oslobodi.



Pretpostavimo da u klasi *TForm1* postoji podatkovni član *int suma*. Ideja je da ispitamo konkurentni (paralelni) pristup tom podatkovnom članu iz 10 dretvi, pri čemu će svaka dretva 100 puta uvećavati varijablu *suma* za vrijednost 1 (Primjer 5.3.1).

#### **Primjer 5.3.1. Konkurentni pristup zajedničkom resursu**

```
// posao pojedine dretve...
void __fastcall SumThread::Execute() {
    for(int i = 0; i < 100; i++) {
        Form1->suma++;
        Sleep(1); // čekaj 1 ms
    }
}

// kreiranje i izvršavanje dretvi...
void __fastcall TForm1::Button1Click(TObject *Sender) {
    suma = 0;
    const int brojDretvi = 10;
    SumThread* dretva[brojDretvi];
    for(int i = 0; i < brojDretvi; i++)
        dretva[i] = new SumThread(false);
    for(int i = 0; i < brojDretvi; i++) {
        dretva[i]->WaitFor();
        delete dretva[i];
    }
    ShowMessage(suma);
}
```

Kreiranjem i izvršavanjem 10 dretvi te ispisom rezultata očekivali bismo da varijabla *suma* ima vrijednost 1000 (10 dretvi \* 100 uvećavanja za vrijednost 1). Međutim, ta vrijednost će gotovo uvijek biti manja od 1000. Naime, naredba `Form1->suma++` može uzrokovati situaciju podatkovne utrke (eng. *data race*) jer se dijeli na nekoliko atomarnih operacija:

- 1) Dohvati trenutnu vrijednost varijable *suma*
- 2) Uvećaj vrijednost varijable *suma* za 1
- 3) Pohrani novu vrijednost

Za primjer pretpostavimo da trenutna vrijednost varijabla *suma* iznosi 1. Ukoliko dvije dretve paralelno uvećaju vrijednost te varijable može se dogoditi da zbog prve atomarne

operacije obje dretve pročitaju njenu istu početnu vrijednost (1). Zbog toga će obje dretve u konačnici postaviti varijablu *suma* na vrijednost 2. Upravo zato se izvršavanjem prethodnog primjera gotovo uvijek dobije vrijednost manja od 1000.

Jedno od rješenja ovog problema je korištenje kritične sekcije koja će spriječiti konkurentan pristup varijabli *suma*. U klasi *TForm1* potrebno je dodati još jedan podatkovni član:

```
CRITICAL_SECTION kriticnaSekcija;
```

Ovom deklaracijom kreiramo objekt kritične sekcije. Njega je prije početka korištenja potrebno inicijalizirati, a na kraju i izbrisati korištenjem metoda *InitializeCriticalSection* i *DeleteCriticalSection* (Primjer 5.3.2).

#### **Primjer 5.3.2. Inicijalizacija i brisanje objekta kritične sekcije**

```
// inicijalizacija objekta kritične sekcije
void __fastcall TForm1::FormCreate(TObject *Sender){
    InitializeCriticalSection(&kriticnaSekcija);
}

// brisanje objekta kritične sekcije
void __fastcall TForm1::FormDestroy(TObject *Sender){
    DeleteCriticalSection(&kriticnaSekcija);
}
```

Konačno, u metodi *Execute* potrebno je korištenjem funkcija *EnterCriticalSection* i *LeaveCriticalSection* odrediti koji dio programskog kôda će se izvršavati u kritičnoj sekciji.

#### **Primjer 5.3.3. Pristup resursu unutar kritične sekcije**

```
void __fastcall SumThread::Execute(){
    for(int i = 0; i < 100; i++){
        EnterCriticalSection(&Form1->kriticnaSekcija);
        Form1->suma++;
        LeaveCriticalSection(&Form1->kriticnaSekcija);
        Sleep(1);
    }
}
```

Korištenjem programskog kôda iz prethodnog primjera varijablu *suma* moći će uvećavati samo jedna po jedna dretva. Zbog kontroliranog pristupa zajedničkom resursu više nije moguće da se dogodi situacija podatkovne utrke, pa će konačna vrijednost varijable *suma* uvijek iznositi 1000.

## 5.4. Mutex

Koncept mutex (*Mutual Exclusion*) vrlo je sličan kritičnoj sekciji te također služi za kontrolirani (serijski) pristup zajedničkim resursima dretve. Za razliku od kritične sekcije čiji je objekt vidljiv samo unutar jednog procesa, mutex je kernel objekt koji je vidljiv unutar cijelog sustava. Zbog toga je kritična sekcija ograničena samo na sinkronizaciju dretvi unutar jednog procesa, dok se mutex objekt može koristiti i za sinkronizaciju između različitih procesa.

Mutex objekt istovremeno može biti u vlasništvu samo jedne dretve. Kada ga nitko ne posjeduje on je u stanju signaliziranja, te čim jedna od dretvi postane njegov vlasnik signaliziranje prestaje. Za demonstraciju, prethodni problem (Primjer 5.3.1) riješen kritičnom sekcijom sada možemo riješiti upotrebom mutex objekta.

Za kreiranje i rad s mutex objektom moguće je koristiti dva pristupa:

- klasu *TMutex* (zaglavlje *SyncObjs.hpp*)
- Win32 API *HANDLE* token (predstavlja kernel resurs)

U klasi *TForm1* potrebno je kreirati mutex objekt. Ovisno o odabranom pristupu to je moguće realizirati na sljedeće načine:

```
TMutex* mutex;    // VCL  
// HANDLE mutex;  // Win32 API
```

Ukoliko se koristi klasa *TMutex* prethodno je potrebno uključiti zaglavlje *SyncObjs.hpp*. Zatim, mutex objekt potrebno je kreirati, a na kraju i uništiti kada više nije potreban (Primjer 5.4.1).

**Primjer 5.4.1. Kreiranje i uništavanje mutex objekta**

```
// Kreiranje mutex objekta
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner){
    // VCL
    mutex = new TMutex((_SECURITY_ATTRIBUTES*)NULL, false, "Mutex98", false);
    // ili Win32 API
    // mutex = CreateMutex(NULL, false, L"Mutex98");
}
// Uništavanje mutex objekta
void __fastcall TForm1::FormDestroy(TObject *Sender){
    delete mutex; // VCL
    // CloseHandle(mutex); // Win32 API
}
```

Mutex objekt može imati svoje ime, ali to nije nužno. Ukoliko postoji, ime bi trebalo biti jedinstveno kako bi se moglo razlikovati od imena ostalih kernel objekata.

**Primjer 5.4.2. Sinkronizacija dretve upotrebom mutex objekta**

```
void __fastcall SumThread::Execute(){
    for(int i = 0; i < 100; i++){
        Form1->mutex->Acquire(); // VCL
        // WaitForSingleObject(Form1->mutex, INFINITE); // Win32 API
        Form1->suma++;
        Form1->mutex->Release(); // VCL
        // ReleaseMutex(Form1->mutex); // Win32 API
        Sleep(1); // uspori rad dretve 1 ms
    }
}
```

Ukoliko je korišten objekt klase *TMutex* metodom *Acquire* čeka se da mutex objekt počne signalizirati, nakon čega dretva nad njime preuzima vlasništvo. Alternativno, Win32 funkcija *WaitForSingleObject* radi identičnu stvar kada se koristi HANDLE token (Primjer 5.4.2).

Tek nakon što je dretva preuzela vlasništvo nad mutex objektom, ona može nastaviti sa svojim izvršavanjem. Ostale dretve su u stanju čekanja, te je tako spriječen konkurentni pristup dijeljenim resursima. Konačno, dretva se odriče vlasništva pozivom metode *Release* ili funkcije *ReleaseMutex*.

**Primjer 5.4.3. Sinkronizacija dretvi različitih procesa**

```
HANDLE hMutex;  
  
hMutex = CreateMutex(NULL, FALSE, L"JedinstvenoIme");  
  
if(hMutex == NULL)  
    ShowMessage(GetLastError());  
  
else  
    // je li ovaj mutex objekt već u nečijem vlasništvu?  
    if(GetLastError() == ERROR_ALREADY_EXISTS) {  
        ShowMessage("Aplikacija je već pokrenuta!");  
        return -1;  
    }  
}
```

Budući da mutex objekt postoji globalno u cijelom sustavu, Primjer 5.4.3 prikazuje kako ga upotrijebiti za sinkronizaciju dretvi različitih procesa. Programski kôd iz primjera potrebno je dodati na početku glavne funkcije (WINAPI \_tWinMain), te zatim pokušati pokrenuti dvije instance aplikacije.

Pokretanjem prve instance aplikacije stvara se mutex objekt koji je u vlasništvu glavne dretve te instance. Prilikom pokretanja druge instance javit će se greška da je aplikacija već pokrenuta jer u drugoj instanci aplikacije neće biti moguće stvoriti mutex objekt istog imena (*JedinstvenoIme*). Tek kada se prva instanca aplikacije zatvori otpustit će se vlasništvo nad tim mutex objektom te će se moći pokrenuti nova instanca.

Kompleksnija inačica mutexa je semafor. Dok mutex dopušta pristup samo jednoj dretvi semafor istovremeno može dopustiti pristup više dretvi. Također, mutex može biti otpušten samo od strane dretve koja nad njime ima vlasništvo, dok semafor može biti otpušten od strane bilo koje dretve. [17]

## 5.5. Biblioteka paralelnog programiranja

Počevši od izdanja XE7, Delphi i C++ Builder omogućuju korištenje nove biblioteke paralelnog programiranja (eng. *Parallel Programming Library*). Ona je dio RTL-a (eng. *Run-Time Library*) i njena definicija se nalazi u System.Threading.hpp zaglavlju. Omogućuje jednostavnije pisanje višedretvenih aplikacija koje koriste više procesorskih

jezgri, te ju je moguće koristiti u VCL i FireMonkey aplikacijama na Windows, Mac, iOS i Android platformama.

Paralelna biblioteka omogućuje korištenje;

- Paralelnih petlji *for*
- Zadataka (eng. *tasks*)
- Budućnosti (eng. *futures*)

Za demonstraciju pretpostavimo da trebamo pretražiti koliko prim brojeva postoji u vektoru *brojevi* od 1 milijun elemenata (Primjer 5.1.1). Primjer 5.1.2 već demonstrira rješenje koje koristi samo jednu (glavnu) dretvu, te traženu pretragu obavlja za 8009 ms. Sada ovaj isti problem možemo riješiti pomoću biblioteke paralelnog programiranja korištenjem paralelne petlje *for* (Primjer 5.5.1).

#### **Primjer 5.5.1. Pretraga prim brojeva (*Parallel::For*)**

```
#include <chrono>
#include <System.Threading.hpp>
// ...

void __fastcall TForm1::ParallelForClick(TObject *Sender){
    auto start_t = std::chrono::high_resolution_clock::now();
    int kol = 0;
    TParallel::For(0, brojevi.size(), _di_TProc__1<int>([&kol, this](int i){
        if(IsPrim(brojevi[i]))
            TInterlocked::Increment(kol);
    }));
    auto end_t = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>
        (end_t - start_t).count();
    ShowMessage("Kolicina: " + String(kol) + " vrijeme: " +
        IntToStr(duration) + " ms.");
}
```

Korištenjem procesora sa 16 jezgri (32 dretve) do rezultata se dođe tek za ~500 ms, što predstavlja ~1600% bolje performanse u odnosu na jednodretveno rješenje. Performanse su gotovo identične kao i u slučaju implementacije dretvi pomoću *TThread* klase. Ipak, paralelna biblioteka nudi mnogo jednostavniju i bržu implementaciju.

Paralelna petlja *for* ima mnogo oblika. Inicijalno radi na način da korištenjem podrazumijevanog bazena dretvi (eng. *default thread pool*) prati opterećenje procesora, broj zauzetih dretvi i dostupnu količinu ostalih sistematskih resursa (npr. memorije), te na osnovu toga automatski kreira maksimalan broj iskoristivih dretvi u kojima će izvršavati tražene operacije. Čim jedna dretva završi sa svojim radom automatski se kreira i počinje izvršavati sljedeća. Po potrebi je moguće kreirati i koristiti vlastite bazene dretvi gdje je ručno moguće odrediti broj korištenih dretvi, ali to ipak nije preporučljivo jer će takav pristup najčešće rezultirati slabijim performansama [18].

#### Primjer 5.5.2. Pretraga prim brojeva (*\_di\_ITask*)

```
#include <chrono>
#include <System.Threading.hpp>
// ...

void __fastcall TForm1::ParallelTaskClick(TObject *Sender){
    int count = numbers.size();
    _di_ITask* task = new _di_ITask[count];
    ProgressBar1->Position = 0;
    ProgressBar1->Max = count;
    int kol = 0;
    for(int i = 0; i < count; i++){
        // kreiraj zadatak
        task[i] = TTask::Create([i, &kol, this]() {
            if(IsPrim(i))
                TInterlocked::Increment(kol);
            // UI sinkronizacija
            TThread::Queue(NULL, [&]() {
                ProgressBar1->Position++;
            });
        });
        // izvrši zadatak
        task[i]->Start();
    }
    // čekaj završetak svih zadataka i prikazuju napredak svakih 500 ms
    while(!TTask::WaitForAll(task, count - 1, 500))
        Application->ProcessMessages();
}
```

Isti problem moguće je riješiti i korištenjem zadataka iz biblioteke paralelnog programiranja (Primjer 5.5.2). Zadatak (eng. *task*) predstavlja asinkronu jedinicu rada koju treba izvršiti.

Za izvršavanje pojedinog zadatka neće se nužno uvijek kreirati zasebna dretva, pa je za zadatke najčešće potrebno koristiti manje sistemskih resursa nego kada se isti rad izvršava korištenjem paralelne petlje *for*.

Kada će se zadaci izvršiti ovisi o bazenu dretvi, tj. trenutnom opterećenju procesora, a njihov završetak moguće je pričekati pozivom metoda *WaitForAny* i *WaitForAll*. Zadatak se kreira pozivom metode *Create* te naknadno izvršava pozivom metode *Start*. Alternativno, zadatak je moguće kreirati i odmah pokrenuti pozivom metode *Run*. Moguće je koristiti i metode *Wait* i *Cancel*, te dohvatiti status izvršavanja zadatka korištenjem svojstva *Status*.

Bilo da je riječ o korištenju paralelne petlje *for* ili zadataka, za njihovu sinkronizaciju moguće je koristiti metode poput *TInterlocked::Increment*, *TInterlocked::Decrement*, *TInterlocked::Add* itd. Također, unutar njih moguće je koristiti kritične sekcije, mutex i semafore, dok se za sinkronizaciju grafičkog sučelja aplikacije (npr. unutar glavne dretve) koriste metode *TThread::Synchronize* i *TThread::Queue* (Primjer 5.5.2).

Zadatak (eng. *task*) možemo poistovjetiti s procedurom, tj. radom koji nema nikakvu povratnu vrijednost. Alternativno, možemo koristiti i budućnost (eng. *future*), a to je zadatak s povratnom vrijednošću.

#### **Primjer 5.5.3. Korištenje budućnosti (eng. *future*)**

*// deklaracija u širem dosegu (unutar klase ili u globalnom prostoru)*

```
_di_IFuture__1<String> FutureObj;
```

```
void __fastcall TForm1::GCreateFutureClick(TObject *Sender){
```

```
    FutureObj = TTask::Future<String>([]()-> String {
```

```
        Sleep(5000);
```

```
        return "Gotovo!";
```

```
    });
```

```
}
```

```
void __fastcall TForm1::GetFutureValueClick(TObject *Sender){
```

```
    ShowMessage(FutureObj->Value);
```

```
}
```



U C++ Builderu moguće je koristiti Delphi sučelje `_di_IFuture` kako bismo kreirali objekt koji predstavlja *budućnost*. Prilikom njegove deklaracije navodi se povratni tip, dok je sam objekt potrebno smjestiti u područje šireg dosega (npr. unutar klase) kako bi mu se moglo pristupiti iz svih potrebnih dijelova programa. Konačno, objekt tipa *budućnost* potrebno je inicijalizirati (metoda `GCreateFutureClick`, Primjer 5.5.3), prilikom čega odmah započinje njegovo izvršavanje. Tokom izvršavanja *budućnosti* aplikacija u pravilu neće biti zamrznuta. To će se dogoditi samo ukoliko pokušamo dohvatiti rezultat izvršavanja *budućnosti* prije nego li se programski kod unutar nje izvršio. Aplikacija će se tada zamrznuti, tj. čekati sve dok ta *budućnost* ne završi sa svojim radom i ne vrati rezultat.

## 5.6. Procesi

Proces je instanca aplikacije koja se izvršava na računalu. Jedan proces može instancirati i druge procese, pri čemu nastavak rada prvog procesa može biti uvjetovan uspješnim završetkom drugog procesa.

Primjerice, pretpostavimo da proces *setup.exe* zahtjeva da se tijekom instalacije aplikacije uspješno instalira i Microsoft .NET Framework 4.5. Proces *setup.exe* u nekom će trenutku instancirati proces *dotNetFx45\_Full\_setup.exe* te će čekati njegov završetak. Ukoliko je instalacija .NET Framework-a uspješno završila, proces *setup.exe* može dalje nastaviti sa svojim radom. Također, svoj rad može i odlučiti prekinuti ukoliko se instalacija .NET Frameworka iz nekog razloga nije uspješno završila.

Da bi jedan proces znao kako je završio rad drugog procesa koristi se povratna vrijednost glavne funkcije (`main`, `_tWinMain`). Ukoliko je aplikacija uspješno završila s radom, standardno se koristi 0 kao povratna vrijednost, a u slučaju greške neka druga cjelobrojna vrijednost. Osim preko glavne funkcije, povratna vrijednost može se odrediti i pozivom funkcije *exit* koja kao argument prima cjelobrojnu vrijednost.

### Primjer 5.6.1. Aplikacija komandne linije

```
#include <iostream>

int main() {
    int n;
    std::cout << "Unesi broj: ";
```

```
std::cin >> n;
return n;
}
```

Primjerom jednostavne aplikacije komandne linije (Primjer 5.6.1) možemo prikazati kako se povratna vrijednost njene glavne funkcije može dohvatiti iz nekog drugog procesa.

#### **Primjer 5.6.2. Instanciranje procesa i dohvaćanje povratne vrijednosti**

```
void __fastcall TForm1::Button1Click(TObject *Sender){
    STARTUPINFO startInfo;
    PROCESS_INFORMATION processInfo;
    char CommandLine[255] = "consoleapp.exe";

    GetStartupInfo(&startInfo);
    // kreiraj instancu aplikacije komandne linije (consoleapp.exe)
    if (!CreateProcess(NULL, UnicodeString(CommandLine).w_str(),
                      NULL, NULL, FALSE, 0, NULL, NULL,
                      &startInfo, &processInfo)) {
        ShowMessage("Proces nije moguće kreirati!");
        return;
    }
    // čekaj kraj rada procesa
    WaitForSingleObject(processInfo.hProcess, INFINITE);
    unsigned long retVal;
    GetExitCodeProcess(processInfo.hProcess, &retVal);
    // prikaz povratne vrijednosti
    ShowMessage((int)retVal);
}
```

Proces se može instancirati pozivom funkcije *CreateProcess* (Primjer 5.6.2). Završetak rada tog procesa čekamo pozivom funkcije *WaitForSingleObject*, nakon čega povratnu vrijednost dohvaćamo funkcijom *GetExitCodeProcess*. Tijekom čekanja, glavna dretva aplikacije bit će zauzeta, a samu povratnu vrijednost na kraju je potrebno pretvoriti (eng. *cast*) u tip *int* kako bi se mogle dohvatiti i negativne povratne vrijednosti.

Osim funkcije *CreateProcess*, moguće je koristiti i funkcije *WinExec* i *ShellExecute*. Tako se naredbom

```
WinExec("Notepad.exe", SW_NORMAL);
```

pokreće Notepad aplikacija. Drugim parametar određuje se način prikaza prozora (SW\_NORMAL, SW\_MINIMIZE, SW\_MAXIMIZE, SW\_HIDE itd.).

Funkcija *ShellExecute* korisna je prilikom rada s datotekama. Naime, ona će po ekstenziji datoteke sama provjeriti i pokrenuti aplikaciju koja obrađuje sadržaj tog tipa datoteka. Izvršavanjem naredbe

```
ShellExecute(0, NULL, L"dokument.docx", NULL, NULL, SW_SHOW);
```

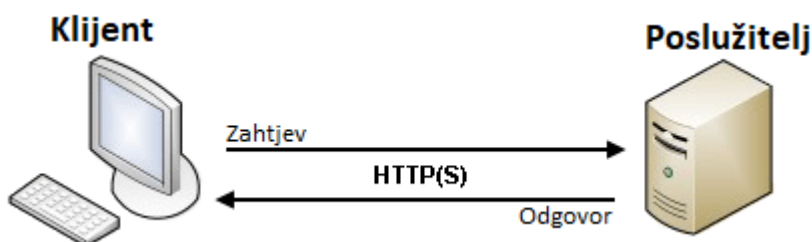
na računalu će se pokrenuti aplikacija registrirana za obradu dokumenata s DOCX ekstenzijom. Detaljnu dokumentaciju za korištenje funkcija *WinExec* i *ShellExecute* moguće je pronaći na stranicama MSDN-a.



## 6. Mrežne aplikacije

### 6.1. Komunikacija klijent-poslužitelj

Komunikacija među računalima u mreži najčešće se odvija korištenjem modela klijent-poslužitelj (Slika 6.1.1). Tipično, klijent šalje zahtjev prema poslužitelju (eng. *server*), koji nakon zaprimanja i obrade tog zahtjeva klijentu vraća odgovor. Sljedeća komunikacija opet započinje na isti način, odnosno slanjem klijentskog zahtjeva i završava odgovorom poslužitelja. Ovakav pristup ujedno predstavlja sinkroni oblik komunikacije klijent-poslužitelj. [19]



Slika 6.1.1. Model klijent-poslužitelj

Komunikacija klijent-poslužitelj može biti i asinkrona. Primjerice, iako klijent može poslati e-poruku u bilo kojem trenutku nikada ne zna kada (i ako) će na nju dobiti odgovor. Umjesto da klijent čeka odgovor na e-poruku koju je upravo poslao, on u asinkronom obliku komunikacije može raditi i neke druge stvari dok ne dobije traženi odgovor (pročitati već dospjele e-poruke, poslati nove itd.).

Pojmovi sinkrone i asinkrone komunikacije povezani su i s korištenjem blokirajućih i neblokirajućih utičnica (eng. *socket*). Utičnica predstavlja krajnju točku komunikacije opisanu mrežnim protokolom, IP adresom i mrežnim priključkom (eng. *port*). Mrežnim protokolom definira se način spajanja i razmjene podataka. IP adresa služi za identifikaciju računala u mreži, dok se mrežnim priključkom identificira aplikacija (servis) koja ima ulogu poslužitelja.

Blokirajuće utičnice koriste se u sinkronom obliku komunikacije klijent-poslužitelj gdje se nakon slanja zahtjeva klijent stavlja u stanje čekanja (blokiranja) sve dok od poslužitelja ne

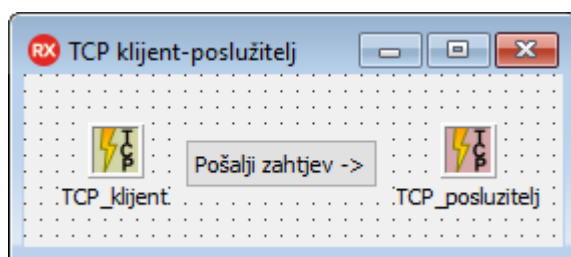
dobije odgovor. Zbog neizvjesnog vremena čekanja to može predstavljati problem u višeklijentskom okruženju, pa se u kombinaciji s blokirajućim utičnicama često koristi višedretvenost (eng. *multithreading*). Zahtjev svakog klijenta tada se obrađuje u zasebnoj dretvi kako jedan klijent ne bi morao čekati na obradu zahtjeva klijenata prije njega.

Za razliku od prethodnih, neblokirajuće utičnice neće staviti klijenta u stanje čekanja (blokiranja). Nakon slanja zahtjeva kontrola je automatski vraćena klijentu koji će dodatnim pozadinskim radnjama provjeravati dospijeće odgovora, dok će u međuvremenu biti u mogućnosti obavljati i neke druge radnje. Ipak, zbog puno jednostavnije programske implementacije danas se najčešće koriste blokirajuće utičnice.

Za razvoj mrežnih klijent-poslužitelj aplikacija u C++ Builderu primarno se koriste Indy (*Internet Direct*) komponente. Riječ je o rješenju otvorenog koda (eng. *open source*) koje je moguće koristiti u bibliotekama VCL i FireMonkey te pri razvoju .NET aplikacija. Indy koristi blokirajuće utičnice, pri čemu automatski implementira višedretvenost za sve dolazne zahtjeve klijenata prema poslužiteljima. Nudi široku paletu komponenti klijent-poslužitelj, pomoću kojih podržava protokole poput TCP/IP, UDP, HTTP, ICMP, FTP, SMTP, POP3 itd. Indy podržava i implementaciju zaštićene (kriptirane) mrežne komunikacije korištenjem biblioteke OpenSSL.

## 6.2. Indy TCP/IP

TCP (*Transmission Control Protocol*) jedan je od osnovnih mrežnih protokola. Riječ je o spojnem protokolu koji za razmjenu podataka zahtjeva prethodnu uspostavu veze. TCP karakterizira pouzdan prijenos podataka, pri čemu se kontrolira i sam redoslijed podataka. Koristi se i u nekim drugim protokolima poput HTTP, SMTP, FTP itd.



Slika 6.2.1. Indy TCP klijent i poslužitelj

Pretpostavimo da želimo implementirati TCP komunikaciju klijent-poslužitelj gdje klijent kao zahtjev šalje dva cijela broja, a od poslužitelja kao odgovor očekuje njihovu sumu. Obično su klijent i poslužitelj zasebne aplikacije smještene na različitim računalima, no u svrhu demonstracije oboje ćemo sada smjestiti u samo jednu aplikaciju. Tako Slika 6.2.1 prikazuje komponente *TIdTCPClient* (*TCP\_klijent*) i *TIdTCPServer* (*TCP\_posluzitelj*) čija svojstva je moguće odrediti na sljedeći način (Primjer 6.2.1):

#### *Primjer 6.2.1. Slanje zahtjeva korištenjem Indy TCP klijenta*

```
void __fastcall TForm1::Button1Click(TObject *Sender) {
    TCP_klijent->Host = "127.0.0.1"; // localhost
    TCP_klijent->Port = 5887; // mrežni priključak poslužitelja

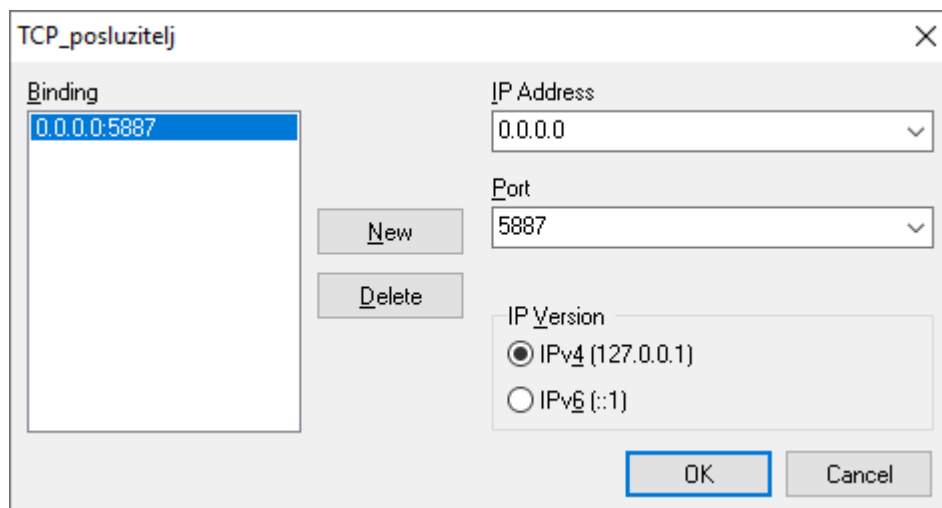
    TCP_klijent->Connect();
    TCP_klijent->Socket->Write(17); // pošalji prvi broj
    TCP_klijent->Socket->Write(35); // pošalji drugi broj
    // čekaj i ispiši odgovor poslužitelja
    ShowMessage(TCP_klijent->Socket->ReadLongInt());
    TCP_klijent->Disconnect();
}
```

Lokaciju poslužitelja (svojstvo *Host*) i mrežni priključak na kojem sluša zahtjeve moguće je odrediti pomoću programskog kôda ili objektnog inspektora. Zatim, TCP klijent pozivom metode *Connect* uspostavlja vezu s poslužiteljem, nakon čega mu metodom *Write* šalje dva cijela broja. Metoda *Write* ima nekoliko preopterećenja koja osim slanja cijelih brojeva omogućuju slanje teksta (*UnicodeString*), bitovnih blokova itd. Osim metode *Write*, Indy TCP klijent omogućuje i korištenje metoda *WriteLn*, *WriteFile* itd.

Konačno, TCP klijent čeka odgovor poslužitelja (sumu prethodno poslanih cijelih brojeva) pozivom metode *ReadLongInt*. U ovom trenutku klijent je blokiran sve dok mu ne stigne odgovor poslužitelja ili dok ne istekne vrijeme određeno svojstvom *ReadTimeout*. Kada klijent dobije odgovor odspaja se od poslužitelja pozivom metode *Disconnect*.

Za razliku od klijenta, poslužitelj Indy TCP ima dva ključna svojstva; *Bindings* i *Active*. Svojstvom *Bindings* određuje se područje (IP adresa i mrežni priključak) koje poslužitelj

osluškuje, pri čemu se za IP adresu najčešće koristi 0.0.0.0 ("prihvati zahtjev sa bilo koje IP adrese").



Slika 6.2.2. Svojstvo Bindings

Jedan TCP poslužitelj u isto vrijeme može osluškivati više IP adresa i mrežnih priključaka, pri čemu je moguće kombinirati adrese IPv4 i IPv6. Slika 6.2.2 prikazuje kako *TCP\_poslužitelj* namjerava zaprimati zahtjeve s bilo koje IP adrese na mrežnom priključku 5887. Da bi *TCP\_poslužitelj* počeo osluškivati navedeno područje prethodno je potrebno svojstvo *Active* postaviti na vrijednost *true*.

#### Primjer 6.2.2. Implementacija Indy TCP poslužitelja

```
void __fastcall TForm1::TCP_poslužiteljExecute(TIdContext *AContext){
    // pročitaj dva cijela broja
    int a = AContext->Connection->Socket->ReadLongInt();
    int b = AContext->Connection->Socket->ReadLongInt();
    // pošalji odgovor klijentu
    AContext->Connection->Socket->Write(a + b);
    // odspoji klijenta
    AContext->Connection->Disconnect();
}
```

Indy TCP poslužitelj će za svaki dolazni zahtjev automatski u zasebnoj dretvi izvršiti metodu zaduženu za *OnExecute* događaj (Primjer 6.2.2). Pri obradi zahtjeva važno je tek da su klijent i poslužitelj usklađeni u redoslijedu slanja i čitanja podataka. Primjerice, *TCP\_klijent* prvo je poslao dva cijela broja, a zatim od poslužitelja čeka odgovor, odnosno



njihovu sumu (*Primjer 6.2.1*). Sukladno tome, *TCP\_poslužitelj* prvo čita ta dva cijela broja, pa zatim klijentu šalje traženu sumu (*Primjer 6.2.2*).

Kako se svaki zahtjev obrađuje u zasebnoj dretvi, nije sigurno iz dretve izravno ažurirati grafičko sučelje poslužitelj aplikacije. Kada je to potrebno, sva takva ažuriranja moraju se sinkronizirati s glavnom dretvom.

### Primjer 6.2.3. Sinkronizirano ažuriranje grafičkog okruženja poslužitelja

```
void __fastcall TForm1::TCP_poslužiteljExecute(TIdContext *AContext) {
    // pročitaj dva cijela broja
    int a = AContext->Connection->Socket->ReadLongInt();
    int b = AContext->Connection->Socket->ReadLongInt();
    // pošalji klijentu odgovor
    AContext->Connection->Socket->Write(a + b);
    // sinkronizirano ažuriranje grafičkog okruženja
    TThread::Synchronize(TThread::CurrentThread,
        [&]() {
            String klijent = AContext->Connection->Socket->Binding->PeerIP;
            String zahtjev = "(" + IntToStr(a) + ", " + IntToStr(b) + ")";
            ListBox1->Items->Add(klijent + " " + zahtjev);
        });
    // odspoji klijenta
    AContext->Connection->Disconnect();
}
```

Primjer 6.2.3 demonstrira kako se u komponenti *ListBox1* aplikacije poslužitelja pohranjuje povijest svih dosadašnjih zahtjeva. Taj dio programskog kôda omeđen je anonimnom funkcijom lambda koja je kao argument predana statičkoj metodi *Synchronize*, klase *TThread*.

Bilo kakav oblik podataka moguće je preko mreže razmjenjivati korištenjem tokova (eng. *stream*). To je pogotovo korisno kada se radi s nestrukturiranim podacima. Tako Primjer 6.2.4 demonstrira kako Indy TCP klijent prenosi datoteku (njen naziv, veličinu i sadržaj) Indy TCP poslužitelju.

### Primjer 6.2.4. Slanje datoteke Indy TCP poslužitelju

```
#include <memory>
```

```
// ...  
void __fastcall TForm1::Button1Click(TObject *Sender){  
    if(!OpenDialog1->Execute()) return; //odabir datoteke  
    TCP_klijent->Host = "127.0.0.1"; // localhost  
    TCP_klijent->Port = 5887; // mrežni priključak poslužitelja  
    TCP_klijent->Connect();  
    // pošalji datoteku  
    UnicodeString datoteka = OpenDialog1->FileName;  
    std::unique_ptr<TFileStream> fs(new TFileStream(datoteka, fmOpenRead));  
    TCP_klijent->Socket->WriteLn(ExtractFileName(fs->FileName)); // naziv  
    TCP_klijent->Socket->Write(fs->Size); // veličina  
    TCP_klijent->Socket->Write(fs->get()); // tok (sadržaj)  
    TCP_klijent->Disconnect();  
}
```

Sadržaj odabrane datoteke čita se s diska te se njen naziv, veličina i sadržaj (objekt tipa *TFileStream*) šalju TCP poslužitelju. On će poslani sadržaj dohvatiti na sljedeći način (Primjer 6.2.5):

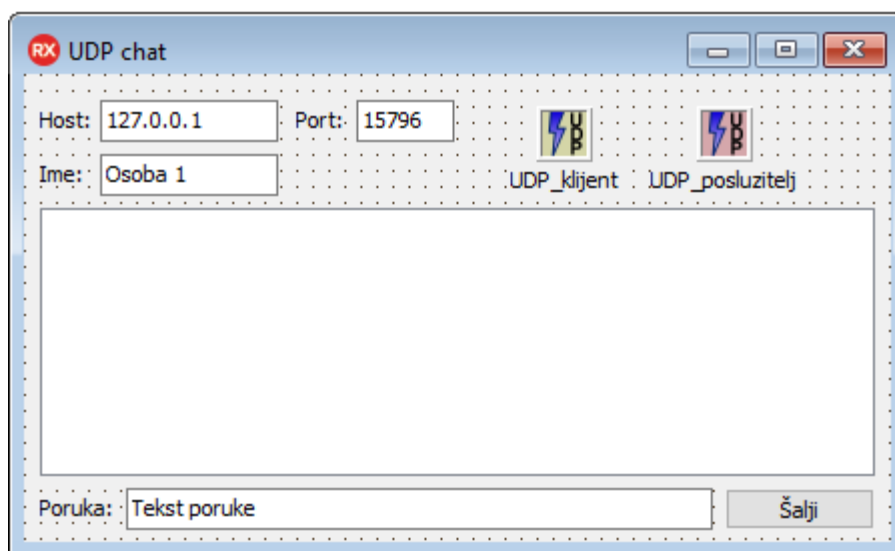
#### **Primjer 6.2.5. Preuzimanje datoteke u TCP poslužitelju**

```
#include <memory>  
// ...  
void __fastcall TForm1::TCP_posluziteljExecute(TIdContext *AContext){  
    UnicodeString naziv = AContext->Connection->Socket->ReadLn(); // naziv  
    int velicina = AContext->Connection->Socket->ReadInt64(); // veličina  
    // pohrani sadržaj datoteke  
    std::unique_ptr<TFileStream> fs(new TFileStream(naziv, fmCreate));  
    AContext->Connection->Socket->ReadStream(fs->get(), velicina);  
    AContext->Connection->Disconnect();  
}
```

Čitanje toka podataka na strani poslužitelja (metoda *ReadStream*) zahtjeva da prethodno znamo veličinu tog toka. Zbog toga je TCP klijent prije slanja sadržaja datoteke (toka) uz naziv datoteke poslao i njenu veličinu.

## 6.3. Indy UDP

Za razliku od protokola TCP/IP, UDP (*User Datagram Protocol*) nudi puno veću brzinu. Međutim, uopće ne jamči dospijeće podataka do odredišta pa čak niti njihov ispravan redoslijed. Jedino što je pouzdano jest njihov sadržaj. Također, UDP protokol ne zahtjeva uspostavu veze kao preduvjet za razmjenu podataka, pa se za njega kaže i da je bezspojni (eng. *connectionless*) protokol.



Slika 6.3.1. UDP aplikacija za razmjenu poruka

UDP protokol možemo demonstrirati jednostavnom aplikacijom za razmjenu poruka (Slika 6.3.1). U komponenti *UDP\_posluzitelj* svojstvom *Bindings* potrebno je odrediti područje na kojemu poslužitelj osluškuje zahtjeve (Slika 6.2.2), a zatim ga postaviti u aktivno stanje (*Active = true*). UDP klijent mora znati IP adresu poslužitelja (svojstvo *Host*) i broj odabranog mrežnog priključka (svojstvo *Port*).

### Primjer 6.3.1. Struktura paketa

```
class Paket{
public:
    wchar_t vrijeme[50]; // kada je poruka poslana
    wchar_t autor[25];    // autor poruke
    wchar_t poruka[255];  // sadržaj poruke
};
```

Kako UDP protokol ne jamči dospijanje svih niti primitak poslanih podataka u ispravnom redoslijedu, svi podaci koje razmjenjujemo (vrijeme, autor i sadržaj poruke) grupirani su kao podatkovni paket (Primjer 6.3.1). Na taj način sva tri dijela poruke mogu se poslati odjednom kao jedan podatak, odnosno kao binarna reprezentacija objekta tipa *Paket* (Primjer 6.3.2).

### Primjer 6.3.2. Slanje paketa korištenjem UDP klijenta

```
void __fastcall TForm1::GSaljiClick(TObject *Sender){
    // podaci o UDP poslužitelju
    UDP_klijent->Host = EHost->Text;
    UDP_klijent->Port = EPort->Text.ToInt();
    // formiranje sadržaja podatkovnog paketa
    Paket poruka;
    wcsncpy(poruka.vrijeme, Now().DateTimeString().w_str(), 50);
    wcsncpy(poruka.autor, EAutor->Text.w_str(), 25);
    wcsncpy(poruka.poruka, EPoruka->Text.w_str(), 255);
    // slanje podatkovnog paketa
    UDP_klijent->SendBuffer(RawToBytes(&poruka, sizeof(poruka)));
}
```

Za slanje tekstualnih poruka Indy UDP klijent sadrži metodu *Send*. Međutim, kako je sada riječ o slanju podatkovnog paketa, odnosno binarnog sadržaja, koristi se metoda *SendBuffer*. Ona kao argument prima objekt tipa *TIdBytes* pa je objekt tipa *Paket* prethodno potrebno pretvoriti u taj tip korištenjem funkcije *RawToBytes*.

### Primjer 6.3.3. Implementacija Indy UDP poslužitelja

```
void __fastcall TForm1::UDP_posluziteljUDPRead(TIdUDPListenerThread *AThread,
    const TIdBytes AData, TIdSocketHandle *ABinding){
    Paket poruka;
    BytesToRaw(AData, &poruka, AData.Length);
    ListBox1->Items->Add("<" + String(poruka.vrijeme) + ">" +
        String(poruka.autor) + " : " +
        String(poruka.poruka));
}
```

Komponenta poslužitelja (*UDP\_posluzitelj*) za zaprimanje podataka i zahtjeva koristi događaj *OnUDPRead*. Primljeni podaci bit će tipa *TIdBytes*, pa ih je potrebno pretvoriti nazad u tip *Paket* pozivom metode *BytesToRaw* (Primjer 6.3.3).

Potrebno je primijetiti da ažuriranje grafičkog sučelja (dodavanje poruke u komponentu *ListBox1*) nije sinkronizirano. Naime, komponenta UDP poslužitelja sadrži svojstvo *ThreadedEvent*. Podrazumijevana vrijednost ovog svojstva je *false*, odnosno svi zahtjevi će se obrađivati u glavnoj dretvi poslužitelj aplikacije. Ukoliko svojstvo *ThreadedEvent* ima vrijednost *true*, svaki se zahtjev obrađuje u zasebnoj dretvi, pa je tada potrebno sinkronizirati svako ažuriranje grafičkog sučelja poslužitelja.

UDP protokol također omogućuje klijentu istovremeno slanje tekstualne poruke ili podatkovnog paketa svim računalima u mreži. Primjerice,

```
UDP_klijent->Broadcast(RawToBytes(&poruka, sizeof(poruka)), UDP_klijent->Port);
```

Metoda *Broadcast* kao prvi argument prima sadržaj, a kao drugi argument broj mrežnog priključka. Zatim se predani sadržaj odjednom šalje svim računalima u mreži. Alternativno, umjesto poziva metode *Broadcast* poruku svim računalima u mreži moguće je poslati korištenjem *broadcast* adrese mreže.

#### ***Primjer 6.3.4. Slanje podatkovnog paketa korištenjem broadcast adrese***

```
UDP_klijent->BroadcastEnabled = true;  
UDP_klijent->SendBuffer("255.255.255.255", UDP_klijent->Port,  
                        RawToBytes(&poruka, sizeof(poruka)));
```

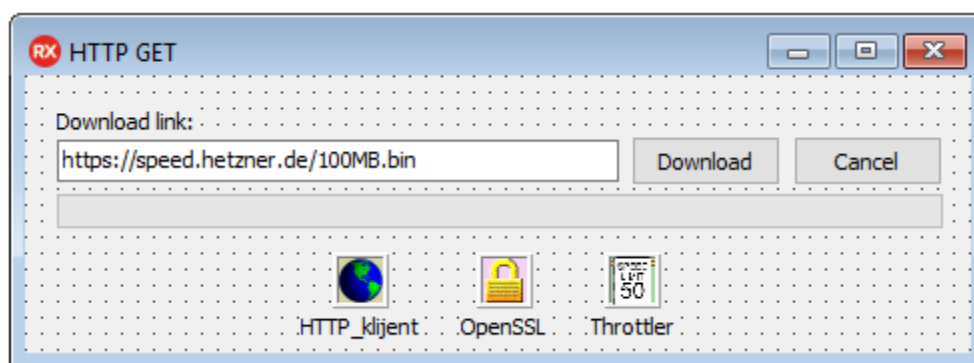
Da bi Primjer 6.3.4 uspješno radio prethodno je potrebno svojstvo *BroadcastEnabled* postaviti na vrijednost *true* (podrazumijevano je *false*). Ista akcija nije potrebna ukoliko se koristi metoda *Broadcast*.

## **6.4. Indy HTTP**

HTTP (*Hypertext Transfer Protocol*) koristi se za prijenos hipermedijskih dokumenata, kao što je HTML. Dizajniran je za komunikaciju između web preglednika i web poslužitelja, ali se može koristiti i u druge svrhe (primjerice, za komunikaciju s web servisima). HTTP slijedi klasični model klijent-poslužitelj s klijentom koji otvara vezu radi

zahtjeva, a zatim čeka dok ne primi odgovor. HTTP je protokol bez stanja, što znači da poslužitelj ne čuva nikakve podatke (stanja) između dvaju zahtjeva. [20]

Za operacije s resursima HTTP koristi metode POST (kreiraj), GET (dohvati), PUT (ažuriraj), DELETE (briši) itd. Metode mogu biti sigurne i idempotentne. Sigurna (eng. *safe*) metoda je ona koja ne mijenja stanje poslužitelja nakon svog poziva, dok će idempotentna metoda za višestruko ponavljanje istog zahtjeva uvijek vratiti isti rezultat. Primjerice, metoda GET sigurna je i idempotentna, dok metoda POST nije sigurna niti idempotentna.



Slika 6.4.1. Preuzimanje datoteke korištenjem Indy HTTP klijenta

Primjerom integriranog Indy HTTP klijenta prikazati ćemo kako VCL aplikacija može preuzeti datoteku s web poslužitelja (Slika 6.4.1). Štoviše, odabrana datoteka smještena je na web poslužitelju koji koristi zaštićeni HTTPS protokol. Zbog toga je uz *TIdHTTP* (*HTTP\_klijent*) potrebno koristiti i komponentu *TIdSSLIOHandlerSocketOpenSSL* (*OpenSSL*).

```
HTTP_klijent->IOHandler = OpenSSL; // podrška za HTTPS protokol
```

Da bi kriptirana komunikacija pomoću OpenSSL-a bila uspješna, u radnoj mapi aplikacije moraju se nalaziti OpenSSL biblioteke *ssleay32.dll* i *libeay32.dll*. Njih je u zip arhivi moguće preuzeti na sljedećoj adresi: <https://indy.fulgan.com/SSL/>.

#### Primjer 6.4.1. Preuzimanje datoteke korištenjem Indy HTTP klijenta

```
// dohvati naziv datoteke
UnicodeString ExtractUrlFileName(UnicodeString url) {
    int i = LastDelimiter('/', url);
    return url.SubString(i + 1, url.Length() - i);
}
```

```
}  
  
// započni preuzimanje datoteke  
void __fastcall TForm1::Button1Click(TObject *Sender){  
    TFileStream* fs = new TFileStream(ExtractUrlFileName(ELink->Text),  
                                     fmCreate);  
  
    HTTP_klijent->Get(ELink->Text, fs);  
    delete fs;  
}
```

Indy HTTP klijent preuzimanje sadržaja započinje pozivom metode *Get* čiji su argumenti lokacija sadržaja (URL) i odredište (adresa *TStream* objekta). Primjer 6.4.1 prikazuje kako se odabrana datoteka prilikom preuzimanja izravno pohranjuje na disk pomoću *TFileStream* objekta.

#### Primjer 6.4.2. Prikaz napretka preuzimanja datoteke

```
// HTTP_klijent - OnWorkBegin događaj  
void __fastcall TForm1::HTTP_klijentWorkBegin(TObject *ASender,  
                                              TWorkMAWorkMode AWorkMode,  
                                              __int64 AWorkCountMax){  
  
    ProgressBar1->Position = 0;  
    ProgressBar1->Max = AWorkCountMax;  
}  
  
// HTTP_klijent - OnWork događaj  
void __fastcall TForm1::HTTP_klijentWork(TObject *ASender,  
                                         TWorkMode AWorkMode,  
                                         __int64 AWorkCount){  
  
    // prikaz napretka preuzimanja datoteke  
    ProgressBar1->Position = AWorkCount;  
    Application->ProcessMessages();  
}  
  
// HTTP_klijent - OnWorkEnd događaj  
void __fastcall TForm1::HTTP_klijentWorkEnd(TObject *ASender,  
                                           TWorkMode AWorkMode){  
  
    ShowMessage("Preuzimanje datoteke je završeno!");  
}
```

Klijent Indy HTTP (*TIdHTTP*) sadrži događaje *OnWorkBegin*, *OnWork* i *OnWorkEnd* pomoću kojih je moguće prikazati napredak preuzimanja sadržaja (Primjer 6.4.2). Tako metoda za obradu događaja *OnWorkBegin* sadrži parametar *AWorkCountMax* čija je

vrijednost veličina sadržaja (broj bajtova) kojeg preuzimamo. U događaju *OnWork* parametrom *WorkCount* dohvaćamo koliko je sadržaja trenutno preuzeto, a metoda za obradu događaja *OnWorkEnd* izvršava se kada je preuzet sav sadržaj.

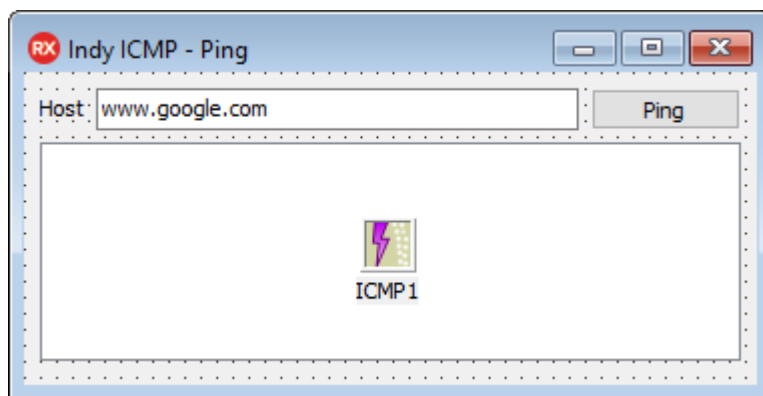
Ukoliko je potrebno prekinuti već započeto preuzimanje moguće je pozvati metodu *Disconnect* komponente *TIdHTTP*.

Komponenta *TIdInterceptThrottler* (*Throttler*) koristi se kada želimo imati mogućnost ograničavanja brzine preuzimanja i slanja podataka. Brzinu je moguće odrediti korištenjem svojstva *BitsPerSec* ili svojstva *RecvBitsPerSec* i *SendBitsPerSec*. Ovu komponentu moguće je povezati s komponentom *TIdHTTP* korištenjem svojstva *Intercept*.

Za potrebe autentifikacije i autorizacije Indy HTTP klijent uz poslani zahtjev može poslužitelju proslijediti korisničko ime i lozinku. U svojstvima *TIdHTTP* komponente također je moguće odrediti tipove sadržaja (*content type*), proizvoljna zaglavlja, inačicu korištenog HTTP protokola te još mnoge druge postavke.

## 6.5. Indy ICMP

ICMP (*Internet Control Message Protocol*) služi za slanje kontrolnih poruka i poruka o greškama. Primjerice, njime možemo provjeriti stanje nekog računala u mreži, mogućnost pristupa računalu i dobivanja ikakvog odgovora od njega. Najčešću implementaciju ICMP protokola možemo pronaći u naredbama *ping* i *tracert* (*tracert*).



Slika 6.5.1. Indy ICMP klijent



U komponenti *TIdlcmpClient* (*ICMP1*) moramo odrediti odredište (svojstvo *Host*), a nakon toga možemo izravno pozvati metodu *Ping*. Klikom na gumb sa slike izvršava se metoda *Button1Click* (Primjer 6.5.1).

#### Primjer 6.5.1. Poziv metode *Ping*

```
void __fastcall TForm1::Button1Click(TObject *Sender){
    ListBox1->Items->Clear();
    ICMP1->Host = EHost->Text;
    // ping 4 puta
    for(int i = 1; i <= 4; i++)
        try{
            ICMP1->Ping();
            Sleep(1000); // pauza od 1 sekunde..
            Application->ProcessMessages();
        }
        catch(...) {
            ListBox1->Items->Add("Ping nije uspio!");
            Application->ProcessMessages();
        }
}
```

Poruke se šalju u razmacima od jedne sekunde kako bismo ispitali dostupnost ciljanog računala u mreži u različitim vremenskim intervalima. Ako se poruka ne može poslati, metoda *Ping* uzrokovat će iznimku. Međutim, ako je računalo dostupno i ako je odgovorilo na našu poruku izvršava se funkcija zadužena za događaj *OnReply*.

#### Primjer 6.5.2. Obrada događaja *OnReply*

```
void __fastcall TForm1::ICMP1Reply(TComponent *ASender,
                                   const TReplyStatus *AReplyStatus){
    UnicodeString Rezultat;
    Rezultat = "Primio " + String(AReplyStatus->BytesReceived) +
        " bajta od " + AReplyStatus->FromIpAddress + "," +
        " vrijeme = " + String((int)AReplyStatus->MsRoundTripTime) +
        " ms, ttl = " + String((int)AReplyStatus->TimeToLive);
    ListBox1->Items->Add(Rezultat);
}
```

Ukoliko četiri puta uspješno "pingamo" stranicu *www.google.hr*, za svaki ping pokrenula bi se metoda koju demonstrira Primjer 6.5.2, pri čemu bismo kao rezultat dobili četiri poruke sljedećeg oblika:

Primio 72 bajta od 209.85.135.147, vrijeme = 37 ms, ttl = 243

Primio 72 bajta od 209.85.135.147, vrijeme = 31 ms, ttl = 243

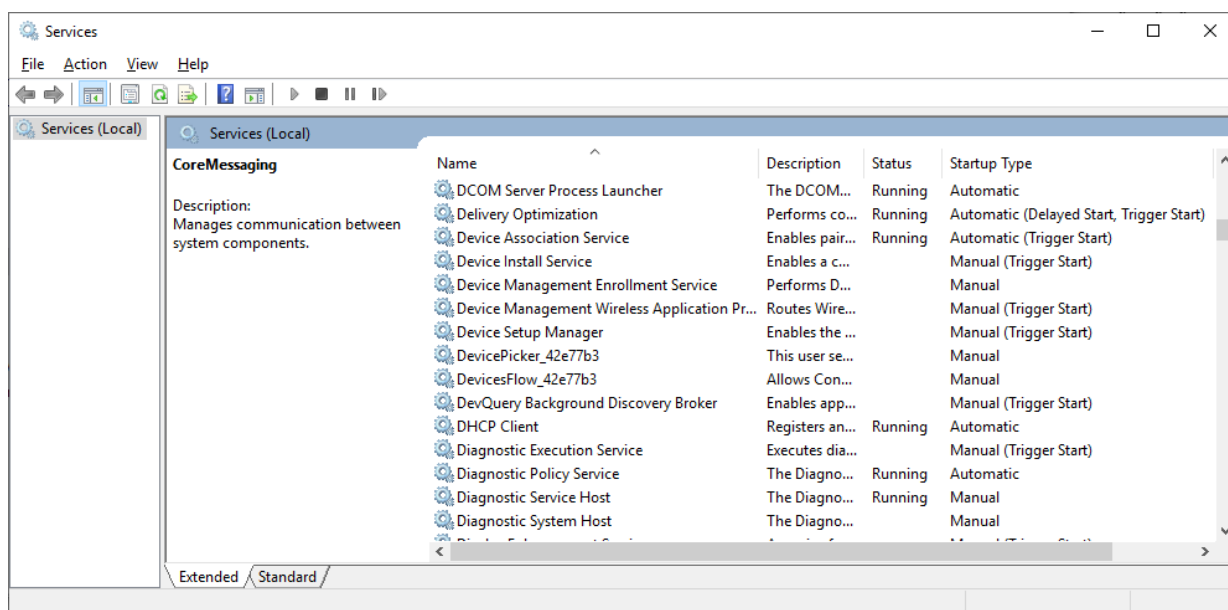
Primio 72 bajta od 209.85.135.147, vrijeme = 30 ms, ttl = 243

Primio 72 bajta od 209.85.135.147, vrijeme = 29 ms, ttl = 243

Parametar TTL skraćenica je od *Time To Live* - vrijeme života, a govori nam koliko je vremena potrebno za obradu paketa. Za svaki mrežni uređaj preko kojeg paket prijeđe TTL se povećava za vrijednost 1.

## 6.6. Windows servisi

Poslužitelji aplikacije vrlo često se implementiraju kao Windows servisi. To su aplikacije koje rade u pozadini operacijskog sustava, čekajući i odgovarajući na klijentske zahtjeve. Najčešće se pokreću automatski s podizanjem operacijskog sustava, a mogu raditi i dugo nakon odjave korisnika iz operacijskog sustava.

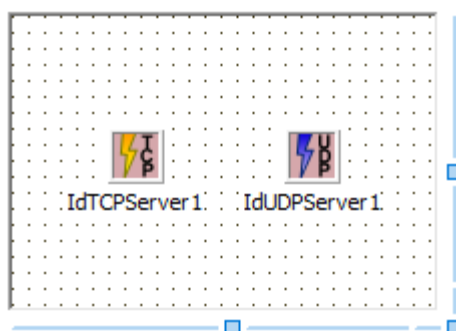


Slika 6.6.1. Popis Windows servisa

Windows servisi obično se izvršavaju pod administratorskim ovlastima čak i kada je prijavljen korisnik koji nije administrator. Zbog toga Windows servisi obično imaju veću kontrolu nad računalom nego obične aplikacije. [21]

Slika 6.6.1 prikazuje primjer popisa Windows servisa na računalu. Neki od najčešće korištenih su *DHCP Client*, *Print Spooler*, *Active Directory*, *Internet Connection Sharing (ICS)* itd. Na svakom računalu obično je instalirano minimalno stotinjak Windows servisa.

Da bismo kreirali Windows servis projekt u glavnom izborniku potrebno je odabrati stavku *File / New / Other...* Zatim, u prikazanom dijalogu pod stavkom *C++ Builder / Windows* potrebno je pronaći i odabrati stavku *Windows Service*.



Slika 6.6.2. Poslužitelj komponente u Windows servisu (*TService*)

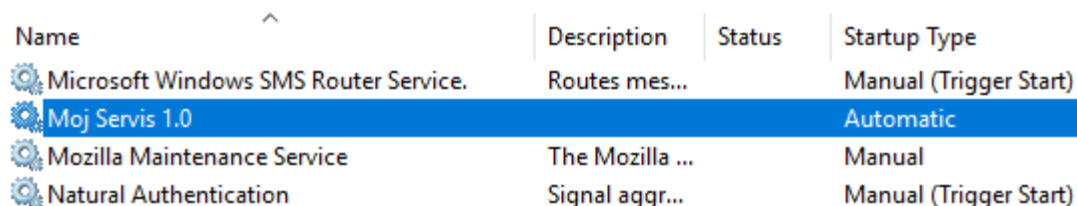
Za svaki novi *Windows Service* projekt automatski će se generirati instanca komponente *TService* (Slika 6.6.2). Među ostalim, ona ima i ulogu kontejnera u kojemu je moguće smjestiti nevizualne komponente poput *TIdTCPServer*, *TIdUDPServer* itd., te time u Windows servis implementirati jedan ili više mrežnih poslužitelja.

Ključna svojstva komponente *TService* su *DisplayName* (naziv servisa), *StartType* (*stAuto*, *stBoot*, *stManual* ili *stSystem*), *Password* (servis zaštićen lozinkom?) itd. Radom Windows servisa moguće je dodatno upravljati i obradom događaja poput *OnStart*, *OnPause*, *OnContinue*, *OnStop* itd.

Rezultat prevođenja bit će izvršna EXE datoteka. Za razliku od obične aplikacije koja se tipično pokreće dvostrukim klikom, aplikaciju Windows servisa prvo je potrebno instalirati na sljedeći način:

```
<ime_servisa.exe> /install
```

Deinstalacija se izvršava na gotovo identičan način, odnosno upotrebom parametra */uninstall*. Nakon instalacije, servis je moguće pronaći i pokrenuti u popisu servisa (*Administrative Tools / Services*), kao što to prikazuje Slika 6.6.3.



Name	Description	Status	Startup Type
Microsoft Windows SMS Router Service.	Routes mes...		Manual (Trigger Start)
Moj Servis 1.0			Automatic
Mozilla Maintenance Service	The Mozilla ...		Manual
Natural Authentication	Signal aggr...		Manual (Trigger Start)

Slika 6.6.3. Moj Servis 1.0

Opis Windows servisa (eng. *description*) nije izravno moguće odrediti u komponenti *TService*, već ažuriranjem Windows registra. Potrebno je pristupiti ključu `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\<NazivServisa>` te unutar njega kreirati i inicijalizirati vrijednost *Description*.

Nakon što je Windows servis instaliran i pokrenut poslužitelj(i) unutar njega počeo će osluškivati i odgovarati na zahtjeve klijenata.

## 7. Web servisi

### 7.1. SOAP klijent

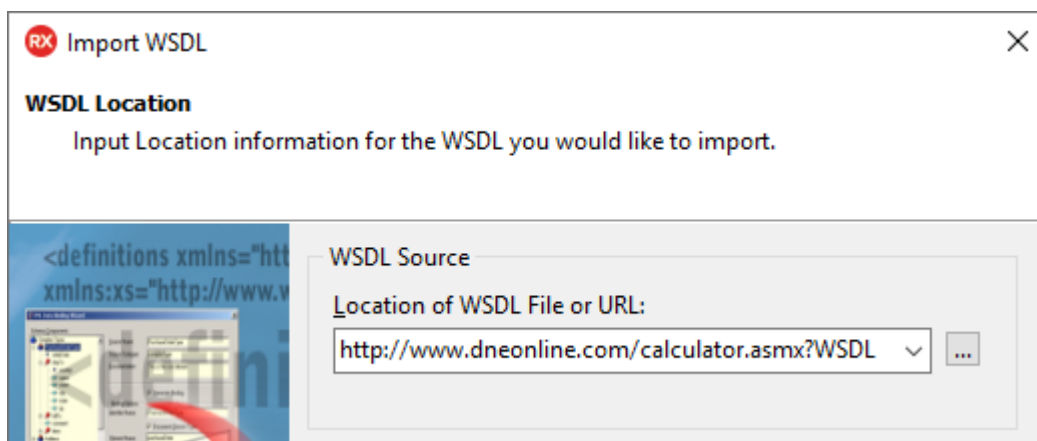
SOAP (*Simple Object Access Protocol*) komunikacijski je protokol koji omogućuje komunikaciju između aplikacija koje se izvode na različitim platformama te pisanih u različitim programskim jezicima. Komunikacija se temelji na razmjeni tekstualnih poruka korištenjem HTTP protokola, pri čemu SOAP klijent u obliku XML dokumenta šalje zahtjev poslužitelju, te nakon toga očekuje odgovor. Na Internetu je moguće pronaći mnoštvo javno dostupnih SOAP web servisa, a jedan od njih je i *Calculator Web Service*:

<http://www.dneonline.com/calculator.asmx>

Da bi SOAP klijent i poslužitelj mogli komunicirati klijent mora znati sadržaj WSDL (*Web Services Description Language*) datoteke ciljanog web servisa. To je XML datoteka koja sadrži lokaciju SOAP web servisa i opis njegovih funkcionalnosti.

<http://www.dneonline.com/calculator.asmx?WSDL>

Na osnovu sadržaja WSDL datoteke C++ Builder je automatski u mogućnosti generirati sav potreban programski kôd koji klijent aplikaciji omogućuje korištenje web servisa, odnosno njegovih funkcionalnosti. U tu svrhu koristi se WSDL uvoznik (eng. *importer*), kao što prikazuje Slika 7.1.1.



Slika 7.1.1. WSDL uvoznik

WSDL uvoznik pokreće se odabirom stavke *File / New / Other...*, nakon čega je u prikazanom dijalogu pod dijelom *C++ Builder / Web* potrebno odabrati stavku *WSDL Importer*. WSDL uvozniku obavezno je potrebno priložiti lokaciju WSDL datoteke, a ako se ona nalazi na zaštićenoj mrežnoj lokaciji moguće je dodatno navesti korisničko ime i lozinku za pristup toj mrežnoj lokaciji.

Za navedeni primjer web servisa WSDL uvoznik generirat će datoteke *calculator.h* i *calculator.cpp*. One će sadržavati sva potrebna Delphi sučelja koja će klijent aplikaciji omogućiti korištenje tog web servisa.

#### **Primjer 7.1.1. Calculator.h - sučelje web servisa Calculator**

```
__interface INTERFACE_UUID("{42FA0556-A79E-45A1-65B8-EDB9F7C6702F}")
CalculatorSoap : public IInvokable{
public:
    virtual int Add(const int intA, const int intB) = 0;
    virtual int Subtract(const int intA, const int intB) = 0;
    virtual int Multiply(const int intA, const int intB) = 0;
    virtual int Divide(const int intA, const int intB) = 0;
};

typedef DelphiInterface<CalculatorSoap> _di_CalculatorSoap;

_di_CalculatorSoap GetCalculatorSoap(bool useWSDL = false,
System::String addr = System::String(), SoapHttpClient::THHTTPRIO* HTTPRIO = 0);
```

U generiranoj datoteci zaglavlja (Primjer 7.1.1) nalazi se definicija Delphi sučelja *CalculatorSoap* koje sadrži četiri metode (*Add*, *Subtract*, *Multiply* i *Divide*). Upravo ove metode predstavljaju funkcionalnosti web servisa *Calculator* koje je sada moguće koristiti u klijent aplikaciji.

Delphi sučelje je poput C++ klase koja ne sadrži podatkovne članove, već isključivo čiste virtualne metode. Sučelja obično imaju pridruženi GUID (*Globally Unique Identifier*). On nije obavezan, ali većina kôda koji koristi sučelja očekuje da će pronaći GUID. Da bi u C++ Builderu bilo jednostavnije koristiti Delphi sučelja podrazumijevano se koristi deklaracija *typedef*, kao što je to prikazuje Primjer 7.1.1. [22]

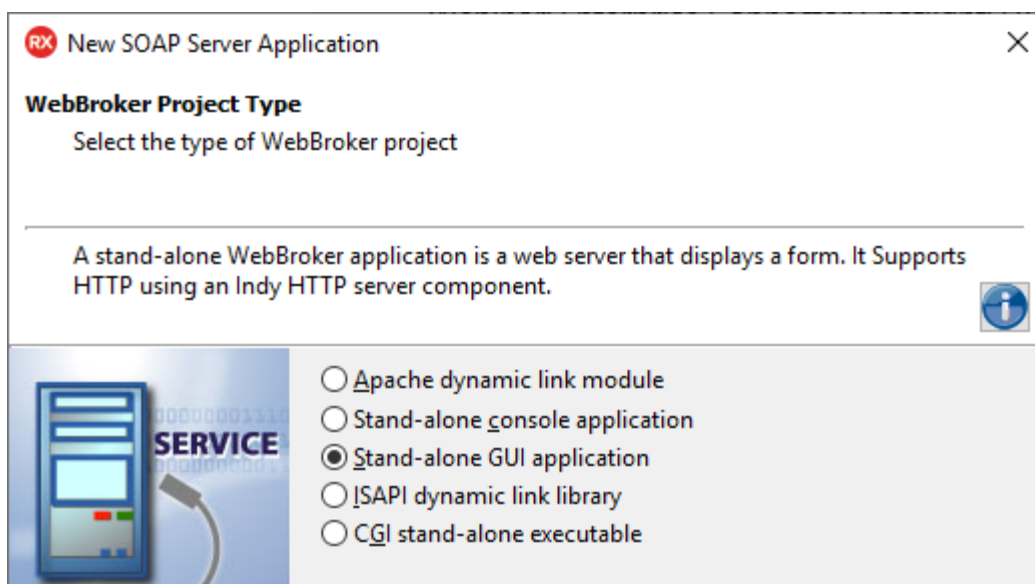
**Primjer 7.1.2. Poziv metode Multiply**

```
#include "calculator.h"
// ...
void __fastcall TForm1::Button1Click(TObject *Sender){
    _di_CalculatorSoap Calc = GetCalculatorSoap();
    ShowMessage(Calc->Multiply(3, 5)); // 15
}
```

U klijent aplikaciji potrebno je uključiti generiranu datoteku zaglavlja *calculator.h* (Primjer 7.1.2). To će klijentu omogućiti kreiranje i inicijalizaciju objekta *Calc* (sučelje *\_di\_CalculatorSoap*) čijim se korištenjem pozivaju metode SOAP web servisa *Calculator*. Za uspješan poziv postojećih metoda klijent mora imati Internet vezu.

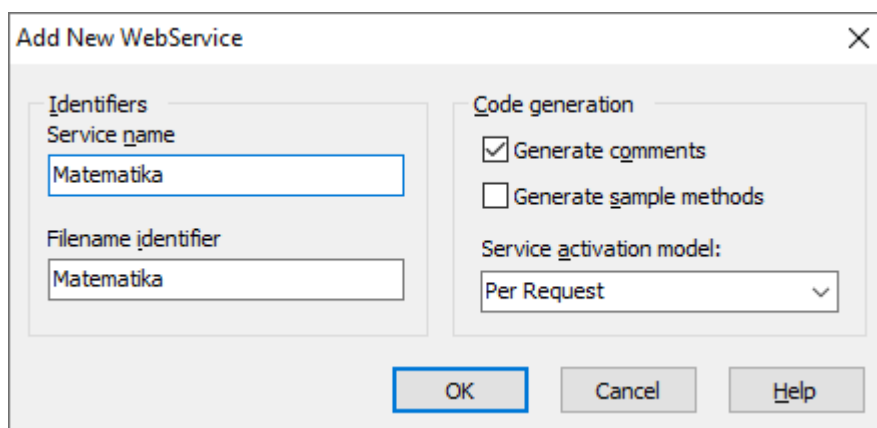
## 7.2. SOAP poslužitelj

Pomoću WebBroker tehnologije C++ Builder omogućuje razvoj SOAP poslužitelja za Windows i Linux platforme. Ovisno o ciljanoj platformi, SOAP poslužitelj je moguće razviti kao Apache modul, samostalnu aplikaciju komandne linije i CGI samostalnu izvršnu datoteku (Linux). Dodatno, na Windows platformi SOAP poslužitelj može biti razvijan i kao samostalna aplikacija s grafičkim sučeljem te ISAPI DLL biblioteka (*plugin*) za IIS web poslužitelj.



Slika 7.2.1. Novi SOAP poslužitelj – tip aplikacije

U svrhu lakšeg testiranja preporuka je započeti razvoj SOAP poslužitelja kao samostalne VCL aplikacije s grafičkim sučeljem (Slika 7.2.1). Iste projektne datoteke kasnije će biti moguće koristiti za prevođenje SOAP poslužitelja u ISAPI (dll), CGI (exe) ili neki drugi tip aplikacije pogodan za smještaj na ciljanom web poslužitelju. Novi VCL SOAP poslužitelj podrazumijevano će koristiti port 8080, a omogućuje i korištenje protokola HTTPS uvozom certifikata X.509.



Slika 7.2.2. Kreiranje novog servisa unutar SOAP poslužitelja

SOAP poslužitelj u sebi treba sadržavati barem jedan SOAP web servis. Štoviše, nakon završetka rada čarobnjaka (Slika 7.2.1) automatski se pojavljuje dijalog za kreiranje novog SOAP web servisa (Slika 7.2.2). Na osnovu njegova imena automatski će se generirati Delphi sučelje u kojemu će biti potrebno smjestiti prototipe virtualnih metoda novog web servisa. Tako će za web servis imena *Matematika* biti generirano sučelje *IMatematika* te implementacijska klasa *TMatematikaImpl* koja treba implementirati navedeno sučelje. Zaglavlje i implementacija klase *TMatematikaImpl* podrazumijevano će se nalaziti u datotekama *Matematika.h* i *Matematika.cpp*.

Preporuka je za svaki zahtjev prema web servisu kreirati novu instancu implementacijske klase (Stavka *Service activation model* – *Per Request*). U tom slučaju svaka nova instanca automatski će se uništiti nakon obrade zaprimljenog zahtjeva. Za obradu svih zahtjeva također je moguće koristiti samo jednu (globalnu) instancu implementacijske klase, no u okruženju s mnogo konkurentnih zahtjeva to nije preporučljivo.



**Primjer 7.2.1. Matematika.h – sučelje IMatematika**

```
__interface INTERFACE_UUID("{D8C2B814-84DE-4C14-856C-3D66FA2FE8B3}")
IMatematika : public IInvokable{
public:
    // metoda kvadrat
    virtual double kvadrat(double a) = 0;
};
typedef DelphiInterface<IMatematika> _di_IMatematika;
```

Sučelje *IMatematika* za demonstraciju sadrži samo metodu *kvadrat* (Primjer 7.2.1). Kako je riječ o čistoj virtualnoj metodi njena implementacija morat će se nalaziti u implementacijskoj klasi *TMatematikaImpl* (Primjer 7.2.2).

**Primjer 7.2.2. Matematika.cpp – implementacija metode kvadrat**

```
class TMatematikaImpl : public TInvokableClass, public IMatematika{
public:
    // implementacija metode kvadrat
    double kvadrat(double a) { return a * a; }
    .../
};
```

Klasa *TInvokableClass* podržava sučelje *ISOAPHeaders* tako da SOAP poslužitelj može obraditi zaglavlja koja prate zahtjeve i dodati zaglavlja za odlazne odgovore. Konačno, pokretanjem VCL SOAP poslužitelj aplikacije na portu 8080 (<http://localhost:8080/>) pojavljuje se info web stranica u kojoj se nalazi objavljeno sučelje *IMatematika* i njena metoda *kvadrat*. Također je dostupna i poveznica na generiranu WSDL datoteku za potrebe klijent aplikacija (<http://localhost:8080/wsd/IMatematika>).

**Primjer 7.2.3. Testni SOAP klijent**

```
#include "IMatematika.h"
//...
void __fastcall TForm1::Button1Click(TObject *Sender){
    _di_IMatematika mat = GetIMatematika();
    ShowMessage(mat->kvadrat(5)); // 25
}
```

Da bismo testirali kreirani web servis možemo napraviti VCL SOAP klijent aplikaciju koja će korištenjem WSDL uvoznika generirati programski kôd za pristup metodi *kvadrat*. Primjer 7.2.3 demonstrira njen poziv.

## 7.3. REST klijent

Roy Fielding 2000. godine u svojoj doktorskoj disertaciji predstavio je REST (*Representational State Transfer*). REST predstavlja arhitekturni stil (načelo) gdje su podaci i funkcionalnosti predstavljeni resursima. Resurs predstavlja sve ono je dostupno putem URI-ja (*Uniform Resource Identifier*), a dohvaćanje i obrada resursa moguća je metodama poput GET, POST, PUT i DELETE. REST je u početku opisan u kontekstu HTTP-a, ali za razliku od njega ne predstavlja protokol, već tek stil koji opisuje kako bi se HTTP trebao koristiti.

REST uglavnom koristi XML i JSON za razmjenu podataka, dok također podržava HTML, ZIP, PDF, PNG, CSV i gotovo sve ostale formate koje podržava HTTP protokol. U svojoj komunikaciji SOAP koristi HTTP protokol, ali također može koristiti SMTP i FTP, dok REST koristi samo HTTP. Međutim, REST daje mnogo bolje performanse pri radu s podacima. Gotovo sve usluge temeljene na oblaku nude podršku za dohvat i obradu podataka korištenjem REST-a, uključujući Amazon, Twitter, Facebook, Google, YouTube i mnoge druge. [23]

Svaki web servis koji slijedi REST načela naziva se RESTful. REST načela su:

1. Klijent - poslužitelj način komunikacije (zahtjev – odgovor)
2. Komunikacija bez stanja (poslužitelj ne pamti stanja između zahtjeva)
3. Slojevit sustav (postoje posrednici između klijenta i poslužitelja)
4. Privremena pohrana podataka (eng. *cache*)
5. Uniformno sučelje (identifikacija resursa, manipulacija putem reprezentacije...)
6. Skripte na zahtjev (neobavezno).

Ova načela ujedno razlikuju RESTful API od HTTP API-ja koji tek zahtjeva da se komunikacija odvija korištenjem HTTP protokola. Ipak, HTTP klijent može komunicirati s REST poslužiteljem tako što korištenjem URL-a identificira resurs, HTTP metodom akciju

nad resursom, a reprezentaciju svojstvom *ContentType* (text/plain, application/xml, application/json, text/html itd.), kao što to prikazuje Primjer 7.3.1.

#### Primjer 7.3.1. Indy HTTP klijent i REST web servis

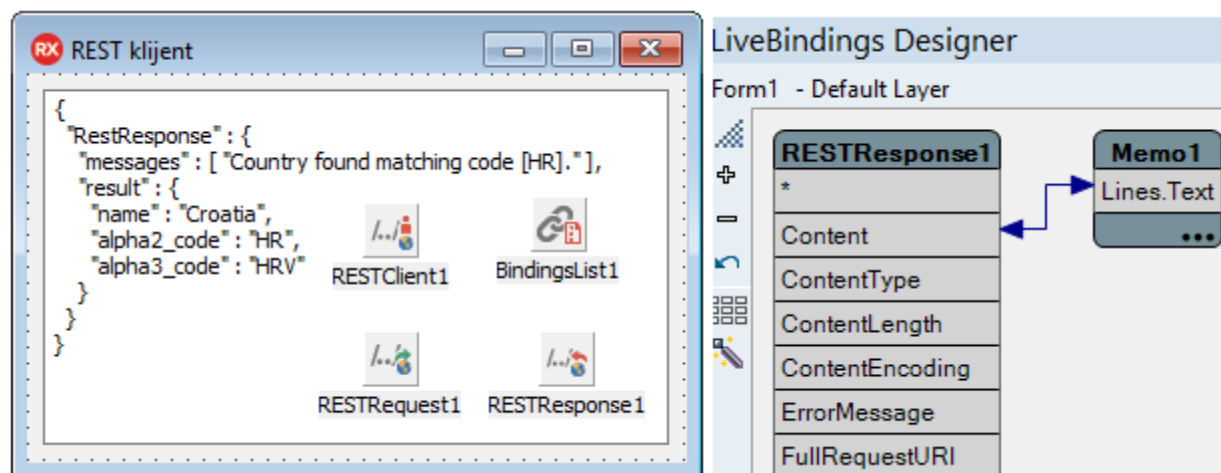
```
void __fastcall TForm1::Button1Click(TObject *Sender){
    String odgovor;
    odgovor = IdHTTP1->Get("http://services.groupkt.com/country/get/iso2code/HR");
    ShowMessage(odgovor);
}
```

Rezultat izvršavanja metode *Button1Click* bit će ispis odgovora REST poslužitelja u JSON formatu (Primjer 7.3.2).

#### Primjer 7.3.2. Odgovor REST poslužitelja

```
{
  "RestResponse" : {
    "messages" : [ "Country found matching code [HR]." ],
    "result" : {
      "name" : "Croatia",
      "alpha2_code" : "HR",
      "alpha3_code" : "HRV"
    }
  }
}
```

Međutim, REST klijent je moguće implementirati i korištenjem *REST Client* skupa komponenti (Slika 7.3.1).



Slika 7.3.1. REST klijent komponente

Ovom prilikom korištene su komponente *REST Client* – *TRESTClient*, *TRESTRequest* i *TRESTResponse*, čija su gotovo sva ključna svojstva (osim *BaseURL*) automatski inicijalizirana na sljedeći način:

```
RESTClient1->BaseURL = "http://services.groupkt.com/country/get/iso2code/HR";
RESTRequest1->Client = RESTClient1;
RESTRequest1->Method = rmGET;
RESTRequest1->Response = TRESTResponse1;
TRESTResponse1->ContentType = "application/json";
```

Dodatno, svojstvo *Content* komponente *TRESTResponse1* je pomoću *LiveBindings* dizajnera potrebno povezano s komponentom *TMemo1* (Slika 7.3.1). Tako se sada u vrijeme dizajna dvostrukim klikom na komponentu *RESTRequest1* može poslati zahtjev REST poslužitelju, a dobiveni odgovor automatski prikazati u komponenti *TMemo1*. Alternativno, zahtjev se može poslati i u vrijeme rada aplikacije pozivom metode *Execute*, komponente *TRESTRequest*.

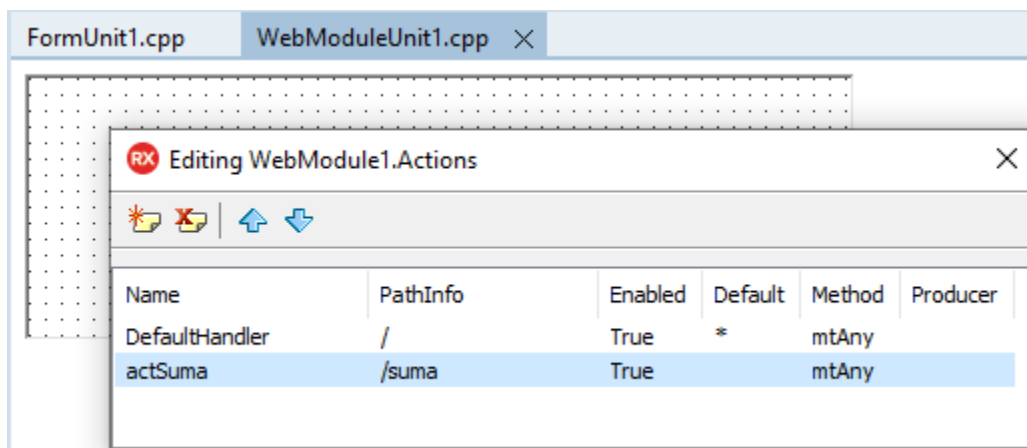
U *REST Client* skupu komponenti nalaze se i komponente za autentifikaciju: *THTTPBasicAuthenticator*, *TOAuth1Authenticator* i *TOAuth2Authenticator*. U slučaju njihova korištenja svojstvo *Authenticator* komponente *TRESTClient* potrebno je inicijalizirati na odgovarajuću vrijednost.

## 7.4. WebBroker REST poslužitelj

Poslužitelj aplikacije (service) trenutno je moguće razvijati korištenjem tehnologija WebBroker, DataSnap i RAD Server. Jedino je WebBroker bez ograničenja dostupan u svim izdanjima alata RAD Studio, dok su DataSnap i RAD Server dostupni samo u izdanjima Enterprise i Architect.

Razvoj novog WebBroker REST poslužitelja počinje kreiranjem novog *Web Server Application* (*File / New / Other... / C++ Builder / Web*) projekta. U svrhu bržeg razvoja i jednostavnijeg testiranja preporuka je započeti njegov razvoj kao Windows VCL aplikaciju s grafičkim sučeljem koja koristi mrežni priključak 8080. Kasnije će iste datoteke

(programski kôd) biti moguće iskoristiti za prevođenje poslužitelja u CGI (exe), ISAPI (dll) ili Apache modul.



Slika 7.4.1. TWebModule akcije

Čarobnjak će generirati aplikaciju koja sadrži komponentu *WebModule1*, a svaka *TWebModule* komponenta sadrži jednu ili više akcija koje služe za obradu klijentskih zahtjeva (Slika 7.4.1). *TWebModule* nasljeđuje klasu *TCustomWebDispatcher* koja definira svojstva *Request* i *Response*. Tako komponenta *TWebModule* prilikom obrade akcija može dohvatiti zahtjeve klijenata i poslati im odgovor. [24] Svi zahtjevi klijenata automatski se obrađuju u zasebnim dretvama, a u svakoj od njih dinamički se alocira nova instanca komponente *TWebModule*.

Razvoj WebBroker REST servisa svodi se na kreiranje i obradu akcija. Svaka akcija ima svojstva poput *Name*, *PathInfo*, *Enabled*, *Default*, *Method* itd. Tako svojstvo *PathInfo* određuje putanju na koju klijent šalje zahtjev, dok svojstvo *Method* određuje metodu, odnosno tip zahtjeva (*mtAny* - podrazumijevano, *mtGet*, *mtPut*, *mtDelete*, *mtPost*, *mtHead* i *mtPatch*). Također, samo jedna akcija može biti podrazumijevana (eng. *default*) i ona inicijalno obrađuje zahtjev iz korijenske putanje (/). Čarobnjak za nju automatski generira implementaciju događaja *OnAction* (Primjer 7.4.1).

#### Primjer 7.4.1. Implementacija podrazumijevane akcije

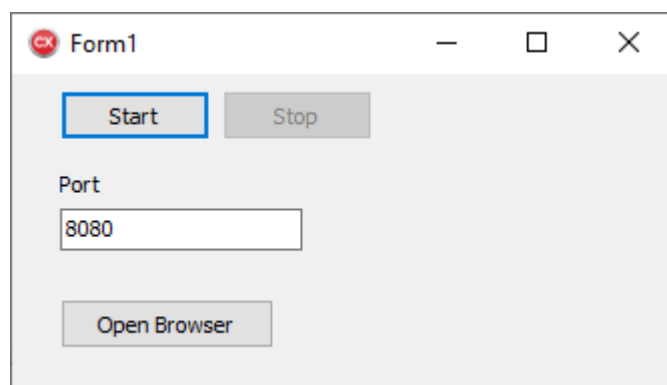
```
void __fastcall TWebModule1::WebModule1DefaultHandlerAction(TObject *Sender,
TWebRequest *Request, TWebResponse *Response, bool &Handled) {
    Response->Content =
        "<html>"
```

```

    "<head><title>Web Server Application</title></head>"
    "<body>Web Server Application</body>"
    "</html>";
}

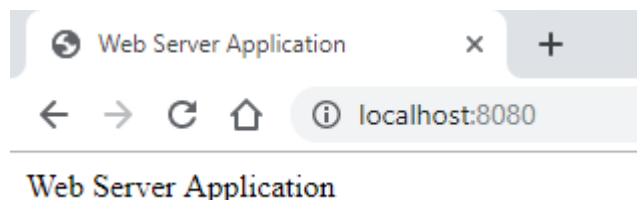
```

Pokretanjem generirane aplikacije pojavljuje se glavni dijalog Windows VCL aplikacije koji omogućuje pokretanje i zaustavljanje rada web poslužitelja, odabir mrežnog priključka i otvaranje web preglednika (Slika 7.4.2).



Slika 7.4.2. Windows VCL aplikacija za upravljanje web poslužiteljem

Nakon pokretanja rada web poslužitelja potrebno je otvoriti web preglednik. On sada ima ulogu klijenta koji metodom GET podrazumijevano dohvaća sadržaj s korijenske putanje (/). Zbog toga se automatski pokreće obrada podrazumijevane akcije, pa se kao rezultat dobije sadržaj koji prikazuje Slika 7.4.3.



Slika 7.4.3. Obrada podrazumijevane akcije

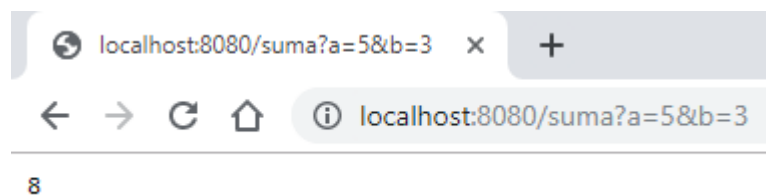
Za primjer pretpostavimo da naš web poslužitelj klijentu želi omogućiti obradu jednostavne matematičke operacije zbrajanja dvaju cijelih brojeva. Klijent bi pozivom metode GET u putanji `/suma` predao dva broja i očekivao odgovor (njihovu sumu). Da bismo omogućili obradu ovakvog zahtjeva u komponenti `TWebModule` potrebno je dodati još jednu akciju

(*actSuma*), kao što to prikazuje Slika 7.4.1. Zatim, njenu obradu potrebno je implementirati događajem *OnAction* (Primjer 7.4.2).

#### Primjer 7.4.2. Implementacija akcije *actSuma*

```
void __fastcall TWebModule1::WebModule1actSumaAction(TObject *Sender,
TWebRequest *Request, TWebResponse *Response, bool &Handled){
    // da li je pozvana metoda GET?
    if(Request->MethodType == mtGet){
        try{
            int A = Request->QueryFields->Values["a"].ToInt();
            int B = Request->QueryFields->Values["b"].ToInt();
            Response->StatusCode = 200;
            Response->ReasonString = "OK";
            Response->ContentType = "text/plain; charset=UTF-8";
            Response->Content = IntToStr(A + B);
        }
        catch(Exception& E){
            Response->StatusCode = 400;
            Response->ReasonString = "Bad request";
            Response->ContentType = "text/plain; charset=UTF-8";
            Response->Content = E.Message;
        }
    }
}
```

Akcija *actSuma* podrazumijevano obrađuje zahtjeve bilo kojeg tipa (*Method = mtAny*). Zbog toga se naredbom *if* prvo provjerava tip metode u zahtjevu (je li riječ o metodi GET). Cijeli brojevi A i B predaju se korištenjem URL parametara, a ukoliko nisu predani ili nije riječ o cijelim brojevima klijent će kao odgovor dobiti informaciju o greški.



Slika 7.4.4. Testiranje akcije *actSuma*

Kao odgovor klijentu poslana su četiri podatka (*StatusCode*, *ReasonString*, *ContentType* i *Content*). U ovom slučaju riječ je o slanju otvorenog teksta (*text/plain*) u kojemu je klijent

dobio odgovor na svoj upit (Slika 7.4.4). Međutim, kada je riječ o razmjeni veće količine podataka web servisi i klijenti vrlo često komuniciraju razmjenom XML i JSON dokumenata.

U sljedećem primjeru pretpostavimo da klijent od našeg web poslužitelja želi dobiti popis svih prim brojeva u intervalu [A, B] u obliku XML dokumenta. Prvo je u komponenti *TWebModule* potrebno dodati novu akciju (*actPrim*) koja zaprima GET zahtjeve (*Method* = *mtGet*) na putanji */prim*, a zatim implementirati obradu akcije (zahtjeva) u događaju *OnAction* (Primjer 7.4.3).

#### **Primjer 7.4.3. Implementacija akcije *actPrim***

---

```
// da li je predani broj prim broj?
bool IsPrim(int broj){
    if (broj <= 1) return false;
    for (int i = 2; i <= floor(pow(broj, 0.5)); i++)
        if (broj % i == 0)
            return false;
    return true;
}

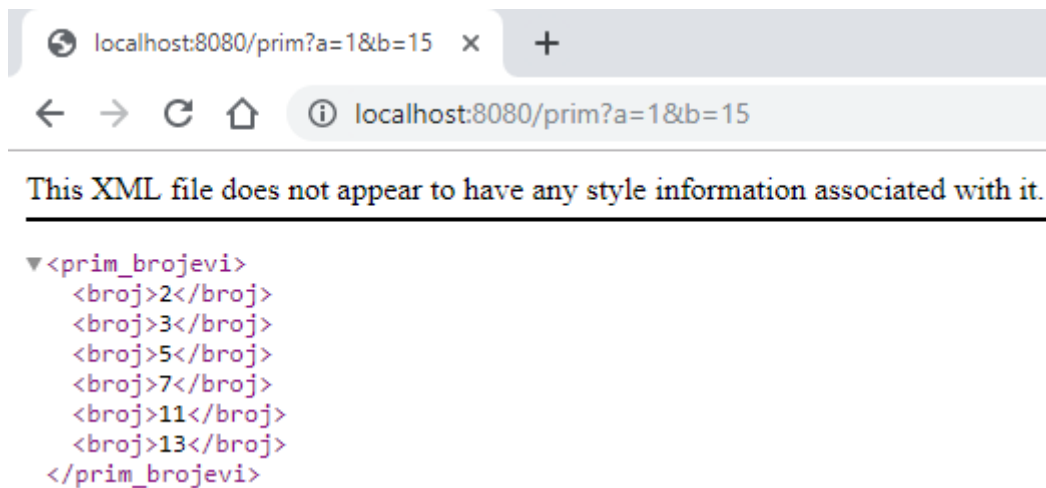
// akcija actPrim
void __fastcall TWebModule1::WebModule1actPrimAction(TObject *Sender,
TWebRequest *Request, TWebResponse *Response, bool &Handled){
    int A = Request->QueryFields->Values["a"].ToInt();
    int B = Request->QueryFields->Values["b"].ToInt();

    // formiranje xml odgovora
    String xml = "<prim_brojevi>";
    for(int i = A; i <= B; i++)
        if(IsPrim(i))
            xml += "<broj>" + IntToStr(i) + "</broj>";
    xml += "</prim_brojevi>";

    Response->StatusCode = 200;
    Response->ReasonString = "OK";
    Response->ContentType = "application/xml; charset=UTF-8";
    Response->Content = xml;
}
```



XML dokument može se sastavljati kao obični tekst (UnicodeString). Međutim, prilikom slanja odgovora svojstvom *ContentType* potrebno je naglasiti da je riječ o XML dokumentu (application/xml) kako bi klijent (primjerice, web preglednik) na ispravan način znao protumačiti dobiveni odgovor (Slika 7.4.5). Na sličan način moguće je formirati odgovor u obliku JSON, HTML ili nekog drugog tipa dokumenta.



Slika 7.4.5. Testiranje akcije *actPrim*

Klijent može koristiti URL parametre za predaju podataka, a može ih predati i u samoj URL putanji. Primjerice, zahtjev `localhost:8080/prim?a=1&b=15` mogao bi se napisati i kao `localhost:8080/prim/1/15`. Da bismo podržali drugu mogućnost potrebno je na drugačiji način implementirati podrazumijevanu akciju komponente *TWebModule* (Primjer 7.4.4).

#### Primjer 7.4.4. Predaja podataka korištenjem URL putanje

```
#include <vector>
#include <sstream>
// ...
// dijelove putanje (stavke) ubaci u vektor
std::vector<String> Putanja_Vector(String putanja){
    std::vector<String> v;
    std::wstringstream wss(putanja.w_str());
    std::wstring stavka;
    while (getline(wss, stavka , L'/'){
        if (!stavka.empty())
            v.push_back(stavka.c_str());
    }
    return v;
}
```

```
}  
  
// implementacija podrazumijevane akcije komponente TWebModule  
void __fastcall TWebModule1::WebModule1DefaultHandlerAction(TObject *Sender,  
TWebRequest *Request, TWebResponse *Response, bool &Handled){  
    // dohvat cjelovite putanje zahtjeva  
    std::vector<String> stavka = Putanja_Vector(Request->PathInfo);  
    if(stavka.size() == 3 && stavka[0].LowerCase() == "prim"){  
        Request->QueryFields->Values["a"] = stavka[1].ToDouble();;  
        Request->QueryFields->Values["b"] = stavka[2].ToDouble();;  
        WebModule1actPrimAction(Sender, Request, Response, Handled);  
    }  
}
```

Sada će web poslužitelj za klijentski zahtjev `localhost:8080/prim/1/15` dati isti odgovor kao i za zahtjev `localhost:8080/prim?a=1&b=15` (Slika 7.4.5).

Osim slanja podataka u tekstualnom obliku, web poslužitelj na zahtjev može odgovoriti i slanjem toka podataka (eng. *stream*). U tom slučaju se umjesto člana *Content* koristi član *ContentStream* koji treba sadržavati željeni tok. Sukladno sadržaju toka potrebno je prilagoditi i vrijednost člana *ContentType*.

#### **Primjer 7.4.5. Podatkovni tok kao odgovor web poslužitelja**

```
void __fastcall TWebModule1::WebModule1WebActionItem1Action(TObject *Sender,  
TWebRequest *Request, TWebResponse *Response, bool &Handled){  
    TFileStream* fs = new TFileStream("slika.jpg", fmOpenRead);  
    Response->StatusCode = 200;  
    Response->ReasonString = "OK";  
    Response->ContentType = "image/jpeg";  
    // Response->SetCustomHeader("Content-Disposition", "filename=slika.jpg" );  
    Response->ContentStream = fs;  
}
```

Primjer 7.4.5 demonstrira kako web poslužitelj za slanje slike u jpg formatu koristi tok *TFileStream*. Sadržaj toka pridružen je članu `Response->ContentStream`, a objekt *fs* nije potrebno dealocirati, već će se to dogoditi automatski od strane *Response* objekta nakon završenog prijenosa podataka. Štoviše, ručno dealociranje objekta *fs* uzrokovalo bi grešku na klijent strani prilikom preuzimanja podataka.

Web poslužitelj možemo razviti i na način da dozvoljava pristup samo autoriziranim korisnicima. Najčešće se korištenjem korisničkog imena i lozinke (*Basic Access Authentication*) identificira korisnika, nakon čega mu se daju odgovarajuće dozvole za pristup i korištenje pojedinih resursa (autorizacija).

#### Primjer 7.4.6. Autentifikacija korisnika web poslužitelja

```
#include <System.NetEncoding.hpp>
#include <System.StrUtils.hpp>
// ...

void __fastcall TWebModule1::WebModule1WebActionItem1Action(TObject *Sender,
TWebRequest *Request, TWebResponse *Response, bool &Handled){
    TBytes bytes;
    UnicodeString login, KorisnickoIme, Lozinka;
    // ukoliko uz zahtjev nije poslano korisničko ime i lozinka...
    if(Request->Authorization == ""){
        Response->StatusCode = 401;
        Response->ReasonString = "Unauthorized";
        Response->ContentType = "text/plain; charset=UTF-8";
        Response->Content = "Niste ovlašteni pristupiti ovom resusu!";
        Response->WWWAuthenticate = "Basic realm=\"MojREST prijava\"";
        Response->SendResponse(); // završi obradu zahtjeva
    }

    // dekodiraj korisničke podatke (počevši od 7. znaka)
    bytes = TNetEncoding::Base64->DecodeStringToBytes(&Request->Authorization[6]);
    login = StringOf(bytes);
    KorisnickoIme = SplitString(login, ":")[0];
    Lozinka = SplitString(login, ":")[1];
    // ispis korisničkih podataka
    Response->Content = Request->Authorization;
    Response->Content = Response->Content + "<p></p>";
    Response->Content = Response->Content +
        "Dekodirano = " + StringOf(bytes) + "</br>";
    Response->Content = Response->Content +
        "Korisničko ime = " + KorisnickoIme + "</br>";
    Response->Content = Response->Content + "Lozinka = " + Lozinka + "</br>";
}
```

Ukoliko prilikom slanja zahtjeva na web poslužitelj nisu poslani korisničko ime i lozinka podatkovni član `Request->Authorization` bit će prazan. U protivnom, taj podatkovni član

sadržavat će korisničke podatke u šifriranom obliku Base64. Šifriranje Base64 vrlo često se koristi upravo na webu kako bi se binarni podaci predstavili tekстом te tako lakše prenijeli mrežom. Tako će se korisničko ime **ABC** i lozinka **123** web poslužitelju poslati kao jedan Base64 šifrirani tekst **Basic QUJDOjEyMw==**. Base64 ne predstavlja kriptografski algoritam pa se ovaj oblik autentifikacije u svrhu zaštite podataka najčešće koristi u kombinaciji s protokolom HTTPS.

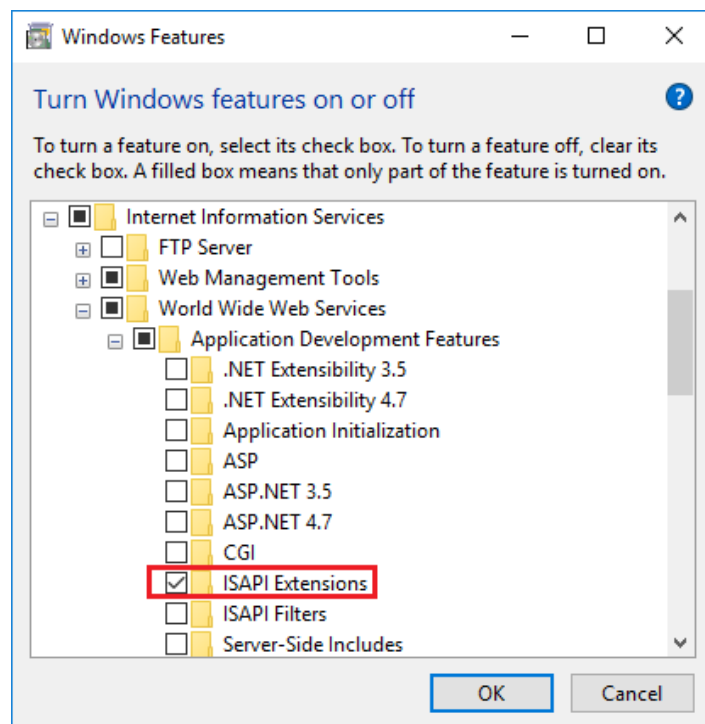
Primjer 7.4.6 demonstrira kako se šifrirani tekst Base64 pretvara u niz bajtova korištenjem metode `TNetEncoding::Base64->DecodeStringToBytes`, a zatim u ASCII tekst funkcijom `StringOf`. Dešifriranje niza Base64 u ovom slučaju treba započeti od sedmog znaka, budući da prvih šest znakova predstavlja fiksni prefiks "Basic". Konačnim dešifriranjem dobije se tekst **ABC:123** iz kojeg zasebno treba dohvatiti korisničko ime, a zasebno lozinku.

## 7.5. Instalacija ISAPI DLL-a u IIS web poslužitelju

Nakon faze razvoja i testiranja poslužitelj aplikaciju će u konačnici biti potrebno instalirati na web poslužitelju kao modul CGI (*Common Gateway Interface*), ISAPI (*Internet Server Application Program Interface*) ili Apache. CGI (exe) tip aplikacije danas se smatra zastarjelim. Umjesto njega preporučuje se korištenje ISAPI (dll) ekvivalenta, dok se Apache modul može koristiti i kao rješenje za web poslužitelje na Linux platformi.

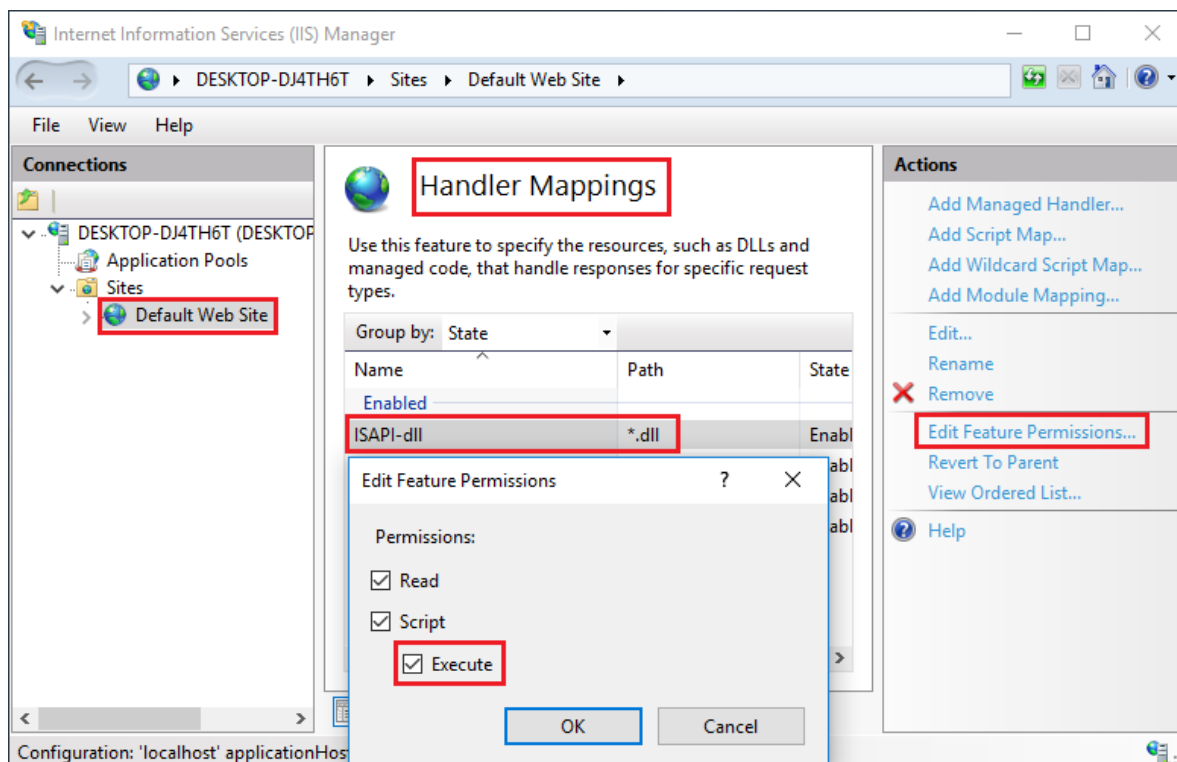
Web poslužitelj će za svaki zahtjev prema CGI aplikaciji kreirati novi proces (instancu CGI aplikacije) prilikom čega se troši velika količina resursa i mnogo procesorskog vremena. Da bi se poboljšale performanse web poslužitelja i smanjilo korištenje resursa prikladnije je koristiti ISAPI (dll) tip aplikacije koja je stalno učitana u adresnom prostoru web poslužitelja i tamo kreira dretvu za obradu svakog klijentskog zahtjeva. [25]

ISAPI aplikaciju moguće je kao dodatak (*plugin*) instalirati u Windows IIS (*Internet Information Services*) web poslužitelju. Podrazumijevano, IIS web poslužitelj prethodno je potrebno instalirati tako da sadrži značajku *ISAPI Extensions* (Slika 7.5.1). Dodatno je moguće instalirati i podršku za ISAPI filtere, CGI aplikacije itd.



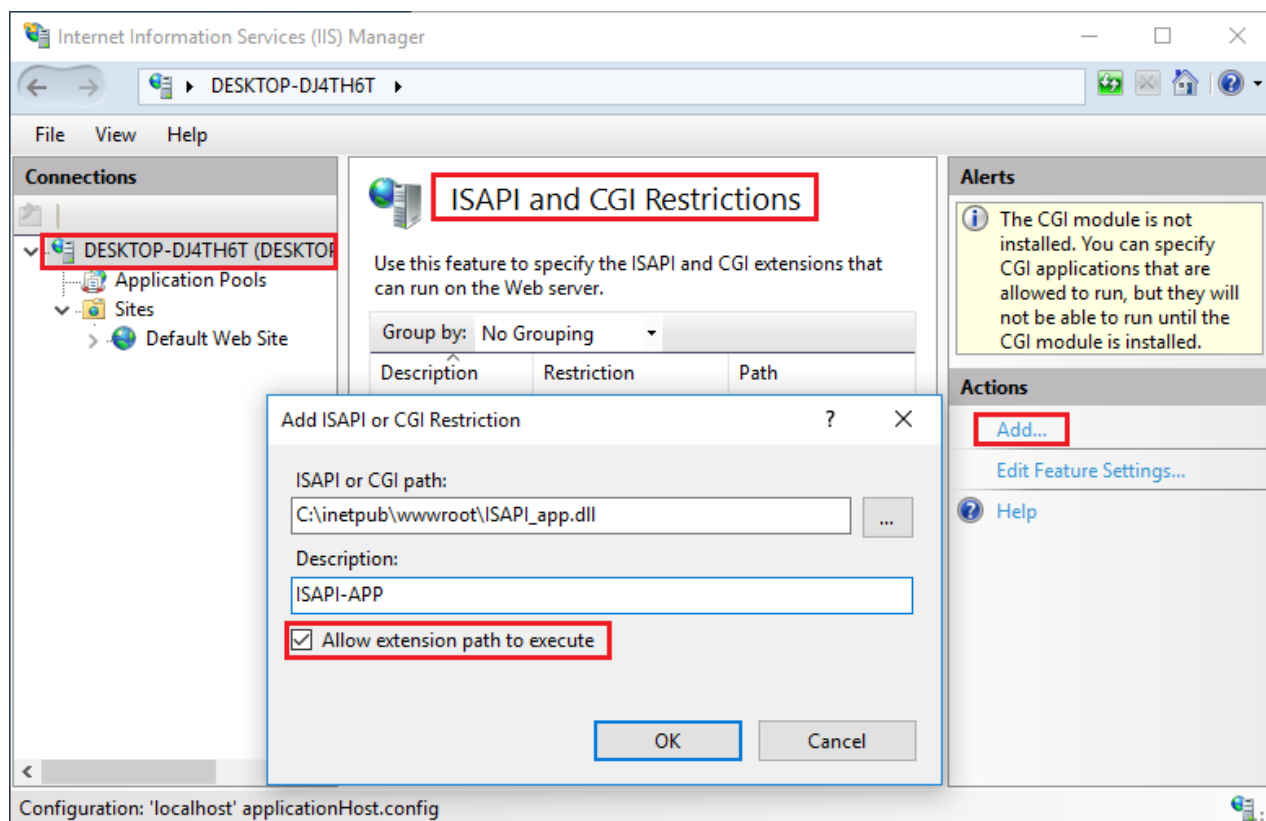
Slika 7.5.1. Instaliranje podrške za ISAPI ekstenzije

Ukoliko ISAPI aplikaciju želimo smjestiti u prostor podrazumijevane web stranice (*Default Web Site*), njenu dll datoteku potrebno je kopirati u mapu *C:\inetpub\wwwroot*. Zatim, na podrazumijevanoj web stranici potrebno je dozvoliti izvršavanje aplikacija ISAPI-dll.



Slika 7.5.2. Dozvola za izvršavanje ISAPI-dll aplikacija

S lijeve strane IIS Manager aplikacije potrebno je odabrati stavku *Sites / Default Web Site*, te u ponuđenom popisu odabrati stavku *Handler Mappings*. U prikazanom dijalogu potrebno je označiti redak *ISAPI-dll* te sa desne strane odabrati stavku *Edit Feature Permissions...* Zatim, potrebno je dozvoliti operaciju izvršavanja aplikacija ISAPI dll odabirom stavke *Execute* (Slika 7.5.2).

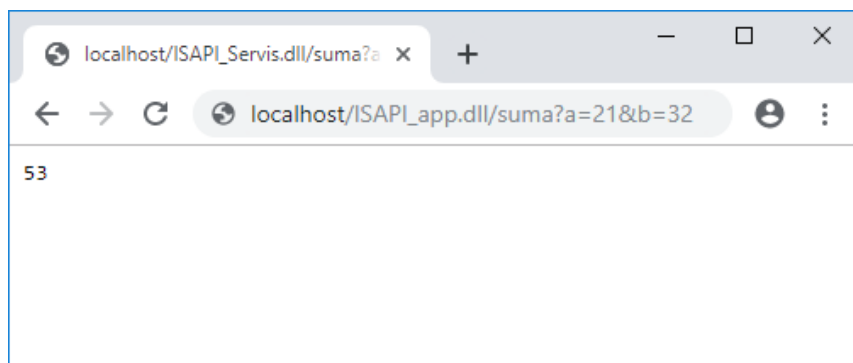


Slika 7.5.3. ISAPI i CGI ograničenja

Dodatno, u samom web poslužitelju potrebno je dozvoliti izvršavanje ISAPI dll aplikacije. Na lijevoj strani IIS Manager aplikacije potrebno je odabrati čvor poslužitelja te u ponuđenom popisu odabrati stavku *ISAPI and CGI Restrictions*. Odabirom stavke *Edit Feature Settings* moguće je dozvoliti izvršavanje svih aplikacija ISAPI u mapi poslužitelja (stavka *Allow unspecified ISAPI modules*), a iz sigurnosnih razloga stavkom *Add* dozvolu za izvršavanje dati svakoj pojedinoj ISAPI aplikaciji (Slika 7.5.3).

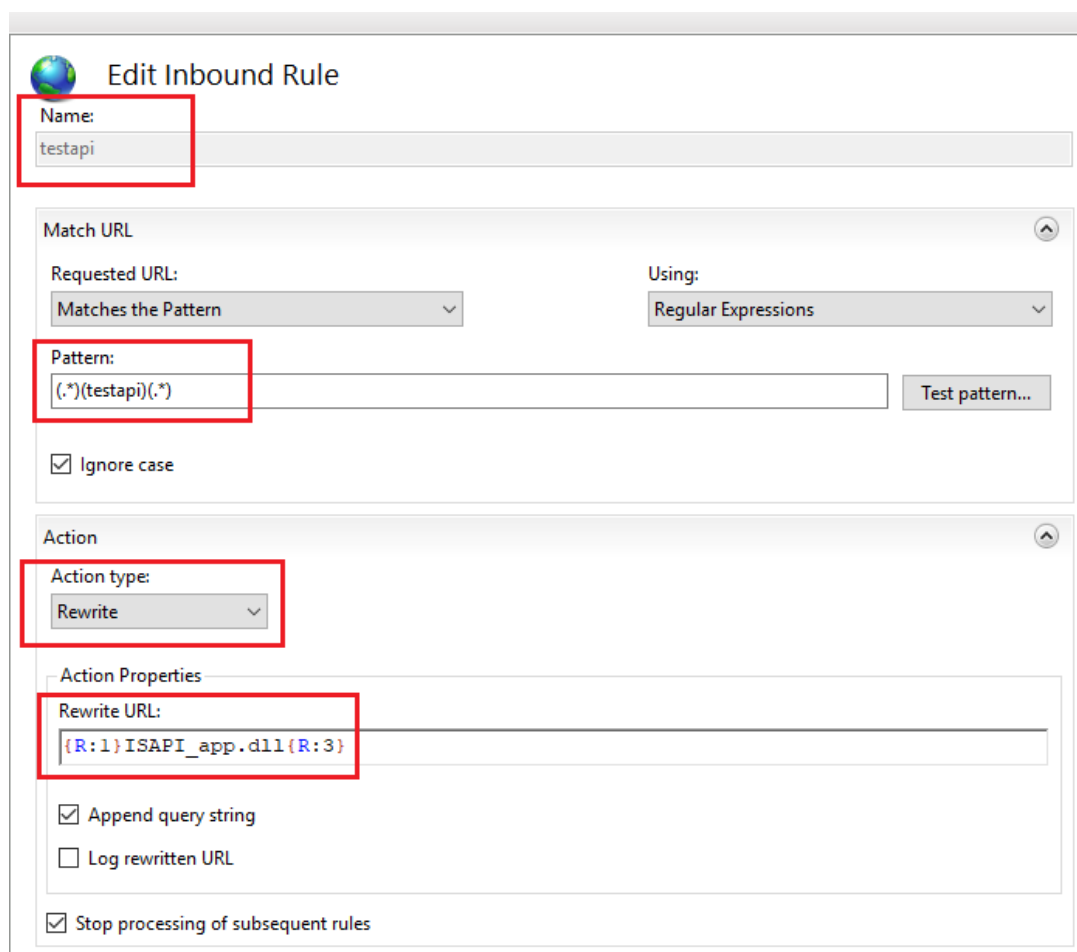
Važno je napomenuti da IIS web poslužitelj podrazumijevano dozvoljava izvršavanje samo 64-bitnih ISAPI aplikacija. Da bi se omogućila podrška za 32-bitne aplikacije potrebno je u

naprednim postavkama korištenog aplikacijskog bazena (eng. *application pool*) dozvoliti izvršavanje 32-bitnih aplikacija.



Slika 7.5.4. Izvršavanje ISAPI aplikacije u web pregledniku

Instaliranu ISAPI aplikaciju sada je moguće koristiti izravno iz web preglednika (Slika 7.5.4). URL sadrži naziv ISAPI dll datoteke, a njega je moguće maknuti ili zamijeniti nekim drugim nizom znakova korištenjem dodatka *URL Rewrite* za IIS (Slika 7.5.5).



Slika 7.5.5. URL Rewrite - kreiranje novog pravila za preusmjerenje

Dodatak *URL Rewrite* potrebno je preuzeti i instalirati s Microsoft web stranica. Kreiranjem pravila za preusmjeravanje s postavkama kao na slici naziv ISAPI dll datoteke (*ISAPI\_app.dll*) u URL-u bit će moguće zamijeniti s nizom znakova *testapi*. Tako će sada zahtjev

```
/testapi/suma?a=56&b=45
```

biti preusmjeren i obrađen kao

```
/ISAPI_app.dll/suma?a=56&b=45
```

Zbog korištenja akcije tipa *Rewrite* klijent (primjerice, web preglednik) neće vidjeti destinaciju na koju je preusmjeren njegov zahtjev. U protivnom bi trebalo koristiti akciju tipa *Redirect*.

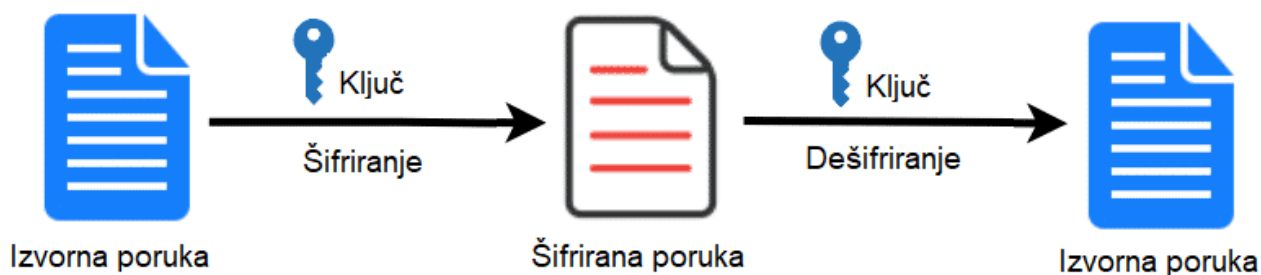


## 8. Kriptografija

### 8.1. Simetrični kriptografski algoritmi

U svrhu zaštite (tajnosti) podataka moguće je koristiti simetrične i asimetrične kriptografske algoritme te funkcije sažimanja (eng. *hash functions*). Šifriranjem pomoću kriptografskih algoritama podaci u čitljivom obliku pretvaraju se u nečitljiv oblik, s idejom da ih samo osoba kojoj su podaci namijenjeni može vratiti nazad u izvorni (čitljivi) oblik. Funkcije sažimanja dodatno se koriste kada je potrebno provjeriti integritet i izvornost podataka, potvrditi pošiljatelja (kod digitalnog potpisa), a vrlo često se koriste i za provjeru ispravnosti lozinki.

Simetrični kriptografski algoritmi razvijeni su da budu brzi. Primarno se koriste kod šifriranja velike količine podataka i dijele se u dvije kategorije: tok (eng. *stream*) i blok (eng. *block*) algoritmi. Tok algoritmi šifriraju podatke bit po bit, dok blok algoritmi u blokovima po 64, 128 ili više bitova. Neki od najčešće korištenih simetričkih kriptografskih algoritama su DES, Triple-DES, Blowfish, IDEA, AES itd.

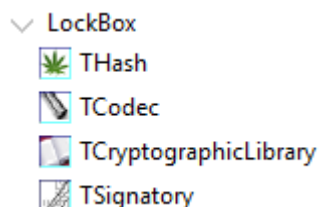


Slika 8.1.1. Princip rada simetričnog kriptografskog algoritma [26]

Ovi tipovi algoritama koriste jedan tajni ključ. Izvorna poruka prvo se šifrira odabranim ključem, a zatim se kod primatelja dešifrira nazad u izvorni oblik korištenjem istog ključa (Slika 8.1.1). Za što bolju zaštitu podataka poželjno je koristiti ključ što veće duljine kojeg pošiljatelj i primatelj inicijalno mogu razmijeniti korištenjem jednog od asimetričnih kriptografskih algoritama.

C++ Builder ima mogućnost korištenja raznih kriptografskih biblioteka u jezicima C, C++ i Delphi. Neke od njih su Crypto++, Botan, DCCrypt, LockBox itd. Primjerice, u repozitoriju

*GetIt Package Manager* moguće je pronaći i instalirati besplatnu kriptografsku biblioteku TurboPower LockBox koja omogućuje implementaciju raznih simetričnih i asimetričnih kriptografskih algoritama (AES, DES, 3DES, Blowfish, Twofish, RSA), funkcija sažimanja (MD5, SHA1, SHA2) i digitalnog potpisa.



Slika 8.1.2. TurboPower LockBox komponente

Nakon instalacije biblioteke LockBox dostupne su četiri komponente (Slika 8.1.2). Pomoću njihovih svojstava moguće je izabrati željeni kriptografski algoritam ili funkciju sažimanja, a njihovim metodama šifrirati, dešifrirati i sažeti sadržaj.

#### Primjer 8.1.1. Šifriranje i dešifriranje teksta korištenjem algoritma AES

```
void __fastcall TForm1::SifrirajTekstClick(TObject *Sender){
    Codec1->CryptoLibrary = CryptographicLibrary1;
    Codec1->BlockCipherId = "native.AES-256";
    Codec1->ChainModeId = "native.CBC";
    Codec1->Password = "tajni_kljuc";

    String Poruka = "Tajna poruka";
    String SifriranaPoruka;
    Codec1->EncryptString(Poruka, SifriranaPoruka, TEncoding::UTF8);
    ShowMessage(SifriranaPoruka); // npr., WfmyxjfoGZh3Ky6UoTryvjDsQzM=
    Codec1->DecryptString(Poruka, SifriranaPoruka, TEncoding::UTF8);
    ShowMessage(Poruka); // tajna poruka
}
```

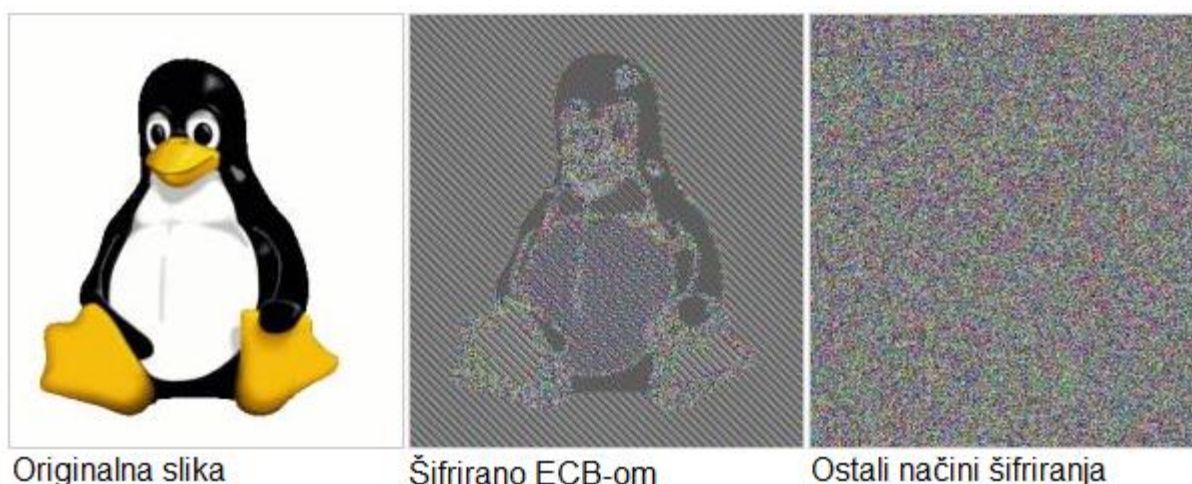
Pretpostavimo da na dijalogu (formi) postoje komponente *CryptographicLibrary1* i *Codec1*. Komponenta *Codec1* koristi kriptografsku biblioteku *CryptographicLibrary1* u kojoj su implementirani svi LockBox podržani algoritmi. Nadalje, u komponenti *Codec1* potrebno je izabrati željeni algoritam (primjerice, AES), njegov način rada (CBC, CFB, CTR, ECB, OFB ili PCBC) te odrediti tajni ključ. Tek tada je moguće početi sa šifriranjem željenog sadržaja

korištenjem metoda *EncryptString*, *EncryptStream*, *EncryptMemory* i *EncryptFileW*. Ekvivalentno, za dešifriranje koriste se metode sa sufiksom *Decrypt* (Primjer 8.1.1).

Rezultat šifriranja je niz bajtova koji se u obliku teksta (oblik Base64) pohranjuje u odredišnu varijablu, odnosno objekt tipa *UnicodeString*. Međutim, ovisno o odabranom načinu rada algoritma (CBC, CFB, ECB itd.) rezultat šifriranja jedne poruke najčešće će biti različit prilikom svakog sljedećeg šifriranja. Primjerice, rezultat šifriranja teksta "Tajna poruka" korištenjem CBC načina rada i ključa "tajni\_kljuc" može dati sljedeće rezultate:

```
WfmyxjfoGZh3Ky6UoTryvjDsQzM= // Dešifrirano: Tajna poruka
hctQpAOxFiF4ovcr7UxPskFlvag= // Dešifrirano: Tajna poruka
YsGT5/oV+C+MRNFCGEHG4xWN7jc= // Dešifrirano: Tajna poruka
O5zyp59imDJXbHcpuViRcGALeGk= // Dešifrirano: Tajna poruka
...
```

Naime, CBC (eng. *Chain Block Coding*) šifrira poruku po blokovima na način da prije šifriranja trenutnog bloka izvrši operaciju XOR s prethodnim blokom podataka. [27] Za izvršavanje operacije XOR nad prvim blokom podataka koristi se inicijalizacijski vektor, odnosno pseudoslučajna vrijednost zbog koje će šifriranje iste poruke svaki puta dati drugačiji rezultat čak i kada je u pitanju korištenje istog ključa. Jedino način šifriranja ECB (eng. *Electronic CodeBook*) uvijek daje isti rezultat šifriranja za istu poruku, što nikako nije poželjno jer tada napadač može prepoznati uzorke u šifriranoj poruci te na taj način otkriti neke ili sve dijelove izvorne poruke.



Slika 8.1.3. Usporedba ECB-a i ostalih načina šifriranja

Slika 8.1.3 prikazuje rezultat šifriranja slike korištenjem ECB načina rada. Šifrirana slika vrlo nalikuje originalnoj slici jer se svaki blok podataka šifrirao zasebno, odnosno neovisno o prethodnim podatkovnim blokovima. Drugi načini rada koji koriste pseudoslučajnu vrijednost, odnosno inicijalizacijski vektor, kao rezultat daju potpuno neprepoznatljivu sliku. Zbog toga bi način šifriranja ECB svakako trebalo izbjegavati.

### Primjer 8.1.2. Šifriranje i dešifriranje datoteke korištenjem tokova

```
#include <memory>
// ...

void __fastcall TForm1::SifrirajDatotekuClick(TObject *Sender){
    Codec1->CryptoLibrary = CryptographicLibrary1;
    Codec1->BlockCipherId = "native.AES-256";
    Codec1->ChainModeId = "native.CBC";
    Codec1->Password = "tajni_kljuc";
    std::unique_ptr<TFileStream> IzvornaDatoteka(new TFileStream(
        "test.txt", fmOpenRead));
    std::unique_ptr<TFileStream> SifriranaDatoteka(new TFileStream(
        "test.sifrirano", fmCreate));

    // šifriraj datoteku test.txt
    Codec1->EncryptStream(IzvornaDatoteka.get(), SifriranaDatoteka.get());
    std::unique_ptr<TFileStream> DesifriranaDatoteka(new TFileStream(
        "test.desifrirano", fmCreate));

    // dešifriraj datoteku test.sifrirano
    Codec1->DecryptStream(DesifriranaDatoteka.get(), SifriranaDatoteka.get());
}
```

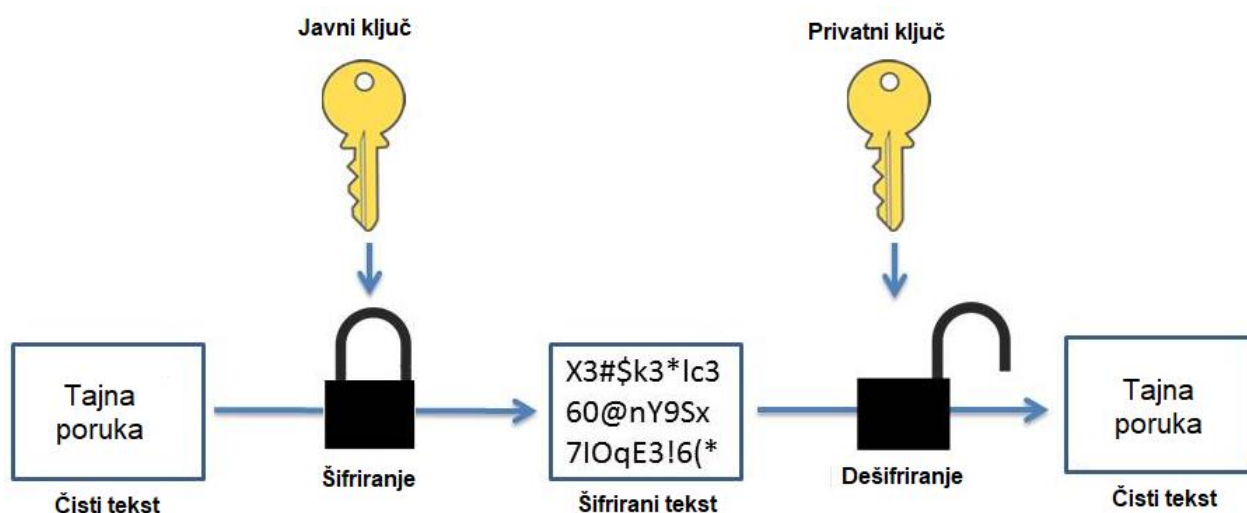
Iako je za šifriranje i dešifriranje datoteka moguće koristiti metode *EncryptFileW* i *DecryptFileW*, Primjer 8.1.2 demonstrira kako ih je moguće šifrirati i dešifrirati korištenjem tokova. Umjesto toka *TFileStream* moguće je koristiti tok bilo kojeg tipa, što na jednostavan način omogućuje šifriranje i dešifriranje bilo kojeg tipa sadržaja, bio on smješten u datoteci na disku ili u radnoj memoriji.

Izvorna datoteka *test.txt* šifrira se korištenjem algoritma AES-256 i načina rada CBC. Šifrirana datoteka pohranjuje se pod nazivom *test.sifrirano*, a zatim se dešifrira i pohranjuje u datoteku *test.desifrirano*. Dešifrirana i izvorna datoteka konačno trebaju imati identični sadržaj.

## 8.2. Asimetrični kriptografski algoritmi

Za razliku od simetričnih kriptografskih algoritama koji koriste samo jedan tajni ključ, asimetrični koriste dva ključa (javni i privatni). Javni ključ dostupan je svima i koristi se za šifriranje poruke koju može dešifrirati samo osoba koja ima odgovarajući privatni ključ. Ipak, asimetrični kriptografski algoritmi puno su sporiji od simetričnih, pa se vrlo rijetko koriste za šifriranje i dešifriranje velike količine podataka. U takvim situacijama pomoću njih se tek razmjenjuju simetrični ključevi, pa se daljnja komunikacija (razmjena poruka) nastavlja korištenjem simetričnih kriptografskih algoritama.

RSA, imenovan po trojici svojih autora (Rivest–Shamir–Adleman), jedan je od najpopularniji asimetričnih kriptografskih algoritama, a bazira se na praktičnim poteškoćama faktorizacije velikih brojeva koji nastaju kao produkt dvaju prostih brojeva. Primjerice, faktorizacija RSA-768 trajala je dvije godine uz pomoć 1000 procesorskih jezgri, a za faktorizaciju RSA-1024 pretpostavlja se da bi bilo potrebno 7481 godinu na istim računalima. RSA se danas može koristiti čak i u varijantama od 2048 i 4096 bita.



Slika 8.2.1. Princip rada asimetričnog kriptografskog algoritma [28]

Izvorna poruka se prvo šifrira javnim ključem primatelja. Javni ključ može objavljen na web stranicama primatelja, biti smješten unutar certifikata koji potvrđuje identitet primatelja (primjerice, certifikat X.509) ili na neki drugi način biti dostupan javnosti. Zatim šifriranu poruku dešifrira primatelj svojim privatnim ključem (Slika 8.2.1).

**Primjer 8.2.1. Generiranje RSA ključeva**

```

#include <memory>
// ...
void __fastcall TForm1::GenerirajKljuceveClick(TObject *Sender){
    Codec1->CryptoLibrary = CryptographicLibrary1;
    Codec1->BlockCipherId = "native.RSA";
    Codec1->AsymmetricKeySizeInBits = 1024;
    Codec1->ChainModeId = "native.CBC";
    Signatory1->Codec = Codec1;

    std::unique_ptr<TMemoryStream> privateKey (new TMemoryStream);
    std::unique_ptr<TMemoryStream> publicKey (new TMemoryStream);
    // generiraj i pohrani ključeve RSA
    if(Signatory1->GenerateKeys()){
        Signatory1->StoreKeysToStream(privateKey.get(),
                                     TKeyStoragePartSet() << partPrivate);
        Signatory1->StoreKeysToStream(publicKey.get(),
                                     TKeyStoragePartSet() << partPublic);
        privateKey->SaveToFile("private_key.bin");
        publicKey->SaveToFile("public_key.bin");
    }
}

```

Ukoliko već ne postoje, prvo će biti potrebno generirati privatne i javne ključeve RSA. LockBox 3 nudi mogućnost generiranja tih ključeva korištenjem komponente *TSignatory*, odnosno njene metode *GenerateKeys*. Generirani ključevi pohranjuju se u tokove, a sadržaj tokova u datoteke na disku *private\_key.bin* i *public\_key.bin* (Primjer 8.2.1). Kasnije će za potrebe šifriranja i dešifriranja poruka te ključeve biti moguće učitati izravno iz navedenih datoteka.

**Primjer 8.2.2. Šifriranje poruke korištenjem javnog ključa primatelja**

```

#include <memory>
// ...
void __fastcall TForm1::SifrirajClick(TObject *Sender){
    Codec1->CryptoLibrary = CryptographicLibrary1;
    Codec1->BlockCipherId = "native.RSA";
    Codec1->AsymmetricKeySizeInBits = 1024;
    Codec1->ChainModeId = "native.CBC";
    // učitaj izvornu poruku iz komponente Edit1

```

```
String Poruka = Edit1->Text;
String sifriranaPoruka;

std::unique_ptr<TMemoryStream> publicKey (new TMemoryStream);
// učitaj javni ključ
publicKey->LoadFromFile("public_key.bin");
Signatory1->LoadKeysFromStream(publicKey.get(),
                                TKeyStoragePartSet() << partPublic);
Codecl->EncryptString(Poruka, sifriranaPoruka, TEncoding::UTF8);
// prikaži šifriranu poruku u komponenti Edit1
Edit1->Text = sifriranaPoruka;
}
```

Postupak šifriranja sličan je kao i u slučaju korištenja simetričnog kriptografskog algoritma. Prvo je iz datoteke potrebno učitati javni ključ primatelja te zatim njime šifrirati željenu poruku. Tako Primjer 8.2.2 metodom *EncryptString* šifrira poruku (tekst) sadržanu u komponenti *Edit1*.

#### ***Primjer 8.2.3. Dešifriranje poruke korištenjem privatnog ključa primatelja***

---

```
#include <memory>
// ...

void __fastcall TForm1::DesifrirajClick(TObject *Sender){
    Codecl->CryptoLibrary = CryptographicLibrary1;
    Codecl->BlockCipherId = "native.RSA";
    Codecl->AsymmetricKeySizeInBits = 1024;
    Codecl->ChainModeId = "native.CBC";
    String Poruka;

    std::unique_ptr<TMemoryStream> privateKey (new TMemoryStream);
    // učitaj privatni ključ
    privateKey->LoadFromFile("private_key.bin");
    Signatory1->LoadKeysFromStream(privateKey.get(),
                                    TKeyStoragePartSet() << partPrivate);
    Codecl->DecryptString(Poruka, Edit1->Text, TEncoding::UTF8);
    // prikaži dešifriranu poruku u komponenti Edit1
    Edit1->Text = Poruka;
}
```

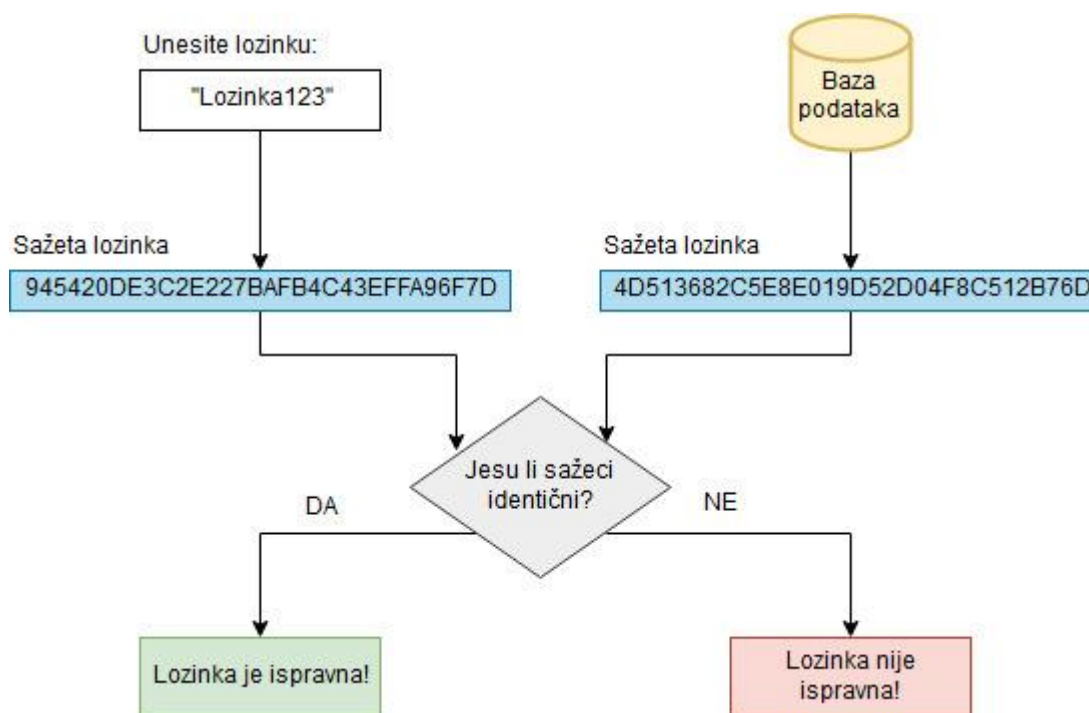


Postupak dešifriranja vrlo je sličan. Događa se kod primatelja poruke koji ju dešifrira korištenjem svog privatnog ključa (Primjer 8.2.3).

### 8.3. Funkcije sažimanja

Funkcije sažimanja (eng. *hash functions*) matematičke su funkcije koje pretvaraju ulazne podatke u sažete bitovne nizove fiksne duljine. Dok kriptografski algoritmi omogućuju šifriranje i dešifriranje podataka, funkcije sažimanja jednosmjerne su funkcije kod kojih je iz sažetog oblika gotovo nemoguće vratiti izvorne podatke. Vrlo često se koriste za provjeru lozinki i integriteta podataka, a njihova primjena prisutna je i u raznim strukturama podataka, algoritmima itd.

Rezultat funkcije sažimanja mora biti jedinstven i uvijek isti za pojedini ulazni podatak. Pri odabiru funkcije treba uzeti u obzir da se zbog već otkrivenih slabosti i nedostataka mnoge od njih danas više ne koriste. Tako se primjerice funkcije MD4, MD5 i SHA-1 zbog otkrivenih preklapanja (više različitih podataka može generirati istu sažetu vrijednost) i drugih uspješnih metoda napada danas više ne smatraju dovoljno sigurnima već se umjesto njih preporučuju SHA-2 i SHA-3 funkcije. [29]



Slika 8.3.1. Provjera ispravnosti lozinke korištenjem funkcije sažimanja



Funkcije sažimanja omogućuju provjeru ispravnosti lozinki čak i ako aplikacija ne zna lozinku u obliku otvorenog teksta. Naime, nakon što korisnik unese lozinku korištenjem odabrane funkcije sažimanja izračuna se njen sažetak koji se zatim provjeri sa sažetkom koji je zapisan u bazi podataka. Ukoliko su oba sažetka identična lozinka je ispravna i korisnik je autentificiran (Slika 8.3.1).

#### *Primjer 8.3.1. Izračun SHA-256 sažetka korištenjem biblioteke LockBox 3*

```
void __fastcall TForm1::IzracunajSazetakClick(TObject *Sender) {  
    Hash1->CryptoLibrary = CryptographicLibrary1;  
    Hash1->HashId = "native.hash.SHA-256";  
    Hash1->HashString("Lozinka123", TEncoding::UTF8);  
    ShowMessage(Stream_To_Hex(Hash1->HashOutputValue));  
}
```

Ukoliko sažetak računamo korištenjem biblioteke LockBox 3 potrebno je koristiti komponentu *THash*. Pozivom metode *HashString* računa se sažetak niza znakova (Primjer 8.3.1), a osim nje moguće je koristiti i metodu *HashFile* za izračun sažetka sadržaja datoteke.

#### *Primjer 8.3.2. Izračun SHA-2 sažetka korištenjem system.hash biblioteke*

```
#include <memory>  
#include <system.hash.hpp>  
//...  
std::unique_ptr<THashSHA2> sha2(new THashSHA2);  
ShowMessage(sha2->GetHashString("Lozinka123", THashSHA2::SHA256));
```

Sažetak je moguće izračunati i pomoću već postojeće *system.hash* biblioteke. Moguće je izravno koristiti funkcije sažimanja poput MD5 i SHA-1, te SHA-2 funkcije sažimanja poput SHA-224, SHA-256, SHA-384 i SHA-512. Primjer 8.3.2 prikazuje korištenje SHA-256 funkcije sažimanja za izračun sažetka lozinke "Lozinka123".

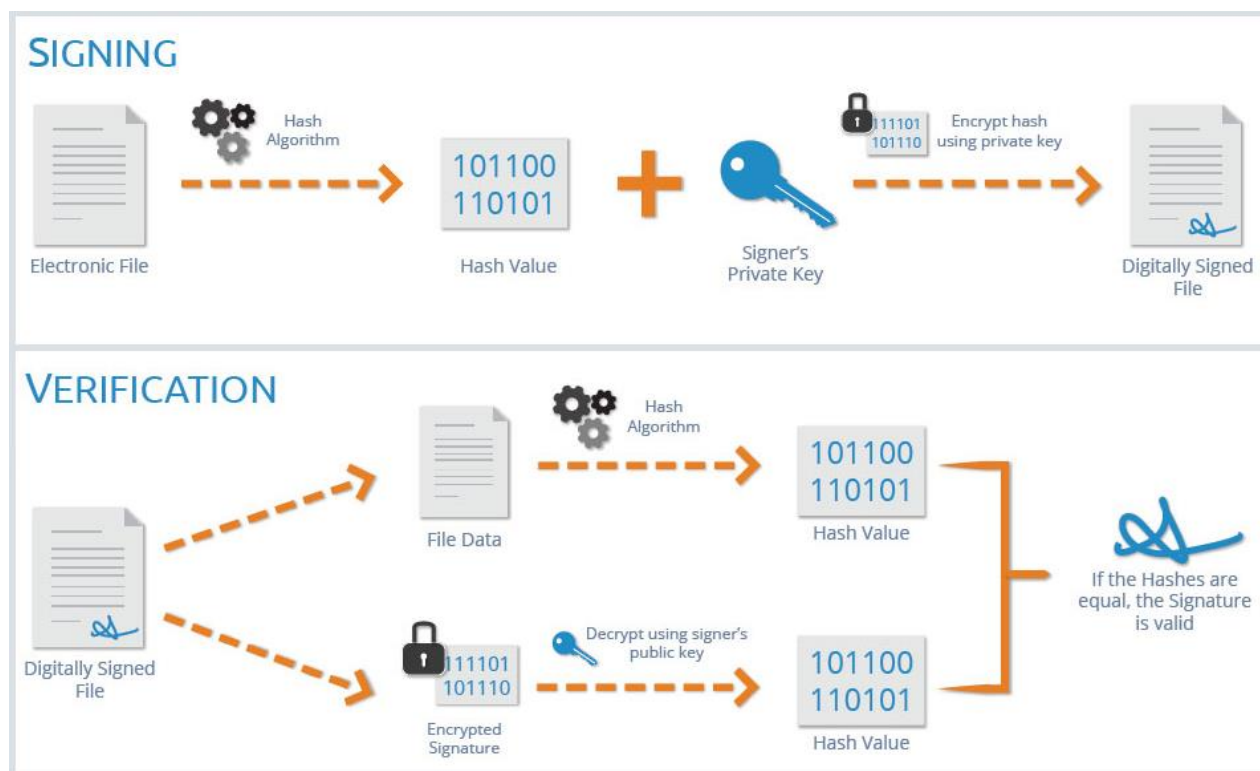
Radi poboljšanja zaštite podataka njih se dodatno može "posoliti" prije računanja sažetka. Primjerice, da bismo spriječili lako otkrivanje kratkih lozinki, uz korisnikovu lozinku možemo dodati i proizvoljni (tajni) niz znakova koji zovemo "sol". Tako bi se sažetak lozinke

"Lozinka123" korištenjem soli "DodatakNaKrajLozinkeOtvorenogTeksta" mogao izračunati kao sažetak niza "Lozinka123DodatakNaKrajLozinkeOtvorenogTeksta".

Sol ne mora nužno biti isti niz znakova, već za svaku lozinku može biti različit (primjerice, izračunat po nekom pravilu). Također, sol se ne mora nužno dodati na kraj lozinke već se može nalaziti i na njenom početku ili nekom drugom mjestu.

## 8.4. Digitalni potpis

U komunikaciji putem mreže digitalni potpis koristi se za potvrđivanje autentičnosti poruke i identifikaciju njenog pošiljatelja. Potpisivanjem poruke ne osigurava se njena tajnost (ona je i dalje svima čitljiva), već tek da nije izmijenjena tijekom prijenosa, da je doista došla od navodnog pošiljatelja te da pošiljatelj ne može poreći njenu autentičnost. [30] Digitalni potpis danas je u mnogim zemljama istovjetan i jednako važeći i kao vlastoručni potpis.



Slika 8.4.1. Postupak potpisivanja i verifikacije digitalnog potpisa [31]

Da bismo kreirali digitalni potpis poruke prvo je potrebno izračunati njen sažetak (eng. *hash*), a zatim taj sažetak šifrirati privatnim ključem. Poruka i njen digitalni potpis šalju se primatelju koji na svojoj strani također računa sažetak poruke. Izračunati sažetak primatelj će usporediti sa sažetkom koji dobije dešifriranjem primljenog digitalnog potpisa korištenjem javnog ključa pošiljatelja. Ukoliko su oba sažetka identična digitalni potpis je uspješno verificiran (Slika 8.4.1).

Postupak kreiranja i verifikacije digitalnog potpisa moguće je implementirati "ručno" kombinirajući asimetrične kriptografske algoritme i funkcije sažimanja. Međutim, biblioteka LockBox 3 pojednostavljuje ove operacije korištenjem komponente *TSignatory* i njenih metoda *Sign* i *Verify*.

#### Primjer 8.4.1. Kreiranje digitalnog potpisa dokumenta

```
void __fastcall TForm1::SignClick(TObject *Sender){
    Codec1->CryptoLibrary = CryptographicLibrary1;
    Codec1->BlockCipherId = "native.RSA";
    Codec1->AsymmetricKeySizeInBits = 1024;
    Codec1->ChainModeId = "native.CBC";
    Signatory1->Codec = Codec1;
    // učitaj dokument za potpis
    std::unique_ptr<TMemoryStream> document(new TMemoryStream);
    document->LoadFromFile("test.txt");
    // učitaj privatni ključ
    std::unique_ptr<TMemoryStream> privateKey(new TMemoryStream);
    privateKey->LoadFromFile("private_key.bin");
    Signatory1->LoadKeysFromStream(privateKey.get(),
                                   TKeyStoragePartSet() << partPrivate);
    // digitalno potpiši dokument i potpis pohrani na disk
    std::unique_ptr<TMemoryStream> signature(new TMemoryStream);
    Signatory1->Sign(document.get(), signature.get());
    signature->SaveToFile("signature.bin");
    ShowMessage("Dokument je digitalno potpisan!");
}
```

Za kreiranje digitalnog potpisa biblioteka LockBox 3 koristi algoritam RSA i njegov privatni ključ (za potrebe generiranja privatnih i javnih ključeva pogledajte Primjer 8.2.1). Digitalni potpis kreira se za podatkovni tok, bilo da je riječ o nekom podatku u memoriji ili sadržaju

datoteke na disku. Tako Primjer 8.4.1 demonstrira generiranje digitalnog potpisa za sadržaj datoteke test.txt.

Metodi *Sign* komponente *TSignatory* predaju se dva argumenta: podatkovni tok dokumenta za koji treba kreirati digitalni potpis, te podatkovni tok u koji će se pohraniti digitalni potpis. Konačno, digitalni potpis pohranjuje se u datoteku signature.bin kako bi se naknadno mogao poslati primatelju poruke.

#### Primjer 8.4.2. Verifikacija digitalnog potpisa

```
void __fastcall TForm1::VerifyClick(TObject *Sender){
    Codec1->CryptoLibrary = CryptographicLibrary1;
    Codec1->BlockCipherId = "native.RSA";
    Codec1->AsymmetricKeySizeInBits = 1024;
    Codec1->ChainModeId = "native.CBC";
    Signatory1->Codec = Codec1;
    // učitaj potpisani dokument
    std::unique_ptr<TMemoryStream> document(new TMemoryStream);
    document->LoadFromFile("test.txt");
    // učitaj javni ključ pošiljatelja
    std::unique_ptr<TMemoryStream> publicKey(new TMemoryStream);
    publicKey->LoadFromFile("public_key.bin");
    Signatory1->LoadKeysFromStream(publicKey.get(),
                                   TKeyStoragePartSet() << partPublic);
    // učitaj digitalni potpis
    std::unique_ptr<TMemoryStream> signature(new TMemoryStream);
    signature->LoadFromFile("signature.bin");
    // verificiraj digitalni potpis
    TVerifyResult vr = Signatory1->Verify(document.get(), signature.get());
    if(vr == vPass)
        ShowMessage("Digitalni potpis je uspješno verificiran!");
    else
        ShowMessage("Digitalni potpis nije valjan!");
}
```

Za verifikaciju potpisa potrebno je učitati primljenu poruku, digitalni potpis i javni ključ pošiljatelja. Pozivom metode *Verify* komponente *TSignatory* usporedit će se sažetak primljenog dokumenta s dešifriranim digitalnim potpisom, te u slučaju poklapanja digitalni potpis bit će verificiran. Ukoliko se poruka tijekom prijenosa izmijenila sažeci se neće

poklapati (digitalni potpis nije valjan), a u protivnom riječ je o izvornoj poruci potpisanoj upravo od pretpostavljenog pošiljatelja (Primjer 8.4.2).

Iako digitalni potpis jamči izvornost podataka i identificira pošiljatelja uvijek postoji opasnost da napadač krivotvori digitalni potpis. Primjerice, napadač može neovlaštenim pristupom web stranici pošiljatelja zamijeniti njegov javni ključ svojim javnim ključem. Da bi primatelj bio siguran da objavljeni javni ključ zaista pripada pretpostavljenoj osobi mogu se koristiti digitalni certifikati (X.509).

Digitalne certifikate možemo zamisliti kao "osobne iskaznice" u kojima pišu osobni podaci vlasnika te njegov javni ključ. Certifikacijske kuće poput DigiCert, Comodo, Symantec itd. djeluju kao posrednici od povjerenja koji izdaju digitalne certifikate na način da prethodno obave identifikaciju osobe koja traži certifikat te time jamče njegov identitet. Takvi certifikati obično vrijede samo neko određeno vrijeme (najčešće jednu godinu) nakon čega se moraju produžiti.

Pomoću OpenSSL biblioteke moguće je generirati i samopotpisane certifikate, ali se oni u pravilu ne smatraju vjerodostojnima, te će se njihovom upotrebom (primjerice, na web stranicama) korisnika prethodno upozoriti da je riječ o certifikatu za koji nitko ne jamči njegov sadržaj.

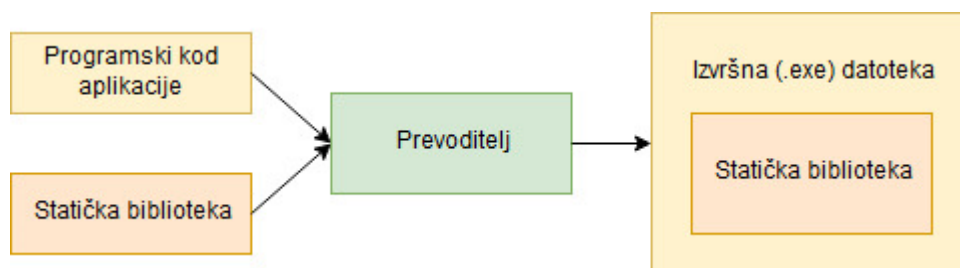


## 9. Biblioteke i komponente

### 9.1. Statičke biblioteke

Biblioteka je paket programskog koda (funkcija, klasa, resursa itd.) namijenjen za ponovno iskorištavanje unutar aplikacija. Primjerice, kada je u aplikaciji potrebno implementirati neku funkcionalnost programer može napisati vlastite funkcije i klase ili ih može preuzeti iz već postojeće biblioteke. Ovisno o tipu korištene biblioteke (statička ili dinamička) taj programski kôd može se kopirati u aplikaciju ili se nalaziti izvan nje u zasebnoj datoteci. [32]

Osim ponovne iskoristivosti, biblioteke omogućuju i skrivanje programskog kôda. Programer koji želi koristiti neku biblioteku vidi tek njenu strukturu (deklaracije klasa, funkcija, metoda itd.), dok je implementacija skrivena. Naime, struktura je vidljiva iz datoteka zaglavlja (.h) koje se isporučuju uz biblioteku, dok je implementacija skrivena u obliku strojnog kôda.



Slika 9.1.1. Korištenje statičke biblioteke

Statičke biblioteke imaju ekstenziju **.lib** u OS Windows te ekstenziju **.a** (archive) u OS Linux. Potrebne se prilikom prevođenja aplikacije, a tada se programski kôd sadržan u statičkoj biblioteci kopira u samu aplikaciju (Slika 9.1.1). Time se povećava veličina izvršne datoteke, ali zato ona za svoj daljnji rad ne zahtjeva prisutnost.lib datoteke.

Da bismo demonstrirali izradu i upotrebu statičke biblioteke potrebno je kreirati dva projekta: projekt statičke biblioteke i projekt aplikacije koja će koristiti statičku biblioteku (primjerice, aplikacija komandne linije).

**Primjer 9.1.1. Sadržaj statičke biblioteke**

```
// MojeFunkcije.h - deklaracije
double Suma(double a, double b);

// MojeFunkcije.cpp - implementacija
double Suma(double a, double b) {
    return a + b;
}
```

U projektu statičke biblioteke potrebno je razdvojiti deklaraciju i implementaciju, odnosno koristiti najmanje dvije datoteke. U datotekama zaglavlja (.h) trebaju se nalaziti deklaracije, a u datotekama.cpp implementacija. Tako Primjer 9.1.1 prikazuje sadržaj projekta statičke biblioteke u kojoj je smještena funkcija *Suma*.

Prilikom prevođenja projekta generirat će se statička biblioteka (datoteka .lib). Ona sada predstavlja ekvivalent svih implementacijskih (.cpp) datoteka, ali je prevedena u strojni jezik te tako skriva implementacijski programski kôd. Aplikacija koja želi implementirati funkcionalnosti te statičke biblioteke treba osim nje preuzeti i sve njene datoteke zaglavlja. Bez njih programer neće znati koje su sve funkcionalnosti dostupne u toj biblioteci.

Da bismo u nekoj aplikaciji mogli koristiti statičku biblioteku prvo je poželjno kopirati sve potrebne datoteke (.lib, \*.h) u mapu projekta aplikacije. Zatim, odabirom stavke glavnog izbornika *Project / Add to Project...* potrebno je dodati datoteku.lib u projekt aplikacije.

**Primjer 9.1.2. Poziv funkcije iz statičke biblioteke**

```
#include "MojeFunkcije.h"
// ...
Suma(2.5, 3.1);
```

Konačno, pretprocesorskom naredbom *include* u aplikaciji je potrebno uključiti datoteke zaglavlja isporučene sa statičkom bibliotekom, te je zatim moguće koristiti njen programski kôd (Primjer 9.1.2).

Statičke biblioteke jednostavno je izrađivati i koristiti, ali ipak imaju određene nedostatke. Primjerice, statičku biblioteku izrađenu u C++ Builderu nije izravno moguće koristiti u Visual Studio aplikacijama. Također vrijedi i obratno jer njihovi prevoditelji generiraju statičke

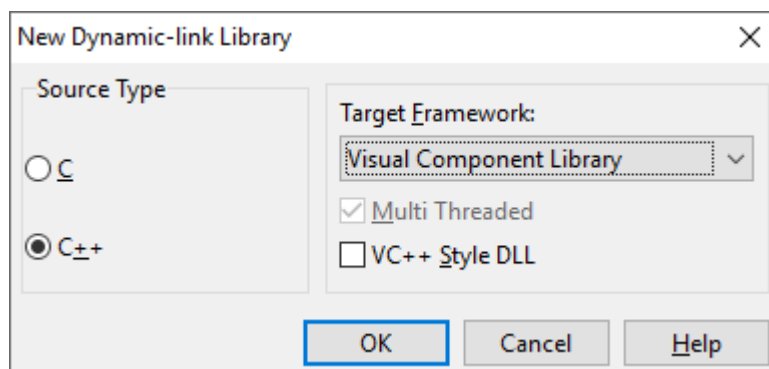


biblioteke u različitim formatima. U slučaju nadogradnje statičke biblioteke ponovno se mora prevoditi i aplikacija koja ju koristi, a ponekad će biti komplikacija i ako se statička biblioteka prevedena u starijoj inačici prevoditelja pokuša koristiti u aplikaciji koja koristi noviju inačicu tog prevoditelja.

## 9.2. Dinamičke biblioteke

Programski kôd sadržan u statičkoj biblioteci kopira se u aplikaciju. Međutim, isti programski kôd moguće je smjestiti i u dinamičku biblioteku, odnosno DLL (*Dynamic Link Library*) datoteku. Sadržaj dinamičke biblioteke ne kopira se u aplikaciju, već aplikacija izravno iz DLL-a koristi potreban programski kôd. Zbog toga je za ispravan rad aplikacije uz izvršnu (.exe) datoteku potrebno distribuirati i sve potrebne DLL datoteke.

Dinamičke biblioteke omogućuju i jednostavnije ažuriranje aplikacija. Naime, da bismo ažurirali neku funkcionalnost aplikacije potrebno je tek zamijeniti DLL datoteku (biblioteku) u kojoj je ta funkcionalnost implementirana. Time se eliminira potrebna za ponovnim instaliranjem kompletne aplikacije, te je ona brže spremna za daljnji rad.



Slika 9.2.1. Kreiranje nove dinamičke biblioteke

Nova dinamička biblioteka kreira se odabirom stavke *File / New / Other... / C++ Builder / Dynamic Library*, nakon čega se pojavljuje prikazani dijalog (Slika 9.2.1). Ukoliko je odabran C++ tip izvornog kôda dodatno je moguće odabrati i programski okvir *Visual Component Library* ili *FireMonkey* koji omogućuju korištenje VCL i FMX komponenti unutar DLL biblioteke.

Nakon kreiranja novog DLL projekta, u podrazumijevanoj datoteci *File1.cpp* nalazit će se funkcija *DllEntryPoint*. Ona nije obavezna, no može se koristiti ukoliko je DLL potrebno inicijalizirati prilikom početka korištenja te "počistiti" nakon što više nije potreban. Općenito se može implementirati na sljedeći način (Primjer 9.2.1).

#### Primjer 9.2.1. Funkcija *DllEntryPoint*

```
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*
lpReserved) {
    switch (reason) {
        case DLL_PROCESS_ATTACH: /* Proces je učitao DLL */
        case DLL_THREAD_ATTACH:  /* Dretva je učitala DLL */
        case DLL_PROCESS_DETACH: /* Proces se odspojio od DLL-a */
        case DLL_THREAD_DETACH:  /* Dretva se odspojila od DLL-a */
    }
    return 1;
}
```

Da bismo demonstrirali korištenje DLL-a prvo je u njega potrebno smjestiti željene funkcije i klase. Međutim, kao i u slučaju sa statičkom bibliotekom, tako je i sada potrebno razdvojiti deklaraciju od implementacije korištenjem datoteka *h* i *.cpp*.





#### Primjer 9.2.2. Sadržaj datoteke *dll\_exports.h*

```
#include <vcl.h> // MessageBox
#ifdef EXPORTS
    #define DLL_EXPORT __declspec(dllexport)
#else
    #define DLL_EXPORT __declspec(dllimport)
#endif
extern "C" {
    // globalna funkcija
    int DLL_EXPORT __stdcall Suma(int a, int b);
    // klasa DllClass i njene metode
    class DLL_EXPORT DllClass{
    public:
        void f1();
        void f2();
    };
}
```

Neka je u projekt DLL dodana datoteka zaglavlja *dll\_exports.h* koja sadrži deklaraciju globalne funkcije *Suma* te deklaraciju klase *DllClass* s metodama *f1* i *f2* (Primjer 9.2.2). Njihove deklaracije imaju dodatne specifikatore (*extern*, *\_\_stdcall* itd.) koje obično ne koristimo pri radu s funkcijama i klasama, pa ih sada možemo objasniti na primjeru deklaracije funkcije *Suma*.

```
extern "C" int DLL_EXPORT __stdcall Suma(int a, int b);
```

Kada se u DLL pohranjuje eksportabilna funkcija (ona koju mogu koristiti druge aplikacije) pohranjuje se njeno ime, tijelo i opis (povratna vrijednost, parametri itd.). Zbog toga se ime funkcije *Suma* u DLL-u inicijalno vidi kao niz znakova *@Suma\$qqsii*. Izrazom *extern "C"* ime DLL funkcije se dekorira tako da ne sadrži opis već tek svoje ime. Zato će se korištenjem tog izraza navedena funkcija vidjeti pod imenom *\_Suma*.

E	Ordinal ^	Hint	Function	Entry Point
	9 (0x0009)	13 (0x000D)	Suma	0x000025B8
	10 (0x000A)	11 (0x000B)	@DllClass@f1\$qv	0x000025E0
	11 (0x000B)	12 (0x000C)	@DllClass@f2\$qv	0x00002628
	12 (0x000C)	3 (0x0003)	@@Dll_exports@Initialize	0x00002CA4

Slika 9.2.2. Dependency Walker - eksportabilna funkcija *Suma*

Da bismo uklonili početni znak '\_' iz imena DLL funkcije koristimo standardnu konvenciju poziva *\_\_stdcall*, pa se navedena funkcija sada vidi pod imenom *Suma* (Slika 9.2.2). Ovakav oblik pohrane imena važan je ukoliko DLL funkciju planiramo dohvatiti korištenjem funkcije *GetProcAddress* koja kao jedan od argumenata upravo traži naziv DLL funkcije.

Identifikator *DLL\_EXPORT* može predstavljati specifikaciju *\_\_declspec(dllexport)* ili *\_\_declspec(dllimport)*. Naime, u dinamičkim bibliotekama potrebno je korištenjem specifikacije *\_\_declspec(dllexport)* označiti sve one funkcije i klase koje trebaju biti dostupne aplikacijama. [33] Međutim, aplikacije bi zbog optimizacije trebale koristiti specifikaciju *\_\_declspec(dllimport)*. Tako bi se funkcija *Suma* u DLL projektu trebala prevesti kao

```
extern "C" int __declspec(dllexport) __stdcall Suma(int a, int b);
```

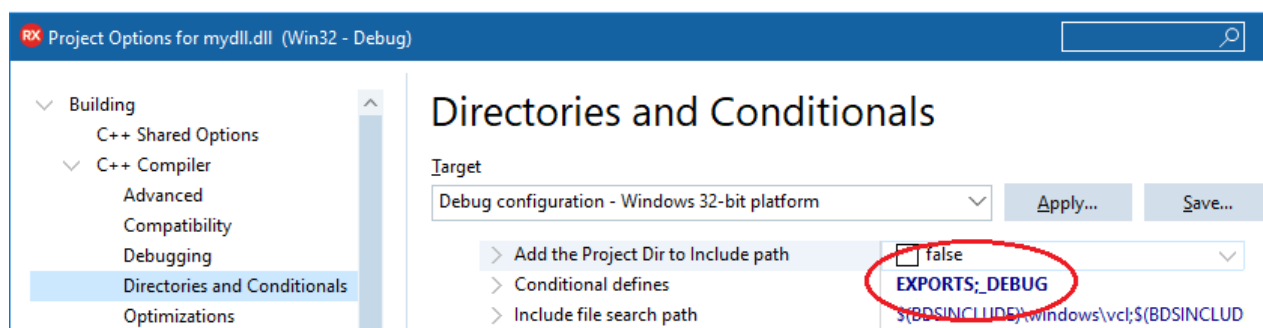
dok bi aplikacije koje žele koristiti ovu DLL funkciju trebale koristiti deklaraciju

```
extern "C" int __declspec(dllimport) __stdcall Suma(int a, int b);
```

Umjesto da kreiramo dvije datoteke zaglavlja (jednu za prevođenje DLL projekta, a drugu za prevođenje aplikacije koja će koristiti DLL) korištene su sljedeće makro naredbe.

```
#ifdef EXPORTS
    #define DLL_EXPORT __declspec(dllexport)
#else
    #define DLL_EXPORT __declspec(dllimport)
#endif
```

Zatim, u postavkama DLL projekta (*Project / Options... / C++ Compiler / Directories and Conditionals*) potrebno je definirati identifikator *EXPORTS* (Slika 9.2.3).



Slika 9.2.3. Definiranje izraza *EXPORTS*

Sada će se pri prevođenju DLL projekta umjesto identifikatora *DLL\_EXPORT* koristiti specifikacija *\_\_declspec(dllexport)*, pa će označene funkcije i klase biti dostupne aplikacijama. Međutim, kada aplikacije koje žele koristiti taj programski kôd uključe datoteku zaglavlja (*dll\_exports.h*) identifikator *DLL\_EXPORT* će automatski biti zamijenjen specifikacijom *\_\_declspec(dllimport)* jer identifikator *EXPORTS* u tim aplikacijama podrazumijevano ne postoji.

Svaku globalnu funkciju za koju želimo da je dostupna aplikacijama potrebno je označiti identifikatorom *DLL\_EXPORT* tj. specifikacijom *\_\_declspec(dllexport)*. Ipak, kada je riječ o klasama moguće je odjednom označiti sve njihove metode izrazom poput *class DLL\_EXPORT DllClass.*

**Primjer 9.2.3. Sadržaj datoteke `dll_exports.cpp`**

```
#pragma hdrstop
#pragma package(smart_init)
#include "dll_exports.h"

// globalna funkcija Suma
int Suma(int a, int b) {
    return a + b;
}

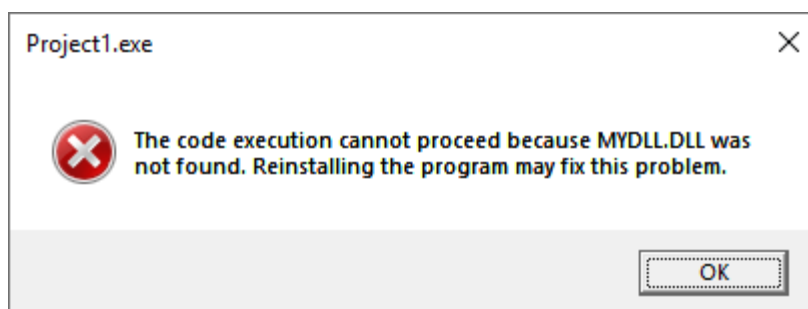
// Funkcije članice klase DllClass
void DllClass::f1() {
    MessageBoxW(0, L"DllClass::f1()", L"DLL", 0);
}
void DllClass::f2() {
    MessageBoxW(0, L"DllClass::f2()", L"DLL", 0);
}
```

U implementacijskoj datoteci (Primjer 9.2.3) nije potrebno navoditi specifikatore *extern "C"*, *\_\_stdcall*, *\_\_declspec(dllexport)* i *declspec(dllimport)*. Na osnovu sadržaja datoteke zaglavlja (Primjer 9.2.2) prevoditelj već zna na koji način treba imenovati funkcije i metode, te koje od njih će dati na korištenje aplikacijama.

Konačno, prevođenjem DLL projekta kao rezultat dobivamo .DLL i .LIB datoteke. U .DLL datoteci nalazi se implementacija u strojnom jeziku, dok .LIB datoteka koristi se u svrhu statičkog DLL povezivanja s aplikacijom. Osim ove dvije datoteke, aplikacijama je potrebno dostaviti i datoteke zaglavlja u kojima se nalaze deklaracije DLL funkcija i klasa (npr. *dll\_exports.h*, Primjer 9.2.2).

## 9.3. Statičko i dinamičko DLL povezivanje

U C++ Builderu moguće je na dva načina povezati aplikaciju i DLL biblioteku tj. korištenjem statičkog i dinamičkog povezivanja. Statičko DLL povezivanje (eng. *static dll linking*) podrazumijevano se događa prilikom pokretanja aplikacije. DLL se učitava u virtualni adresni prostor procesa, nakon čega je moguće pozivati dostupne DLL funkcije. [34] Ukoliko proces (aplikacija) prilikom pokretanja ne može pronaći potrebnu DLL datoteku nasilno će prekinuti s radom i prikazati odgovarajuću poruku (npr. Slika 9.3.1).



Slika 9.3.1. Greška pri pokušaju statičkog DLL povezivanja

Statičko DLL povezivanje implementira se na sljedeći način:

1. U projekt aplikacije potrebno je dodati statičku (LIB) biblioteku koja se isporučuje s DLL bibliotekom (*Project / Add to Project...*).
2. U izvornom kôdu aplikacije potrebno je uključiti (`#include`) sve priložene datoteke zaglavlja DLL biblioteke.
3. DLL datoteka treba biti dostupna aplikaciji (npr. smjestiti ju u istu mapu gdje se nalazi i izvršna EXE datoteka).

Prethodno smo statičku LIB biblioteku koristili kao skladište programskog kôda koje se kopira u aplikaciju. Ipak, LIB biblioteka koja dolazi s DLL-om ima sasvim drugu svrhu. Ona sada djeluje kao posrednik tj. sadrži sve informacije koje su aplikaciji potrebne da bi poveziivač (eng. *linker*) automatski mogao pronaći tražene DLL funkcije. Zato je statičko DLL povezivanje mnogo jednostavnije za implementirati jer programer ne mora "ručno" tražiti adresu svake DLL funkcije.

#### Primjer 9.3.1. Pozivi DLL funkcija

```
#include "dll_exports.h"
// ...
void __fastcall TForm1::Button1Click(TObject *Sender){
    ShowMessage(Suma(3, 5)); // poziv globalne DLL funkcija Suma
    DllClass obj; // objekt DLL klase
    obj.f1(); // poziv metode f1
    obj.f2(); // poziv metode f2
}
```

Nakon što su odrađeni svi prethodno opisani koraci DLL funkcije, klase i njihove metode koristimo kao da su implementirani unutar same aplikacije (Primjer 9.3.1).

Iako je statičko DLL povezivanje jednostavnije za implementirati, nedostatak mu je veća potrošnja radne memorije (DLL je cijelo vrijeme učitano u virtualni adresni prostor procesa), te nemogućnost korištenja aplikacije ukoliko sve potrebne DLL datoteke nisu prisutne. Djelomično je ove probleme moguće riješiti korištenjem stavke *Delay load DLLs (Project / Options... / C++ Linker / Advanced)*, pri čemu se odabrani DLL-ovi neće učitati u aplikaciju sve dok se ne pozove jedna od njihovih funkcija. Alternativno, moguće je koristiti dinamičko DLL povezivanje (eng. *runtime dll linking*). DLL se tada učitava samo kada je potreban (npr. pri pozivu DLL funkcije), te se odmah nakon toga može osloboditi.

### Primjer 9.3.2. Dinamičko DLL povezivanje

---

```
void __fastcall TForm1::Button1Click(TObject *Sender) {
    HINSTANCE MojDll;

    typedef int (*__stdcall pSuma)(int, int); // pokazivač na funkciju Suma
    pSuma Suma;

    // dinamičko DLL povezivanje
    if((MojDll = LoadLibrary(L"MyDll.dll")) == NULL) {
        Application->MessageBox(L"DLL nije moguće učitati!", L"Greška", 0);
        return;
    }

    // Pokazivač na funkciju Suma adresira se na DLL funkciju
    if((Suma = (pSuma)GetProcAddress(MojDll, "Suma")) == NULL) {
        Application->MessageBox(L"U DLL-u nema tražene funkcije!",
                                L"Greška", 0);

        return;
    }

    // Pozovi funkciju Suma
    ShowMessage(Suma(3, 4)); // 7
    FreeLibrary(MojDll);
}
```

Kod implementacije dinamičkog DLL povezivanja ne koristi se statička LIB biblioteka. Štoviše, nije nužno koristiti niti datoteke zaglavlja DLL-a, već one sada primarno služe tek kao informacija programeru o dostupnim DLL funkcijama.

DLL biblioteka učitava se pozivom funkcije *LoadLibrary*. Zatim, korištenjem funkcije *GetProcAddress* potrebno je pronaći adresu željene DLL funkcije te ju pridružiti

odgovarajućem funkcijskom pokazivaču. Ovaj korak je nužan zbog nekorištenja statičke LIB biblioteke, pa aplikacija sada za svaku DLL funkciju mora "ručno" tražiti njenu adresu. Konačno, nakon poziva DLL funkcije biblioteku se može osloboditi funkcijom *FreeLibrary* (Primjer 9.3.2).

Jedna od prednosti DLL biblioteka jest da se mogu koristiti u različitim programskim jezicima. Tako bi DLL funkciju *Suma* mogli pozvati i unutar C# .NET aplikacije korištenjem sljedećeg programskog odsječka (Primjer 9.3.3).

#### Primjer 9.3.3. Poziv DLL funkcije u C# .NET aplikaciji

```
using System.Runtime.InteropServices;

namespace WindowsFormsApp1 {
    public partial class Form1 : Form {
        [DllImport("MyDll.dll")]
        static extern int Suma(int a, int b);

        // ...
        private void Button1_Click(object sender, EventArgs e) {
            MessageBox.Show(Suma(3, 5).ToString()); // 8
        }
    }
}
```

U DLL biblioteke moguće je smjestiti i resurse. To mogu biti razni tipovi slika (bmp, jpg, jpeg, png itd.), ikone, fontovi ili općenito datoteke bilo kojeg formata. Svaki resurs ima svoj identifikator i tip (ICON, BITMAP, CURSOR, FONT, RCDATA – ostali tipovi datoteka), a DLL biblioteci mogu se dodati korištenjem stavke *Project / Resources and Images...*

Jedna od resursnih DLL biblioteka dolazila je s operacijskim sustavom Windows XP. Biblioteka cards.dll sadrži karte u BMP formatu koje su koristile kartaške igre poput Solitaire, FreeCell, Spider itd. Te karte (slike) moguće je učitati i u vlastitoj aplikaciji na sljedeći način (Primjer 9.3.4).

#### Primjer 9.3.4. Učitavanje karti iz DLL biblioteke

```
void __fastcall TForm1::Button1Click(TObject *Sender) {
    HINSTANCE KarteDll;

    if ((KarteDll = LoadLibrary(L"cards.dll")) == NULL) {
        ShowMessage("Ne mogu učitati DLL!");
        return;
    }
}
```

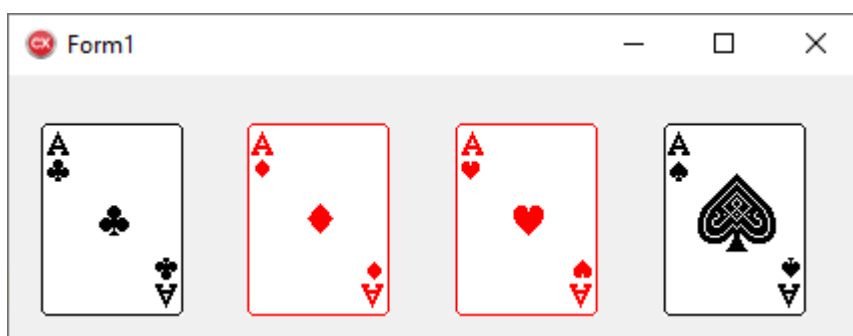


```

Image1->Picture->Bitmap->LoadFromResourceID((int)KarteDll, 1);
Image2->Picture->Bitmap->LoadFromResourceID((int)KarteDll, 14);
Image3->Picture->Bitmap->LoadFromResourceID((int)KarteDll, 27);
Image4->Picture->Bitmap->LoadFromResourceID((int)KarteDll, 40);
FreeLibrary(KarteDll);
}

```

Ukoliko pretpostavimo da aplikacija na svom dijalogu sadrži četiri *TImage* komponente (*Image1*, *Image2*, *Image3* i *Image4*), izvršavanjem prethodnog programskog odsječka pojaviti će se sljedeće.



Slika 9.3.2. Karte iz DLL biblioteke

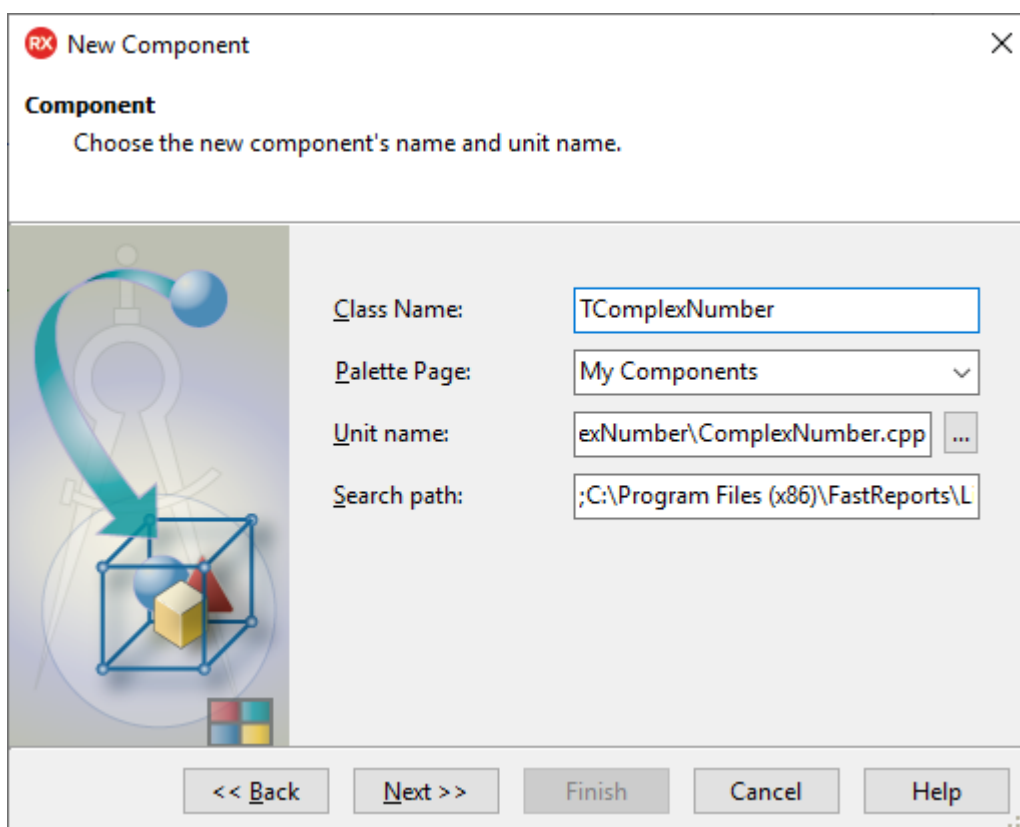
U cards.dll biblioteci karte su spremljene kao BMP resursi korištenjem identifikatora 1-52. Tako identifikatori 1, 14, 27 i 40 predstavljaju asove (Slika 9.3.2). Sadržaj resursnih DLL biblioteke moguće je pregledati korištenjem aplikacija poput Resource Hacker, XN Resource Editor, Restorator itd.

## 9.4. Razvoj VCL komponenti

Komponentu možemo zamisliti kao softverski paket ili modul koji u sebi sadrži programski kôd i podatke. Veliki broj aplikacija najčešće je kompilacija jedne ili više tipova komponenti, pri čemu one mogu biti vizualne (npr. gumbi, liste, dijalozi itd.) i nevizualne (npr. klase i servisi). Korištenjem komponenti smanjuje se složenost razvoja, troškovi održavanja i podrške, te se omogućuje ponovno korištenje istog programskog kôda na mnogim mjestima. [35]

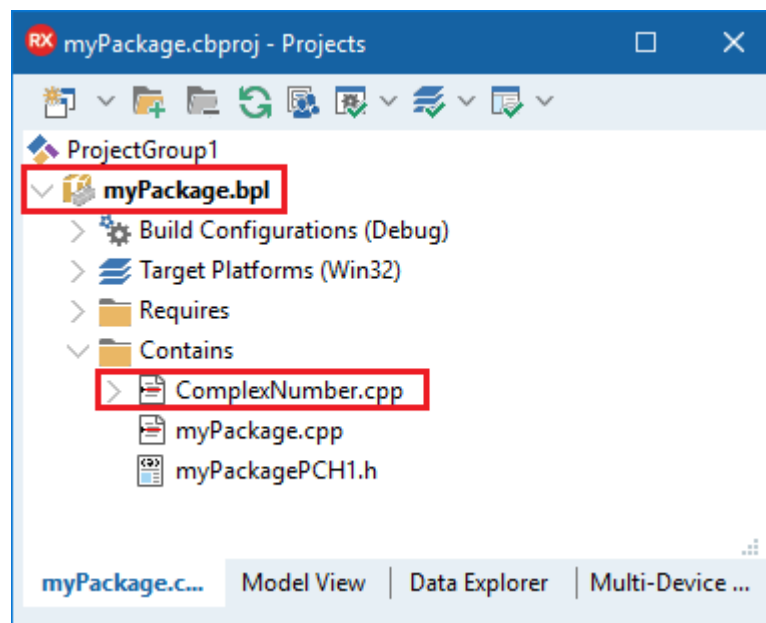
U svrhu demonstracije prikažimo razvoj nevizualne VCL komponente *TComplexNumber* koja će predstavljati kompleksni broj. Prije razvoja same komponente poželjno je napraviti paket (eng. *package*). To je posebna vrsta DLL biblioteke s ekstenzijom BPL. Ona se može koristiti kao i svaka druga DLL biblioteka (*run-time package*), ali i kao skladište za jednu ili više komponenti koje treba instalirati u RAD Studio (*design-time package*).

Za kreiranje paketa potrebno je odabrati stavku *File / New / Other...*, nakon čega je u prikazanom dijalogu pod dijelom *C++ Builder* potrebno odabrati stavku *Package*. U paket je sada moguće dodati novu VCL komponentu odabirom stavke *Component / New Component...*, a zatim odabrati stavku *VCL for C++ Win32*. Kao komponentu pretka (eng. *Ancestor Component*) potrebno je odabrati *TComponent* te kliknuti gumb *Next*.



Slika 9.4.1. Nova VCL komponenta

Slika 9.4.1 prikazuje sljedeći korak u kojemu je potrebno odrediti ime nove komponente (*TComplexNumber*), paletu u kojoj će biti smještena (*My Components*), te naziv datoteke u kojoj će se nalaziti njena implementacija. U narednom dijalogu ovu komponentu moguće je dodati prethodno kreiranom paketu, nakon čega se može pristupiti pisanju implementacije.



Slika 9.4.2. Paket i komponenta

Ukoliko je postupak izrade paketa i komponente uspješno odrađen sadržaj paketa bit će prikazan kao na prethodnoj slici (Slika 9.4.2).

#### Primjer 9.4.1. Sadržaj datoteke *ComplexNumber.h*

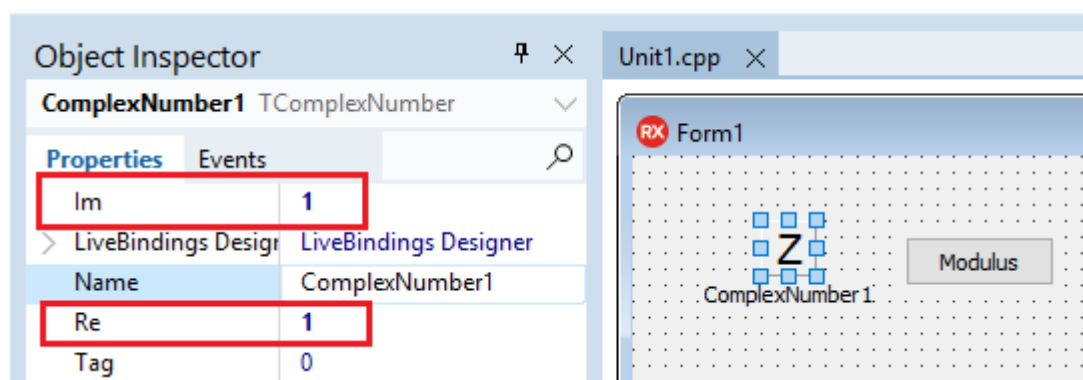
```
class PACKAGE TComplexNumber : public TComponent {
private:
    double pRe, pIm;
public:
    double Modulus();
    __fastcall TKompleksni(TComponent* Owner);
__published:
    // Svojstva Re i Im obrađuju se podatkovnim članovima pRe i pIm
    __property double Re = {write = pRe, read = pRe};
    __property double Im = {write = pIm, read = pIm};
};
```

Primjer 9.4.1 prikazuje sadržaj datoteke zaglavlja u kojoj se nalazi deklaracija klase (komponente) *TComplexNumber*. Ona sadrži svojstva *Re* i *Im* koja će biti prikazana u objektom inspektor, a prilikom njihove obrade (čitanja i pisanja) zapravo će se obrađivati privatni podatkovni članovi *pRe* i *pIm*. Klasa također sadrži i metodu *Modulus* koja vraća modul kompleksnog broja.

**Primjer 9.4.2. Implementacija metode Modulus**

```
double TComplexNumber::Modulus() {
    return pow(pRe * pRe + pIm * pIm, 0.5);
}
```

U implementacijskoj datoteci *ComplexNumber.cpp* potrebno je tek dopisati tijelo metode *Modulus* (Primjer 9.4.2). Zatim, desnim klikom na ime paketa (*myPackage.bpl*, Slika 9.4.2) i odabirom stavke *Install* komponenta *TComplexNumber* bit će instalirana i smještena u paletu s alatima.



Slika 9.4.3. Komponenta *TComplexNumber*

Komponenta *TComplexNumber* vidljiva je samo u vrijeme dizajna, dok su klikom na nju u objektnom inspektoru vidljiva svojstva *Im* i *Re* (Slika 9.4.3). Dodatno, moguće je pozvati metodu *Modulus* na sljedeći način:

```
ComplexNumber1->Modulus(); // 1.414
```

Da bi postavili ikonu komponente potrebno je u projektu (paketu koji sadrži komponentu) klikom na *Project / Resources and Images...* dodati novi BITMAP resurs. To mora biti slika u BMP formatu dimenzija 24x24, čiji identifikator mora biti isti kao i ime komponente (*TComplexNumber*).

Iako je razvoj komponenti VCL moguć u C++ Builderu, za njihov razvoj najčešće se koristi Delphi. Razlog je u tome što C++ Builder podržava Delphi programski kôd, pa komponente razvijene za Delphi mogu se instalirati i koristiti i u C++ Builderu. Alternativno, VCL komponente razvijene u C++ Builderu moguće je koristiti samo u C++ Builderu.

**Primjer 9.4.3. Razvoj komponente *TComplexNumber* u Delphiju**

```
unit ComplexNumber;
interface
uses
    System.SysUtils, System.Classes;
type
    TComplexNumber = class(TComponent)
    private
        pRe : double;
        pIm : double;
    public
        function Modulus() : double;
    published
        property Re : double read pRe write pRe;
        property Im : double read pIm write pIm;
    end;
procedure Register;

implementation
function TComplexNumber.Modulus() : double;
begin
    Result := Sqrt(pRe * pRe + pIm * pIm);
end;
procedure Register;
begin
    RegisterComponents('My Components', [TComplexNumber]);
end;
end.
```

Primjer 9.4.3 demonstrira kako razviti komponentu *TComplexNumber* korištenjem Delphija. Prevođenjem Delphi paketa u kojemu se nalazi ova komponenta generirat će se sve potrebne datoteke i za C++ Builder, pa će i u njemu biti moguće koristiti navedenu komponentu.



## Popis slika

Slika 1.1.1. GetIt Package Manager .....	9
Slika 1.2.1. Podržani operacijski sustavi i platforme [1] .....	10
Slika 1.3.1. Kreiranje novog projekta u C++ Builderu.....	12
Slika 1.3.2. Kreiranje novog projekta u C++ Builderu (glavna paleta) .....	12
Slika 1.3.3. Postavke nove aplikacije komandne linije .....	13
Slika 1.3.4. Pregled projektnih datoteka .....	14
Slika 1.4.1. Uređivač programskog kôda u C++ Builderu.....	15
Slika 1.4.2. IntelliSense značajka .....	15
Slika 1.4.3. Opis parametara funkcije .....	16
Slika 1.5.1. Korištenje aplikacije za otkrivanje grešaka .....	17
Slika 2.1.1. Nova Windows VCL aplikacija .....	19
Slika 2.2.1. Korištenje funkcije ShowMessage.....	22
Slika 2.2.2. Korištenje funkcije MessageBox (prikaz informacije) .....	23
Slika 2.2.3. Korištenje funkcije MessageBox (upit korisniku) .....	24
Slika 2.2.4. Korištenje funkcije MessageDlg .....	25
Slika 2.2.5. Korištenje TTaskDialog komponente .....	26
Slika 2.3.1. Dijalog InputBox.....	28
Slika 2.3.2. Korištenje komponente TTaskDialog pri odabiru podataka .....	30
Slika 2.5.1. Dijalog, gumbi i potvrdni okviri .....	32
Slika 2.5.2. Objektni inspektor - praćenje događaja .....	33
Slika 2.5.3. Polja za unos znakova i liste .....	34
Slika 2.5.4. Windows dijalozi .....	36
Slika 2.6.1. Unos podatka u drugom dijalogu.....	38
Slika 2.7.1. Nova standardna akcija .....	40
Slika 2.7.2. Akcijska lista .....	41
Slika 2.8.1. Dijalog za novog pretplatnika .....	42
Slika 2.8.2. Alat Resource DLL Wizard.....	42
Slika 2.8.3. Resursni DLL projekti.....	43
Slika 2.8.4. Alat Translation Editor.....	43
Slika 3.2.1. Registry Editor .....	49
Slika 3.2.2. Postavke testne aplikacije pohranjene u Windows registru .....	50
Slika 3.3.1. Aplikacija za uređivanje adresara.....	53
Slika 3.3.2. XML Data Binding Wizard alat - struktura XML datoteke.....	54

Slika 3.5.1. Struktura novog formata za pohranu kontakata.....	64
Slika 4.1.1. Komunikacija s bazom podataka.....	70
Slika 4.1.2. Data Controls komponente .....	70
Slika 4.2.1. ADO komponente .....	71
Slika 4.2.2. SQL Server baza podataka "Studij" .....	72
Slika 4.2.3. Podatkovni modul .....	72
Slika 4.2.4. Svojstvo „ConnectionString“.....	73
Slika 4.3.1. Obrada podataka iz tablice "Student" .....	74
Slika 4.4.1. Povezivanje primarnog i stranog ključa .....	80
Slika 4.4.2. Unos novog zapisa u tablicu djeteta.....	81
Slika 4.5.1. Kreiranje novog izračunatog polja .....	82
Slika 4.5.2. Kreiranje novog polja tipa lookup .....	83
Slika 4.6.1. Fast Report komponente .....	85
Slika 4.6.2. Dizajner izvještaja .....	85
Slika 4.6.3. Pregled izvještaja.....	86
Slika 5.1.1. Proces sa dvije dretve.....	89
Slika 5.1.2. Broj procesa i dretvi u operacijskom sustavu .....	89
Slika 5.1.3. Izrada nove dretve .....	91
Slika 5.2.1. Sinkronizacija dretve metodom Synchronize [16].....	96
Slika 6.1.1. Model klijent-poslužitelj .....	109
Slika 6.2.1. Indy TCP klijent i poslužitelj .....	110
Slika 6.2.2. Svojstvo Bindings .....	112
Slika 6.3.1. UDP aplikacija za razmjenu poruka .....	115
Slika 6.4.1. Preuzimanje datoteke korištenjem Indy HTTP klijenta .....	118
Slika 6.5.1. Indy ICMP klijent.....	120
Slika 6.6.1. Popis Windows servisa .....	122
Slika 6.6.2. Poslužitelj komponente u Windows servisu (TService) .....	123
Slika 6.6.3. Moj Servis 1.0 .....	124
Slika 7.1.1. WSDL uvoznik .....	125
Slika 7.2.1. Novi SOAP poslužitelj – tip aplikacije .....	127
Slika 7.2.2. Kreiranje novog servisa unutar SOAP poslužitelja .....	128
Slika 7.3.1. REST klijent komponente.....	131
Slika 7.4.1. TWebModule akcije .....	133
Slika 7.4.2. Windows VCL aplikacija za upravljanje web poslužiteljem .....	134



Slika 7.4.3. Obrada podrazumijevane akcije .....	134
Slika 7.4.4. Testiranje akcije actSuma .....	135
Slika 7.4.5. Testiranje akcije actPrim .....	137
Slika 7.5.1. Instaliranje podrške za ISAPI ekstenzije .....	141
Slika 7.5.2. Dozvola za izvršavanje ISAPI-dll aplikacija .....	141
Slika 7.5.3. ISAPI i CGI ograničenja .....	142
Slika 7.5.4. Izvršavanje ISAPI aplikacije u web pregledniku .....	143
Slika 7.5.5. URL Rewrite - kreiranje novog pravila za preusmjerenje .....	143
Slika 8.1.1. Princip rada simetričnog kriptografskog algoritma [25] .....	145
Slika 8.1.2. TurboPower LockBox komponente .....	146
Slika 8.1.3. Usporedba ECB-a i ostalih načina šifriranja .....	147
Slika 8.2.1. Princip rada asimetričnog kriptografskog algoritma [27] .....	149
Slika 8.3.1. Provjera ispravnosti lozinke korištenjem funkcije sažimanja.....	152
Slika 8.4.1. Postupak potpisivanja i verifikacije digitalnog potpisa [30] .....	154
Slika 9.1.1. Korištenje statičke biblioteke .....	159
Slika 9.2.1. Kreiranje nove dinamičke biblioteke .....	161
Slika 9.2.2. Dependency Walker - eksportabilna funkcija Suma .....	163
Slika 9.2.3. Definiranje izraza EXPORTS .....	164
Slika 9.3.1. Greška pri pokušaju statičkog DLL povezivanja .....	166
Slika 9.3.2. Karte iz DLL biblioteke .....	169
Slika 9.4.1. Nova VCL komponenta .....	170
Slika 9.4.2. Paket i komponenta .....	171
Slika 9.4.3. Komponenta TComplexNumber.....	172



## Popis primjera

Primjer 2.1.1. Sadržaj DFM datoteke .....	20
Primjer 2.2.1. Korištenje funkcije MessageBox (upit korisniku) .....	24
Primjer 2.2.2. Konfiguriranje i korištenje TTaskDialog komponente .....	25
Primjer 2.2.3. TTaskDialog - Dodavanje gumba s proizvoljnim naslovom .....	26
Primjer 2.3.1. Korištenje funkcije InputBox .....	27
Primjer 2.3.2. Korištenje funkcije InputQuery .....	29
Primjer 2.3.3. Korištenje TTaskDialog komponente pri odabiru podataka .....	29
Primjer 2.5.1. Sadržaj datoteke zaglavlja dijaloga .....	32
Primjer 2.5.2. Gumbi i potvrdni okviri .....	34
Primjer 2.5.3. Korištenje Windows dijaloga .....	36
Primjer 2.5.4. Dinamička alokacija VCL komponenti .....	37
Primjer 2.6.1. Sadržaj datoteke Unit1.cpp .....	39
Primjer 2.6.2. Dinamička alokacija dijaloga .....	39
Primjer 2.8.1. Promjena jezika sučelja klikom na gumb .....	45
Primjer 3.1.1. Sadržaj datoteke POSTAVKE.INI .....	47
Primjer 3.1.2. Čitanje podataka iz inicijalizacijske datoteke .....	47
Primjer 3.1.3. Pohrana podataka u inicijalizacijsku datoteku .....	48
Primjer 3.2.1. Pohrana postavki aplikacije u Windows registar .....	50
Primjer 3.2.2. Čitanje postavki aplikacije iz Windows registra .....	51
Primjer 3.2.3. Brisanje ključa iz Windows registra .....	52
Primjer 3.3.1. Inicijalni sadržaj datoteke adresar.xml .....	53
Primjer 3.3.2. Sučelje elementa "adresar" .....	55
Primjer 3.3.3. Sučelje elementa "kontakt" .....	55
Primjer 3.3.4. Čitanje svih kontakata iz XML dokumenta .....	56
Primjer 3.3.5. Dodavanje novog kontakta u XML datoteku .....	57
Primjer 3.3.6. Uređivanje postojećeg kontakta u XML datoteci .....	58
Primjer 3.3.7. Brisanje kontakta iz XML datoteke .....	59
Primjer 3.4.1. Adresar u JSON formatu .....	60
Primjer 3.4.2. Čitanje sadržaja JSON datoteke (pristup DOM) .....	60
Primjer 3.4.3. Čitanje sadržaja JSON datoteke (pristup čitačem) .....	61
Primjer 3.4.4. Generiranje sadržaja JSON dokumenta .....	62
Primjer 3.5.1. Struktura novog formata .....	64
Primjer 3.5.2. Pohranjivanje zapisa u datoteku novog formata .....	65

Primjer 3.5.3. Čitanje podataka iz datoteke novog formata .....	66
Primjer 4.3.1. Sekvencijalno čitanje svih zapisa u podatkovnom setu .....	75
Primjer 4.3.2. Dodavanje novog zapisa u podatkovni set .....	76
Primjer 4.3.3. Lociranje i brisanje zapisa iz podatkovnog seta .....	76
Primjer 4.3.4. Filtriranje podatkovnog seta (svojstvo Filter) .....	77
Primjer 4.3.5. Filtriranje podatkovnog seta (događaj OnFilterRecord) .....	78
Primjer 4.3.6. Izvršavanje SQL upita (komponenta TADOQuery) .....	78
Primjer 4.3.7. Uskladištena procedura "DohvatiStudente" .....	79
Primjer 4.3.8. Poziv uskladištene procedure .....	79
Primjer 4.3.9. Pozivanje uskladištene procedure s parametrom .....	79
Primjer 4.5.1. Implementacija izračunatog polja .....	82
Primjer 4.5.2. Konfiguracija komponente TDBLookupComboBox .....	84
Primjer 4.6.1. Uređivanje sadržaja izvještaja korištenjem programskog kôda .....	87
Primjer 5.1.1. Inicijalizacija elemenata vektora slučajno generiranim brojevima .....	90
Primjer 5.1.2. Pretraživanje prim brojeva u glavnoj dretvi .....	90
Primjer 5.1.3. PrimeSearchThread - derivacija klase TThread .....	92
Primjer 5.1.4. Konstruktor klase PrimeSearchThread .....	92
Primjer 5.1.5. Metoda Execute .....	93
Primjer 5.1.6. Kreiranje i pokretanje dretvi .....	93
Primjer 5.2.1. Sinkronizacija dretve pozivom metode Synchronize .....	95
Primjer 5.3.1. Konkurentni pristup zajedničkom resursu .....	97
Primjer 5.3.2. Inicijalizacija i brisanje objekta kritične sekcije .....	98
Primjer 5.3.3. Pristup resursu unutar kritične sekcije .....	98
Primjer 5.4.1. Kreiranje i uništavanje mutex objekta .....	100
Primjer 5.4.2. Sinkronizacija dretve upotrebom mutex objekta .....	100
Primjer 5.4.3. Sinkronizacija dretvi različitih procesa .....	101
Primjer 5.5.1. Pretraga prim brojeva (Parallel::For) .....	102
Primjer 5.5.2. Pretraga prim brojeva (_di_ITask) .....	103
Primjer 5.5.3. Korištenje budućnosti (eng. future) .....	104
Primjer 5.6.1. Aplikacija komandne linije .....	105
Primjer 5.6.2. Instanciranje procesa i dohvaćanje povratne vrijednosti .....	106
Primjer 6.2.1. Slanje zahtjeva korištenjem Indy TCP klijenta .....	111
Primjer 6.2.2. Implementacija Indy TCP poslužitelja .....	112
Primjer 6.2.3. Sinkronizirano ažuriranje grafičkog okruženja poslužitelja .....	113

Primjer 6.2.4. Slanje datoteke Indy TCP poslužitelju .....	113
Primjer 6.2.5. Preuzimanje datoteke u TCP poslužitelju .....	114
Primjer 6.3.1. Struktura paketa .....	115
Primjer 6.3.2. Slanje paketa korištenjem UDP klijenta .....	116
Primjer 6.3.3. Implementacija Indy UDP poslužitelja.....	116
Primjer 6.3.4. Slanje podatkovnog paketa korištenjem broadcast adrese .....	117
Primjer 6.4.1. Preuzimanje datoteke korištenjem Indy HTTP klijenta .....	118
Primjer 6.4.2. Prikaz napretka preuzimanja datoteke.....	119
Primjer 6.5.1. Poziv metode Ping .....	121
Primjer 6.5.2. Obrada događaja OnReplay .....	121
Primjer 7.1.1. Calculator.h - sučelje web servisa Calculator .....	126
Primjer 7.1.2. Poziv metode Multiply .....	127
Primjer 7.2.1. Matematika.h – sučelje IMatematika.....	129
Primjer 7.2.2. Matematika.cpp – implementacija metode kvadrat .....	129
Primjer 7.2.3. Testni SOAP klijent .....	129
Primjer 7.3.1. Indy HTTP klijent i REST web servis .....	131
Primjer 7.3.2. Odgovor REST poslužitelja .....	131
Primjer 7.4.1. Implementacija podrazumijevane akcije .....	133
Primjer 7.4.2. Implementacija akcije actSuma .....	135
Primjer 7.4.3. Implementacija akcije actPrim .....	136
Primjer 7.4.4. Predaja podataka korištenjem URL putanje .....	137
Primjer 7.4.5. Podatkovni tok kao odgovor web poslužitelja .....	138
Primjer 7.4.6. Autentifikacija korisnika web poslužitelja .....	139
Primjer 8.1.1. Šifriranje i dešifriranje teksta korištenjem algoritma AES .....	146
Primjer 8.1.2. Šifriranje i dešifriranje datoteke korištenjem tokova .....	148
Primjer 8.2.1. Generiranje RSA ključeva .....	150
Primjer 8.2.2. Šifriranje poruke korištenjem javnog ključa primatelja .....	150
Primjer 8.2.3. Dešifriranje poruke korištenjem privatnog ključa primatelja .....	151
Primjer 8.3.1. Izračun SHA-256 sažetka korištenjem biblioteke LockBox 3 .....	153
Primjer 8.3.2. Izračun SHA-2 sažetka korištenjem system.hash biblioteke .....	153
Primjer 8.4.1. Kreiranje digitalnog potpisa dokumenta.....	155
Primjer 8.4.2. Verifikacija digitalnog potpisa .....	156
Primjer 9.1.1. Sadržaj statičke biblioteke .....	160
Primjer 9.1.2. Poziv funkcije iz statičke biblioteke .....	160

Primjer 9.2.1. Funkcija DllEntryPoint .....	162
Primjer 9.2.2. Sadržaj datoteke dll_exports.h .....	162
Primjer 9.2.3. Sadržaj datoteke dll_exports.cpp.....	165
Primjer 9.3.1. Pozivi DLL funkcija .....	166
Primjer 9.3.2. Dinamičko DLL povezivanje .....	167
Primjer 9.3.3. Poziv DLL funkcije u C# .NET aplikaciji .....	168
Primjer 9.3.4. Učitavanje karti iz DLL biblioteke.....	168
Primjer 9.4.1. Sadržaj datoteke ComplexNumber.h .....	171
Primjer 9.4.2. Implementacija metode Modulus .....	172
Primjer 9.4.3. Razvoj komponente TComplexNumber u Delphiju .....	173

## Popis tablica

Tablica 2.2.1. Cjelobrojne konstante gumbi i ikona funkcije MessageBox [5] .....	23
Tablica 5.1.1. Rezultati testiranja.....	94





## Reference

- [1] Embarcadero Technologies, Inc., »Supported Target Platforms,« [Mrežno]. Available:  
[http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Supported\\_Target\\_Platforms](http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Supported_Target_Platforms).
- [2] P. Sorfa, »Kylux,« 1 8 2001. [Mrežno]. Available:  
<https://www.linuxjournal.com/article/4781>. [Pokušaj pristupa 12 2 2019].
- [3] L. Navarro, »C++ Builder Pre-Compiled Header with CLANG compiler,« 3 6 2017. [Mrežno]. Available: <https://community.embarcadero.com/blogs/entry/pre-compiled-header-with-clang-compiler>. [Pokušaj pristupa 16 2 2019].
- [4] SmartBear Software, »Compiler Settings for Embarcadero C++Builder XE6 - XE8 and 10 - 10.2 (32-bit),« SmartBear Software, 5 12 2018. [Mrežno]. Available:  
<https://support.smartbear.com/aqtime/docs/profiling-with/prepare-apps/compiler-settings-native/c-builder/c-builder-xe6-x86.html>. [Pokušaj pristupa 19 2 2019].
- [5] Microsoft Corporation, »MessageBox function,« Microsoft Corporation, 12 5 2018. [Mrežno]. Available: <https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-messagebox>. [Pokušaj pristupa 20 2 2019].
- [6] Embarcadero Technologies, Inc., »RAD Studio VCL Reference, Controls.TModalResult Type,« [Mrežno]. Available:  
[http://docs.embarcadero.com/products/rad\\_studio/delphiAndcpp2009/HelpUpdate2/EN/html/delphivclwin32/Controls\\_TModalResult.html](http://docs.embarcadero.com/products/rad_studio/delphiAndcpp2009/HelpUpdate2/EN/html/delphivclwin32/Controls_TModalResult.html). [Pokušaj pristupa 21 2 2019].
- [7] Embarcadero Technologies, Inc., » RAD Studio VCL Reference, UnicodeString Members,« [Mrežno]. Available:  
[http://docs.embarcadero.com/products/rad\\_studio/delphiAndcpp2009/HelpUpdate2/EN/html/delphivclwin32/!!MEMBEROVERVIEW\\_System\\_\\_UnicodeString.html](http://docs.embarcadero.com/products/rad_studio/delphiAndcpp2009/HelpUpdate2/EN/html/delphivclwin32/!!MEMBEROVERVIEW_System__UnicodeString.html). [Pokušaj pristupa 22 2 2019].
- [8] Embarcadero Technologies, Inc., »Localizing Applications,« 26 3 2015. [Mrežno]. Available:  
[http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Localizing\\_Applications](http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Localizing_Applications). [Pokušaj pristupa 2 3 2019].

- [9] T. Fisher, »What Is the Windows Registry?,« 22 12 2018. [Mrežno]. Available: <https://www.lifewire.com/windows-registry-2625992>. [Pokušaj pristupa 5 3 2019].
- [10] Embarcadero Technologies, Inc., »Using the XML Data Binding Wizard,« 12 12 2014. [Mrežno]. Available: [http://docwiki.embarcadero.com/RADStudio/Rio/en/Using\\_the\\_XML\\_Data\\_Binding\\_Wizard](http://docwiki.embarcadero.com/RADStudio/Rio/en/Using_the_XML_Data_Binding_Wizard). [Pokušaj pristupa 7 3 2019].
- [11] »Introducing JSON,« [Mrežno]. Available: <https://www.json.org/>. [Pokušaj pristupa 11 3 2019].
- [12] Ž. Gajić, »The TStream Class in Delphi,« 17 3 2017. [Mrežno]. Available: <https://www.thoughtco.com/tstream-class-in-delphi-4077896>. [Pokušaj pristupa 15 3 2019].
- [13] »Database connectivity frameworks in Delphi,« Ultimate Delphi, 14 4 2016. [Mrežno]. Available: <https://ultimatedelphi.wordpress.com/2016/04/14/database-connectivity-frameworks-in-delphi/>. [Pokušaj pristupa 27 3 2019].
- [14] E. M. Peter Darakhvelidze, »Data Access Technologies,« u *Web Services Development with Delphi*, БХВ-Петербург, 2002, p. 196.
- [15] C. Cunningham, »What is Hyper-Threading and Simultaneous MultiThreading?,« 13 10 2017. [Mrežno]. Available: <http://blog.logicalincrements.com/2017/10/what-is-hyper-threading-simultaneous-multithreading/>. [Pokušaj pristupa 18 4 2019].
- [16] M. Harvey, »Chapter 3. Basic synchronization,« 2000. [Mrežno]. Available: <http://www.nickhodes.com/MultiThreadingInDelphi/Ch3.html>. [Pokušaj pristupa 21 4 2019].
- [17] C. Jensen, »Using Semaphores in Delphi, Part 1,« Embarcadero, 2003. [Mrežno]. Available: <http://edn.embarcadero.com/article/29908>. [Pokušaj pristupa 25 4 2019].
- [18] Embarcadero Technologies, Inc., »System Threading TParallel For,« 6 1 2015. [Mrežno]. Available: <http://docwiki.embarcadero.com/Libraries/Rio/en/System.Threading.TParallel.For>. [Pokušaj pristupa 2020 1 9].
- [19] Micro Focus, »Understanding Synchronous and Asynchronous Communication,« [Mrežno]. Available: <http://documentation.microfocus.com/help/index.jsp?topic=%2Fcom.microfocus.silkp>

- erformer.doc%2FGUID-6CC17B5B-71B7-4703-B9E6-C81835A5335A.html.  
[Pokušaj pristupa 13 4 2019].
- [20] Mozilla Corporation, »HTTP,« 18 3 2019. [Mrežno]. Available:  
<https://developer.mozilla.org/en-US/docs/Web/HTTP>. [Pokušaj pristupa 28 4 2019].
- [21] Stackify, »What are Windows Services? How Windows Services Work, Examples, Tutorials and More,« 2 6 2017. [Mrežno]. Available: <https://stackify.com/what-are-windows-services/>. [Pokušaj pristupa 2 5 2019].
- [22] Embarcadero Technologies, Inc., »Inheritance and Interfaces,« 18 11 2016.  
[Mrežno]. Available:  
[http://docwiki.embarcadero.com/RADStudio/Rio/en/Inheritance\\_and\\_Interfaces](http://docwiki.embarcadero.com/RADStudio/Rio/en/Inheritance_and_Interfaces).  
[Pokušaj pristupa 8 5 2019].
- [23] A. Valdes, »Best Practices for RESTful API Design,« 20 8 2017. [Mrežno]. Available:  
<https://avaldes.com/best-practices-for-restful-api-design/>. [Pokušaj pristupa 12 5 2019].
- [24] eTutorials.org, »Delphi's WebBroker Technology,« [Mrežno]. Available:  
<http://etutorials.org/Programming/mastering+delphi+7/Part+IV+Delphi+the+Internet+and+a+.NET+Preview/Chapter+20+Web+Programming+with+WebBroker+and+Web+Snap/Delphi+s+WebBroker+Technology/>. [Pokušaj pristupa 18 5 2019].
- [25] FUJITSU LIMITED, »Differences between ISAPI and CGI,« [Mrežno]. Available:  
<http://software.fujitsu.com/jp/manual/manualfiles/m150010/b1wd3363/01enz200/b3363-00-02-03-00.html>. [Pokušaj pristupa 20 5 2019].
- [26] Secret Double Octopus, »Symmetric Key Cryptography,« [Mrežno]. Available:  
<https://doubleoctopus.com/security-wiki/encryption-and-cryptography/symmetric-key-cryptography/>. [Pokušaj pristupa 15 6 2019].
- [27] M. Rouse, »Cipher block chaining (CBC),« WhatIs.com, 6 2007. [Mrežno]. Available:  
<https://searchsecurity.techtarget.com/definition/cipher-block-chaining>. [Pokušaj pristupa 17 6 2019].
- [28] SSL2BUY, »Symmetric vs. Asymmetric Encryption – What are differences?,« [Mrežno]. Available: <https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences>. [Pokušaj pristupa 18 6 2019].
- [29] Ž. Kovačević, Modeliranje, implementacija i administracija baza podataka, Zagreb: Tehničko veleučilište u Zagrebu, 2018.

- [30] B. Wyro, »Encrypting vs. Signing with OpenPGP. What's the Difference?,« 29 5 2018. [Mrežno]. Available: <http://blogs.mdaemon.com/index.php/2018/05/29/encrypting-vs-signing-with-openpgp-whats-the-difference-2/>. [Pokušaj pristupa 20 6 2019].
- [31] Reva Solutions, »digital-signatures-methodology,« [Mrežno]. Available: <https://www.revasolutions.com/expertise/process-management/digital-signatures/digital-signatures-methodology/>. [Pokušaj pristupa 20 6 2019].
- [32] Yevol, »Using Additional Libraries - Custom Libraries,« [Mrežno]. Available: <http://www.yevol.com/bcb/Lesson06.htm>. [Pokušaj pristupa 22 6 2019].
- [33] Microsoft Corporation, »Exporting from a DLL Using \_\_declspec(dllexport),« [Mrežno]. Available: <https://docs.microsoft.com/en-us/cpp/build/exporting-from-a-dll-using-declspec-dllexport?view=vs-2019>. [Pokušaj pristupa 30 6 2019].
- [34] Microsoft Corporation, »About Dynamic-Link Libraries,« [Mrežno]. Available: <https://docs.microsoft.com/en-us/windows/desktop/dlls/about-dynamic-link-libraries>. [Pokušaj pristupa 1 7 2019].
- [35] J. Spacey, »7 Examples of Software Components,« Simplicable, 23 5 2017. [Mrežno]. Available: <https://simplicable.com/new/software-components>. [Pokušaj pristupa 2 7 2019].

## Ključne riječi

### A

ADO · 69  
 Akcijska lista · 40  
 Asinkrona komunikacija · 109

### B

Base64 · 140, 147  
 BDE · 69  
 Biblioteka · 159  
 Bindings · 111  
 Blokirajuća utičnica · 109  
 Broadcast · 117  
 Budućnost · 104

### C

CBC · 147  
 ConnectionString · 73  
 CreateSuspended · 92

### D

Data Controls · 70, 75  
 DataSource · 70  
 Debugger · 16  
 Dfm · 20  
 Digitalni potpis · 154  
 Dinamičke biblioteke · 161  
 Dinamičko DLL povezivanje · 167  
 DLL resursi · 168  
 DllEntryPoint · 162  
 dllexport · 163  
 dllimport · 163  
 Događaji · 31  
 Dretva · 89

### E

ECB · 148  
 EncryptString · 151  
 EnterCriticalSection · 98  
 Events · 33  
 Execute · 93  
 extern "C" · 163

### F

Fast Report · 84  
 FieldByName · 76  
 Filter · 77  
 FireDAC · 69  
 Funkcije sažimanja · 152  
 future · 104

### G

GetExitCodeProcess · 107  
 GetIt Package Manager · 9

### H

HashFile · 153  
 HashString · 153  
 HTTP · 117

### I

ICMP · 120  
 IDE · 10  
 IndexFieldNames · 77  
 INI datoteke · 47  
 InputBox · 27  
 InputQuery · 28

IntelliSense · 15

ISAPI · 127

Izračunata polja · 82

---

## **J**

Javni ključ · 150

JSON · 59

JSON čitač · 61

JSON DOM pristup · 60

---

## **K**

Klijent-poslužitelj model · 109

Ključ i vrijednost · 59

Kritična sekcija · 96

---

## **L**

LeaveCriticalSection · 98

LoadNewResourceModule · 45

Locate · 77

Lokalizacija · 41

Lookup polja · 82

---

## **M**

Master-detail · 80

MasterFields · 81

MasterSource · 81

MessageBox · 22, 24

MessageDlg · 25

Metoda komponente · 31

Mutex · 99

---

## **N**

Neblokirajuće utičnice · 110

Nit · 89

NumbersOnly · 35

---

## **O**

Objektni inspektor · 21

OnExecute · 112

OnFilterRecord · 78

OnReply · 121

---

## **P**

Parallel Programming Library · 101

Parallel::For · 102

Podatkovni modul · 72

Post · 76

Prilagođeni format · 64

Privatni ključ · 150

Proces · 105

---

## **R**

RAD · 9

RAD Studio · 11

RawToBytes · 116

ReadStream · 114

Refaktoriranje · 16

REST · 130

REST načela · 130

RSA · 149

---

## **S**

SHA-1 · 153

SHA-256 · 153

ShellExecute · 107

ShowMessage · 22

ShowModal · 39

Sign · 156

Simetrični kriptografski algoritmi · 145

Sinkrona komunikacija · 109

SOAP · 125

Statičke biblioteke · 159

Statičko DLL povezivanje · 167

Svojstva · 31

Synchronize · 95, 113

UTF-8 · 54

---

## T

TADOConnection · 73

TADOQuery · 78

TADOStoredProc · 79

TADOTable · 76

task · 104

TCP · 110

TDBLookupListBox · 84

TEdit · 35

TFileStream · 114

TIdBytes · 116

TIdInterceptThrottler · 120

TIniFile · 48

TMemo · 35

TMutex · 99

TSignatory · 156

TStream · 66

TTaskDialog · 25, 29

TThread · 91

---

## U

UDL · 73

UDP · 115

UnicodeString · 28

UniDAC · 69

Uređivač programskog kôda · 15

---

## V

VCL · 10, 19

Verify · 156

---

## W

WaitFor · 94

WaitForSingleObject · 100

wchar\_t · 65

WebBroker · 127, 132

WebBroker REST · 133

Windows dijalozi · 36

Windows IIS · 140

Windows registar · 49

Windows servisi · 122

WinExec · 107

WSDL uvoznik · 125

---

## X

XML · 52

XML Data Binding Wizard · 54

---

## Z

Zadatak · 104