©Rivet... objektově-relační vrstva

PLNĚ OBJEKTOVÝ SYSTÉM PERSISTENCE OBJEKTŮ A JEJICH VAZEB

Prostředí:

- **C#** (.NET framework 3.5+)
- **MS SQL**

Charakteristika:

- Přístup **Code First** - navrhnou se třídy a systém z nich vygeneruje databázový skript
- V případě změny se nemusí přegenerovat kód, je možno ručně upravit třídu + tabulku v DB
- Reflection je optimalizováno s použitím **cache** a **proxy** objektů
- **Univerzální přístup** k objektům (každý persistentní objekt je potomkem třídy Proxy)
- Persistentní třídy plně zachovávají **dědičnost** i **polymorfismus**
- Každé persistentní třídě odpovídá tabulka v databázi
- Každá instance má záznam v tabulce své třídy + v tabulkách tříd svých „předků“
- Vazba přes primární klíč OID typu **Guid** (odpadají problémy s autoinkrementem)
- Každý objekt má záznam v tabulce Proxy (je-li znám typ objektu, nenačítá se z ní)
- **Seznamy** pracují výhradně s lehkými objekty proxy (obsahují prakticky pouze OID)
- Objekty se plně načítají až v případě potřeby metodou **Cast**
- Seznamy se plní standardními SQL dotazy (**zachována mocnost příkazu SELECT**)
- Seznamy mají zabudováno **stránkování**, **třídění** a **filtrování**
- Podpora **transakcí**
- Relace **1xN** i **NxN** (ve vazbách objekty vystupují jednotně jako typ Proxy)
- Metodou **LightCast** získáme vazby objektu, aniž by se načítaly ostatní vlastnosti
- Zabudovaná **Cache** je vždy aktuální - kontrolováno dotazem do základní tabulky třídy
- free open source

Příklad použití

Jako příklad jsem zvolil intuitivní záznam o osobách a jejich rodinných vazbách – **RODOKMEN**.

Pozn.:
Budiž mi prominuto trošku nekorektní (a nesmyslné) odlišení záznamu u muže a u ženy. Tyto záznamy jsem chtěl více vyzájemně odlišit a použít při tom pokud možno různé typy polí. Jedná se konec konců o testovací program.

Persistentní třída Osoba

Persistentní třída **Osoba** dědí od třídy **Proxy**. Je to základní třída, uchovávající záznam o osobě. Od této třídy odvodíme třídy **Muz** a **Zena**.

Poznámky ke kódu:

- třída **Proxy** je předkem všech persistentních tříd a obsahuje primární klíč **OID**
- vždy je požadován základní konstruktor
- [MaxLength(20)] - zadání pro generátor DB skriptu - šířka sloupce v tabulce
- **DateTime? -** lze použít nulovatelné typy
- obrázek uložíme jako **Stream**
- [ForeignKey(typeof(Muz))] - takto se definuje cizí klíč – Otec je typu **Muz**
- **public Proxy** Otec - vazba 1xN je vždy typu **Proxy**
- [ForeignKey(typeof(Zena))] - Matka je typu **Zena**
- metoda GetDeti() bude přetížena v potomcích
- Matka.LightCast<Zena>() - metoda **LightCast** provádí „lehkou“ konverzi na cílový typ - načítá pouze vazby, nikoli všechny hodnoty
- GetDeti() - vrátí seznam dětí
- Union(...) - operace sjednocení seznamů
- Subtract(this) - nejsme sami svým sourozencem :-)
- ToString() - příklad, jak se definuje „zkratka“ objektu

```
using System;

using System.IO;
using AVRivet;

namespace Rodokmen
{
        public class Osoba : Proxy      //persistentní třída „Osoba“. Dědí se vždy od třídy „Proxy“
        {                               //třída „Proxy“ obsahuje primární klíč OID (GUID)
                public Osoba() { }      //vždy je požadován základní konstruktor

                [MaxLength(20)]         //zadání pro generátor DB skriptu - šířka sloupce v tabulce
                public string Jmeno;
                [MaxLength(50)]
                public string Prijmeni;
                public DateTime? DatNarozeni;       //lze použít nulovatelné typy...
                public Stream Portret;              //obrázek uložíme jako Stream...
                [ForeignKey(typeof(Muz))]           //takto se definuje cizí klíč – Otec je typu „Muz“
                public Proxy Otec;                  //vazba 1xN je vždy typu Proxy
                [ForeignKey(typeof(Zena))]          //...Matka je typu „Zena“
```

```
        public Proxy Matka;

        public virtual ProxyList GetDeti() { return null; }          //bude přetíženo v potomcích...
        public ProxyList GetSourozenci()          //LightCast provádí „lehkou“ konverzi na cílový
        {                                         //typ - načítá pouze vazby, nikoli všechny hodnoty.
            return Matka.LightCast<Zena>()
                .GetDeti()                        //vrátí seznam dětí matky
                .Union(                           //operace sjednocení seznamů
                    Otec.LightCast<Muz>()
                    .GetDeti()                    //vrátí seznam dětí otce
                ).Subtract(this);                 //nejsme sami svým sourozencem :-)
        }

        public override string ToString()         //příklad, jak se definuje „zkratka“ objektu
        {
            return string.Format("{0} {1} {2}",
                Jmeno,
                Prijmeni,
                DatNarozeni.HasValue ?
                    DatNarozeni.Value.Year.ToString()
                    :
                    string.Empty
                ).Trim();
        }
    }
}
```

Vazební třída Partnerstvi

Definuje vazební třídu **Partnerstvi** jako potomka třídy **RelationNxN**.

Poznámky ke kódu:

- vazba NxN přes tabulku „Partnerstvi“
- třída mezi sebou sváže třídy **Muz** a **Zena**

```
using AVRivet;

namespace Rodokmen
{
    public class Partnerstvi : RelationNxN  //vazba NxN přes tabulku „Partnerstvi“– viz. výše...
    {
        public Proxy Muz;
        public Proxy Zena;
    }
}
```

Třída Muz

Persistentní třída **Muz** dědí od třídy **Osoba**.

Poznámky ke kódu:

- persistentní třída **Muz** dědí od třídy **Osoba** a potažmo od třídy **Proxy**
- **public decimal?** Plat - nepovinné nulovatelné pole
- GetDeti() - přetěžujeme metodu, definovanou ve třídě **Osoba -** „zpětná“ vazba Nx1 je seznamem. Mnemo: „mé děti jsou všechny osoby, kde já jsem otec“.
- GetPartneri() - vrátí všechny partnerky, navázané přes vazbu **Partnerstvi**. Mnemo: „všechny ženy z partnerství, kde já jsem muž“
- AddPartner(Zena zena) - přidá vazbu na partnerku. Mnemo: „přidej partnerství, kde já jsem muž a partner je žena“
- RemovePartner(Zena zena) - odebere vazbu na partnerku. Mnemo: „odeber partnerství, kde já jsem muž a partner je žena“

```
using AVRivet;

namespace Rodokmen
{
	public class Muz : Osoba		//persistentní třída „Muz“ dědí od třídy „Osoba“
	{					//...a potažmo od třídy „Proxy“
		public Muz() { }

		[MaxLength(50)]
		public string Povolani;
		public bool Vousy;
		public decimal? Plat;		//nepovinné pole - nulovatelné

		public override ProxyList GetDeti()	//...přetěžujeme...
		{
			return GetNx1<Osoba>(a => a.Otec);	//„zpětná“ vazba Nx1 je seznamem
		}					//mnemo: „mé děti jsou všechny osoby, kde já jsem otec“

		public ProxyList GetPartneri()	//vrátí všechny partnerky, navázané přes vazbu „Partnerstvi“
		{
			return GetNxN<Partnerstvi>(a => a.Muz, b => b.Zena);
		}				//mnemo: „všechny ženy z partnerství, kde já jsem muž“

		public void AddPartner(Zena zena)	//přidá vazbu na partnerku
		{
			AddNxN<Partnerstvi>(a => a.Muz, b => b.Zena, zena);
		}				//mnemo: „přidej partnerství, kde já jsem muž a parametr je žena“

		public void RemovePartner(Zena zena)		//odebere vazbu na partnerku
		{
			RemoveNxN<Partnerstvi>(a => a.Muz, b => b.Zena, zena);
		}				//mnemo: „odeber partnerství, kde já jsem muž a parametr je žena“
	}
}
```

Třída Zena

Persistentní třída **Zena** dědí od třídy **Osoba**.

Poznámky ke kódu:

- persistentní třída **Zena** dědí od třídy **Osoba** a potažmo od třídy **Proxy**
- **public int** ObvodPasu - povinné nenulovatelné pole (:-) sorry, potřebuji příklad číselného pole)
- GetDeti() - přetěžujeme metodu, definovanou ve třídě **Osoba -** „zpětná“ vazba Nx1 je seznamem. Mnemo: „mé děti jsou všechny osoby, kde já jsem matka“.
- GetPartneri() - vrátí všechny partnery, navázané přes vazbu **Partnerstvi**. Mnemo: „všichni muži z partnerství, kde já jsem žena“
- AddPartner(Muz muz) - přidá vazbu na partnera. Mnemo: „přidej partnerství, kde já jsem žena a partner je muž“
- RemovePartner(Muz muz) - odebere vazbu na partnerku. Mnemo: „odeber partnerství, kde já jsem žena a partner je muž“

```
using AVRivet;

namespace Rodokmen
{
        public class Zena : Osoba          //persistentní třída „Zena“ dědí od třídy „Osoba“
        {                                  //...a potažmo od třídy „Proxy“
                public Zena() { }

                 [MaxLength(50)]
                 public string Rozena;
                 public int ObvodPasu;           //povinné pole – nenulovatelné (sorry - pro ukázku)

                public override ProxyList GetDeti()      //...přetěžujeme...
                {
                        return GetNx1<Osoba>(a => a.Matka);   //„zpětná“ vazna Nx1 je seznamem
                }                                  //mnemo: „mé děti jsou všechny osoby, kde já jsem matka“

                public ProxyList GetPartneri()  //vrátí všechny partnery, navázané přes vazbu „Partnerstvi“
                {
                        return GetNxN<Partnerstvi>(a => a. Zena, b => b. Muz);
                }                                  //mnemo: „všichni muži z partnerství, kde já jsem žena“

                public void AddPartner(Muz muz)       //přidá vazbu na partnera
                {
                        AddNxN<Partnerstvi>(a => a. Zena, b => b. Muz, muz);
                }                                  //mnemo: „přidej partnerství, kde já jsem žena a parametr je muž“

                public void RemovePartner(Muz muz) //odebere vazbu na partnera
                {
                        RemoveNxN<Partnerstvi>(a => a. Zena, b => b. Muz, muz);
                }                                  //mnemo: „odeber partnerství, kde já jsem žena a parametr je muž“
        }
}
```

A to je vše.

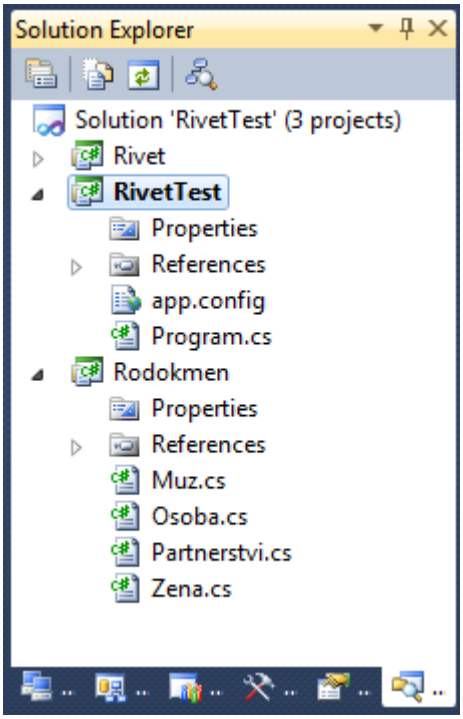
Tímto jsme nadefinovali naše persistentní třídy.
Tento kód přeložme jako samostatnou assembly (projekt typu „**Class Library**“).

Konzolová testovací aplikace

Pro práci s vrstvou **Rivet** si vytvoříme ve **Visual Studiu** jednoduchou konzolovou aplikaci **RivetTest**.

Dále si někde, kam máme přístup, vytvoříme prázdnou **databázi** RivetTest, například pomocí **Management Studia**. Následně budeme potřebovat znát **Connection String** pro připojení aplikace k této databázi.

Naše řešení bude vypadat asi takto:



Vygenerování skriptu databáze

Základní kód konzolové aplikace rozšíříme o volání metod:

- Init() - propojí vytvořenou assembly s definicí persistentních tříd a s databází. Tato metoda nás bude nadále trvale provázet...
- ScriptDatabase() - metoda ScriptDatabase nám vygeneruje databázový script, kterým následně vytvoříme potřebné databázové tabulky a vazby...

Inicializace sestává ze tří příkazů:

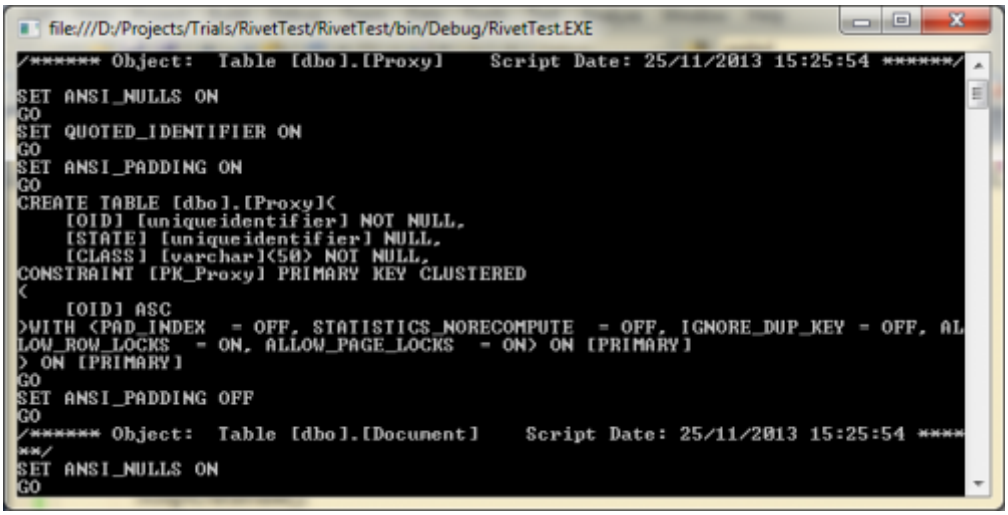
- Rivet.ConnectionString - zde uvedeme platný Connection String
- Rivet.UserAssemblyFullPath - zde uvedeme plnou cestu k assembly s definicí tříd
- Rivet.Namespace - zde uvedeme namespace, použité v assembly

Hodnoty načteme ze souboru **app.config**, kam doplníme sekci **appSettings**, např.:

```
<appSettings>
   <add key="ConnectionString" value="Data Source=PETRT\SQLEXPRESS;  Initial Catalog=RivetTest;  User
ID=sa;  Password=P@ssw0rd" />
   <add key="UserAssemblyFullPath"
value="D:\Projects\Trials\RivetTest\RivetTest\bin\Debug\Rodokmen.dll" />
   <add key="Namespace" value="Rodokmen" />
 </appSettings>
```

```
using System;
using AVRivet;
using Rodokmen;

namespace RivetTest
{
        class Program
        {
                static void Main(string[] args)
                {
                        //metoda Init propojí vytvořenou assembly s definicí persistentních tříd a s databází
                        //tato metoda nás bude nadále trvale provázet...
                        Init();

                        //metoda ScriptDatabase nám vygeneruje databázový script, kterým následně vytvoříme
                        //potřebné databázové tabulky a vazby...
                        ScriptDatabase();
                }

                private static void Init()
                {
                        Rivet.ConnectionString =
                                ConfigurationManager.AppSettings["ConnectionString"];
                        Rivet.UserAssemblyFullPath =
                                ConfigurationManager.AppSettings["UserAssemblyFullPath "];
                        Rivet.Namespace =                   }
                                ConfigurationManager.AppSettings["Namespace "];

                private static void ScriptDatabase()
                {
                        string script = Rivet.Instance.ScriptDatabase();
                        Console.Write(script);
                        Console.ReadLine();
                }
        }
}
```

Výsledkem je skript, kterým následně (třeba pomocí **Management Studia**) vytvoříme potřebné databázové tabulky a vazby:

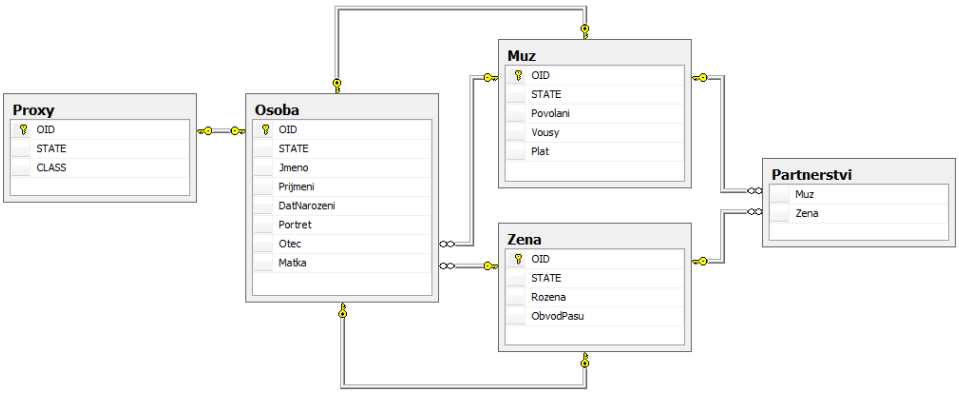


Pozn.:
Výpis do konzole zalamuje řádky. Před spuštěním skriptu je třeba chybná zalomení opravit.
Lepší možnost je, nechat si např. na ploše vytořit soubor se skriptem:

```
private static void ScriptDatabase()
{
        string path = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
        string script = Rivet.Instance.ScriptDatabase();
        System.IO.File.WriteAllText(string.Format("{0}\\RivetScript.sql", path), script);
}
```

Schema databáze

Zde je schema databáze, která bude následně skriptem vygenerována:
O položkách **OID**, **STATE** a **CLASS** si povíme za chvilku...



Nyní můžeme se systémem pracovat...

Pozn.:
Ve skutečnosti systém obsahuje ještě dvě tabulky: Document a Document2Proxy.
Systém obsahuje také persistentní třídu Document, která umožňuje navázat k jakémukoli persistentnímu objektu obsah libovolného (nikoli však nepřiměřeně objemného) souboru.
Záměrně jsem tuto funkčnost v příkladu ignoroval, abych jej nekomplikoval.

Naplnění systému daty

Níže uvedená metoda naplní systém pěti osobami – rodiči a třemi dětmi.

Příklad demonstruje:

- objekty je možno vytvářet „odpojené“ a připojit je k databázi teprve před uložením
- připojení objektu k databázi se provádí metodou „**Bind**“
- databázové příkazy INSERT a UPDATE se provádí příkazem „**Update**“ na připojeném objektu
- připojení se provádí prostřednictvím bloku using a objektu typu **Context** (metoda **Update** vyžaduje transakční připojení **ContextTran**)
- připojení k databázi trvá uvnitř bloku **using**. Po opuštění tohoto bloku je spojení uzavřeno.
- pokud je použito transakční připojení **ContextTran**, je třeba před jeho uzavřením explicitně zavolat metodu **Commit**, v opačném případě dojde před ukončením připojení k automatickému odrolování transakce a k žádným změnám v databázi nedojde.

```
private static void Naplnit()

{
        Muz janSvoboda =
                new Muz() { Jmeno = "Jan", Prijmeni = "Svoboda",
                DatNarozeni = new DateTime(1949, 4, 9) };
        Zena janaSvobodova =
                new Zena() { Jmeno = "Jana", Prijmeni = "Svobodová",
                DatNarozeni = new DateTime(1951, 8, 2) };
        Muz jiriSvoboda =
                new Muz() { Jmeno = "Jiří", Prijmeni = "Svoboda",
                DatNarozeni = new DateTime(1971, 7, 3),
                Otec = janSvoboda, Matka = janaSvobodova };
        Muz pavelSvoboda =
                new Muz() { Jmeno = "Pavel", Prijmeni = "Svoboda",
                DatNarozeni = new DateTime(1975, 6, 19),
                Otec = janSvoboda, Matka = janaSvobodova };
        Zena evaSvobodova =
                new Zena() { Jmeno = "Eva", Prijmeni = "Svobodová",
                DatNarozeni = new DateTime(1976, 7, 21),
                Otec = janSvoboda, Matka = janaSvobodova };

        using (ContextTran ctx = Rivet.Instance.GetContextTran())
        {
                janSvoboda.Bind(ctx).Update();
                janaSvobodova.Bind(ctx).Update();
                jiriSvoboda.Bind(ctx).Update();
                pavelSvoboda.Bind(ctx).Update();
                evaSvobodova.Bind(ctx).Update();

                ctx.Commit();
        }
}
```

Výpis seznamu osob

Následující metoda vypíše na konzoli setříděný seznam osob.

Příklad demonstruje:

- třída „**ProxyList**“ reprezentuje seznam objektů „**Proxy**“
- instance třídy „**Rivet**“ (je to singleton) nabízí metodu „**Query**“, vybavenou vnitřním jednorázovým připojením k databázi. (Pokud bychom chtěli volat metodu „**Query**“ vícekrát po sobě pod stejným připojením, lze použít konstrukci typu:

```
using (Context ctx = Rivet.Instance.GetContext())
{
        ProxyList  muzi = ctx.GetProxyList().Query("select OID from Muz");
        ProxyList  zeny = ctx.GetProxyList().Query("select OID from Zena");
        ...
}
```

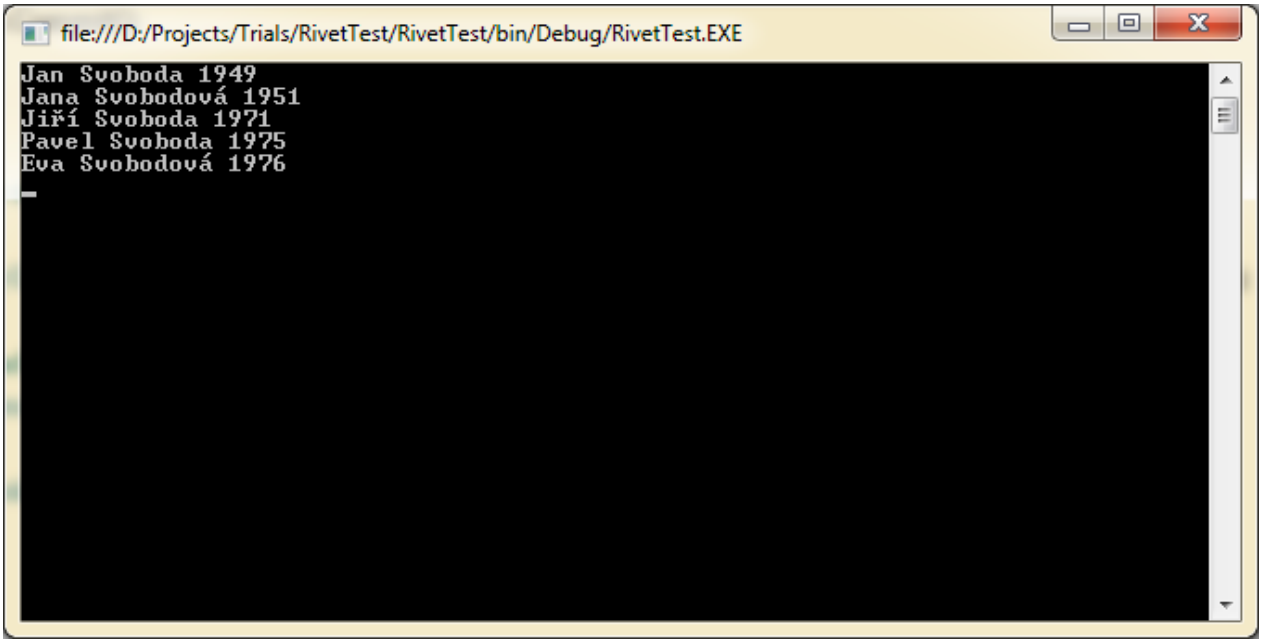
- metoda „**Query**“ používá klasický (a mocný) **SQL SELECT**, přičemž vrácené objekty jsou plně identifikovány hodnotou primárního klíče „**OID**“. Lze používat libovolné platné konstrukce dotazu, které vrátí kolekci hodnot **OID** (výhoda - příkazy lze snadno ladit např. v **Management studiu**).
- generická metoda „**Sort**“ provádí setřídění seznamu na úrovni SQL dotazu. Parametrem je **lambda výraz**, určující, podle kterého pole (sloupce) se bude třídit.
- seznamy v sobě nesou pouze hodnoty **OID** – generickou metodou „**Cast**“ lze seznam převést na příslušnou generickou kolekci (zde List).
- příklad demonstruje použití přetížené metody „**ToString**“ u persistentních objektů.

```
private static void Zobrazit()
{
        ProxyList osoby =
                Rivet.Instance.Query("select OID from Osoba")
                .Sort<Osoba>(SortDirection.Asc, a => a.DatNarozeni);

        foreach (Osoba osoba in osoby.Cast<Osoba>())
                Console.WriteLine(osoba.ToString());

        Console.ReadLine();
}
```

Ověření vytvořených metod

Výše uvedené metody doplníme do konzolové aplikace. Metoda **Naplnit** naplní databázi pěti osobami a metoda **Zobrazit** je zobrazí.

```
using System;
using AVRivet;
using Rodokmen;

namespace RivetTest
{
        class Program
        {
                static void Main(string[] args)
                {
                        Init();
                        Naplnit ();
                        Zobrazit ();
                }
                private static void Init()
                {
                        //viz. výše...
                }
                private static void Naplnit()
                {
                        //viz. výše...
                }
                private static void Zobrazit()
                {
                        //viz. výše...
                }
        }
}
```



Mazání záznamů

Následující metoda **Smazat** vymaže všechny osoby z databáze.
Metodu doplníme do konzolové aplikace a jejím voláním nahradíme volání metody **Naplnit**.

Příklad demonstruje:

- zkusme nejprve nechat zakomentované sestupné třídění (**Sort**). Metoda pravděpodobně nahlásí chybu a akce se celá „odroluje“. Databáze obsahuje tzv. „**cizí klíče**“, které hlídají její **referenční integritu**. Zde to znamená, že nelze smazat osobu, která je odkazovaná z jiné osoby jako její rodič. Pokud platí, že dítě je mladší než jeho rodiče, sestupné třídění podle data narození tento problém řeší – nejprve se smažou děti a teprve potom jejich rodiče. Odstraňme komentář před „**Sort**“ a ověřme to. Osoby by tentokrát měly z databáze zmizet.
- tímto také demonstrujeme funkci **transakce** – buď se provede v pořádku **všechno nebo nic**. Používání transakce je vhodné vždy, když nějak měníme vztahy uvnitř systému.
- na tomto příkladu je také vidět typickou vlastnost objektu třídy **ProxyList** – pokud chceme procházet objekty **Proxy**, které v sobě nese, musíme použít generickou metodu „**Cast**“:

```
foreach (Proxy proxy in osoby.Cast<Proxy>())
        proxy.Bind(ctx).Delete();
```

- Další možností je použití indexeru:

```
for (int i = 0; i < osoby.Count; i++)
        osoby[i].Bind(ctx).Delete();
```

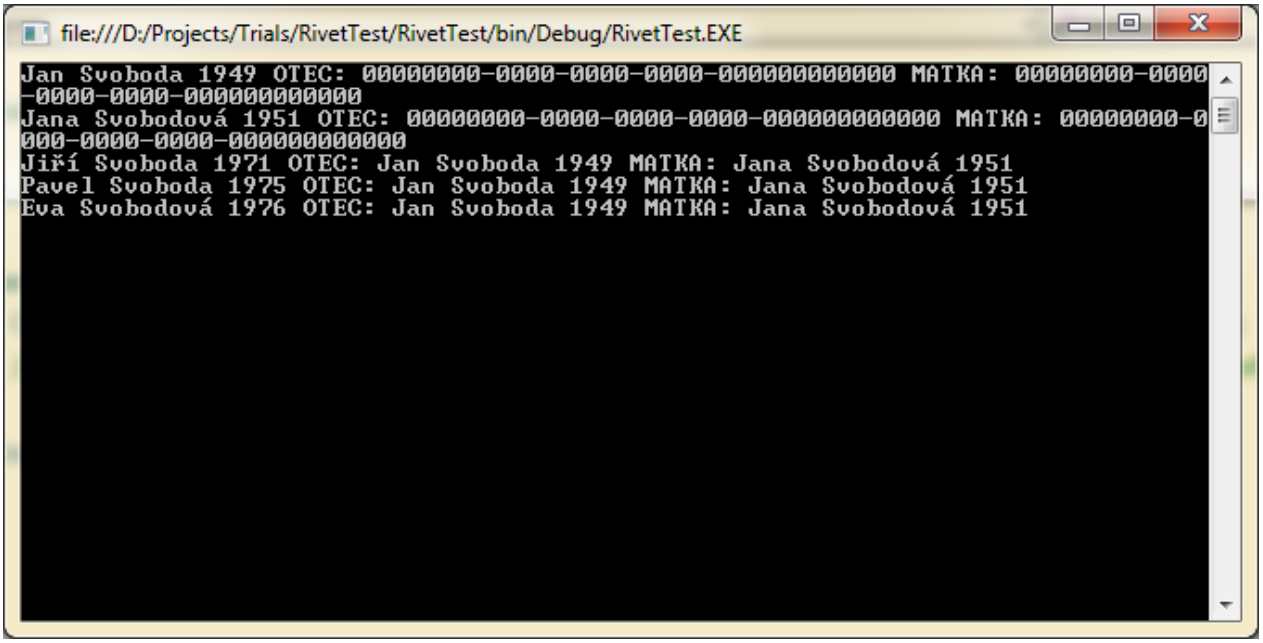
```
private static void Smazat()
{
    ProxyList osoby =
        Rivet.Instance.Query("select OID from Osoba")
        // .Sort<Osoba>(SortDirection.Desc, a => a.DatNarozeni)
        ;

    using (ContextTran ctx = Rivet.Instance.GetContextTran())
    {
        foreach (Proxy proxy in osoby.Cast<Proxy>())
            proxy.Bind(ctx).Delete();

        ctx.Commit();
    }
}
```

Zobrazení dopředných vazeb

Dopředné vazby **1xN** jsou reprezentovány polem typu **Proxy**.
Upravíme metodu **Zobrazit** tak, aby nám u každé osoby vypsala i její rodiče.

```
private static void Zobrazit()
{
        ProxyList osoby =
                Rivet.Instance.Query("select OID from Osoba")
                .Sort<Osoba>(SortDirection.Asc, a => a.DatNarozeni);

        foreach (Osoba osoba in osoby.Cast<Osoba>())
                Console.WriteLine(
                        string.Format("{0} OTEC: {1} MATKA: {2}",
                                osoba,
                                osoba.Otec,
                                //osoba.Otec.IsPersistent ? osoba.Otec : null,
                                osoba.Matka
                                //osoba.Matka.IsPersistent ? osoba.Matka : null
                        ));

        Console.ReadLine();
}
```

Dostaneme:



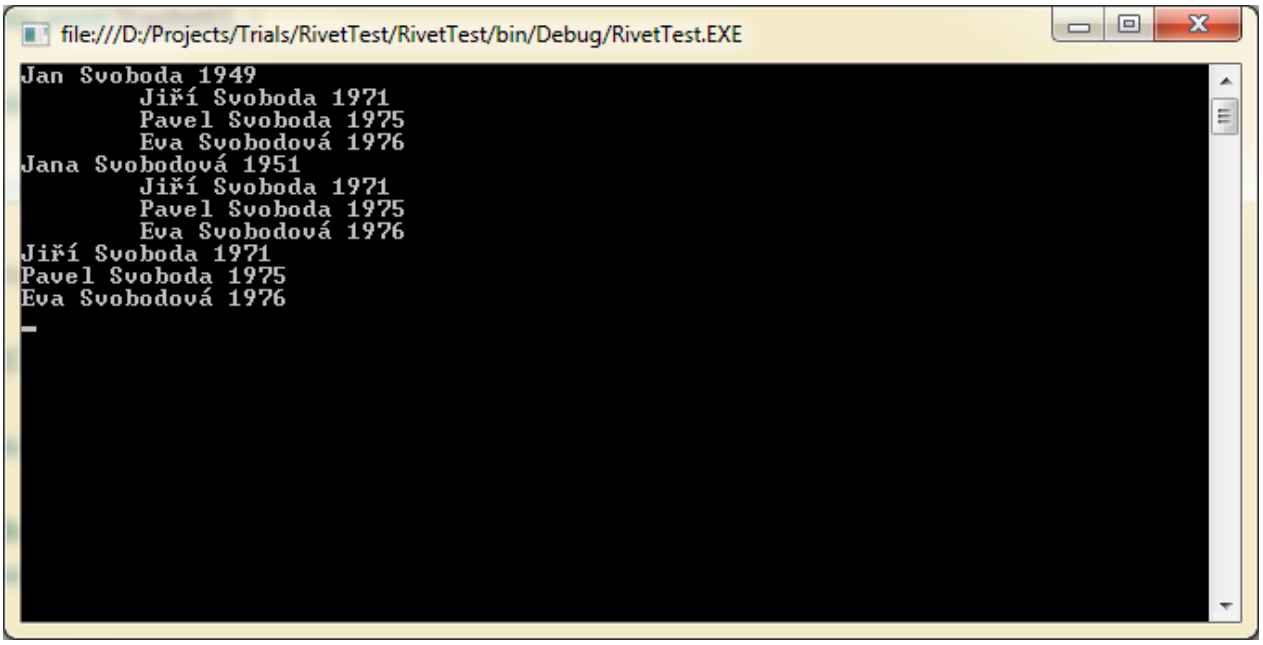
- Podivné hodnoty „00000000-0000-0000-0000-000000000000“ jsou způsobeny tím, že objekt **Proxy** vrací v metodě „**ToString**“ implicitně textovou podobu svého **OID**. A protože příslušné osoby nemají v databázi zavedeny své rodiče (a tedy není zde metoda „**ToString**“ přetížena), vrací systém místo nich prázdný objekt **Proxy** (a jeho prázdný **OID**).
- Metodu **Zobrazit** můžeme vylepšit tak, že příslušné řádky nahradíme těmi zakomentovanými. Vlastnost **IsPersistent** nám říká, jestli je daný objekt uložen v databázi (true) nebo se nachází pouze v paměti.

Zobrazení zpětných vazeb

Zpětné vazby **Nx1** jsou reprezentovány seznamem **ProxyList**.
Upravíme metodu **Zobrazit** tak, aby nám u každé osoby vypsala všechny její děti.

```
private static void Zobrazit()
{
        ProxyList osoby =
                Rivet.Instance.Query("select OID from Osoba")
                .Sort<Osoba>(SortDirection.Asc, a => a.DatNarozeni);

        foreach (Osoba osoba in osoby.Cast<Osoba>())
        {
                Console.WriteLine(osoba.ToString());    //vypíšeme osobu...
                //a pod ní její děti, odsazené o jeden tabelátor
                foreach (Osoba dite in osoba.GetDeti().Cast<Osoba>())
                        Console.WriteLine(string.Format("\t{0}", dite));
        }

        Console.ReadLine();
}
```

Dostaneme:

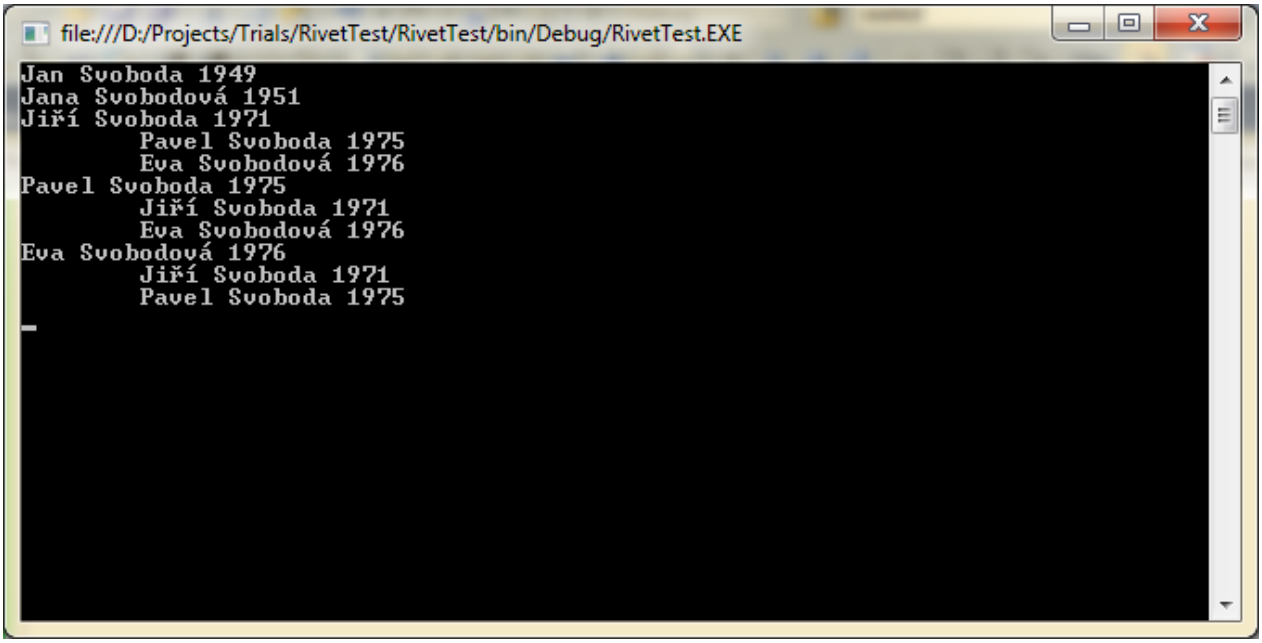


Zobrazení sourozenců

Upravíme metodu **Zobrazit** tak, aby nám u každé osoby vypsala všechny její sourozence.

```
private static void Zobrazit()
{
        ProxyList osoby =
                Rivet.Instance.Query("select OID from Osoba")
                .Sort<Osoba>(SortDirection.Asc, a => a.DatNarozeni);

        foreach (Osoba osoba in osoby.Cast<Osoba>())
        {
                Console.WriteLine(osoba.ToString());    //vypíšeme osobu...
                //a pod ní její sourozence, odsazené o jeden tabelátor
                foreach (Osoba sourozenec in osoba.GetSourozenci().Cast<Osoba>())
                        Console.WriteLine(string.Format("\t{0}", sourozenec));
        }

        Console.ReadLine();
}
```

Dostaneme:



Práce s vazbami NxN (vytvoření, zobrazení a rušení partnerství)

Nejprve přidáme osoby, které mají být partnery dětí z našeho příkladu, tedy dvě ženy a jednoho muže. Pro tento účel vytvoříme metodu **„PridatPartnery“**.

```
private static void PridatPartnery()
{
        Muz petrNovak =
                new Muz() { Jmeno = "Petr", Prijmeni = "Novák",
                DatNarozeni = new DateTime(1974, 4, 21) };
        Zena veraTicha =
                new Zena() { Jmeno = "Věra", Prijmeni = "Tichá",
                DatNarozeni = new DateTime(1977, 3, 3) };
        Zena pavlaMala =
                new Zena() { Jmeno = "Pavla", Prijmeni = "Malá",
                DatNarozeni = new DateTime(1976, 9, 5) };

        using (ContextTran ctx = Rivet.Instance.GetContextTran())
        {
                petrNovak.Bind(ctx).Update();
                veraTicha.Bind(ctx).Update();
                pavlaMala.Bind(ctx).Update();

                ctx.Commit();
        }
}
```

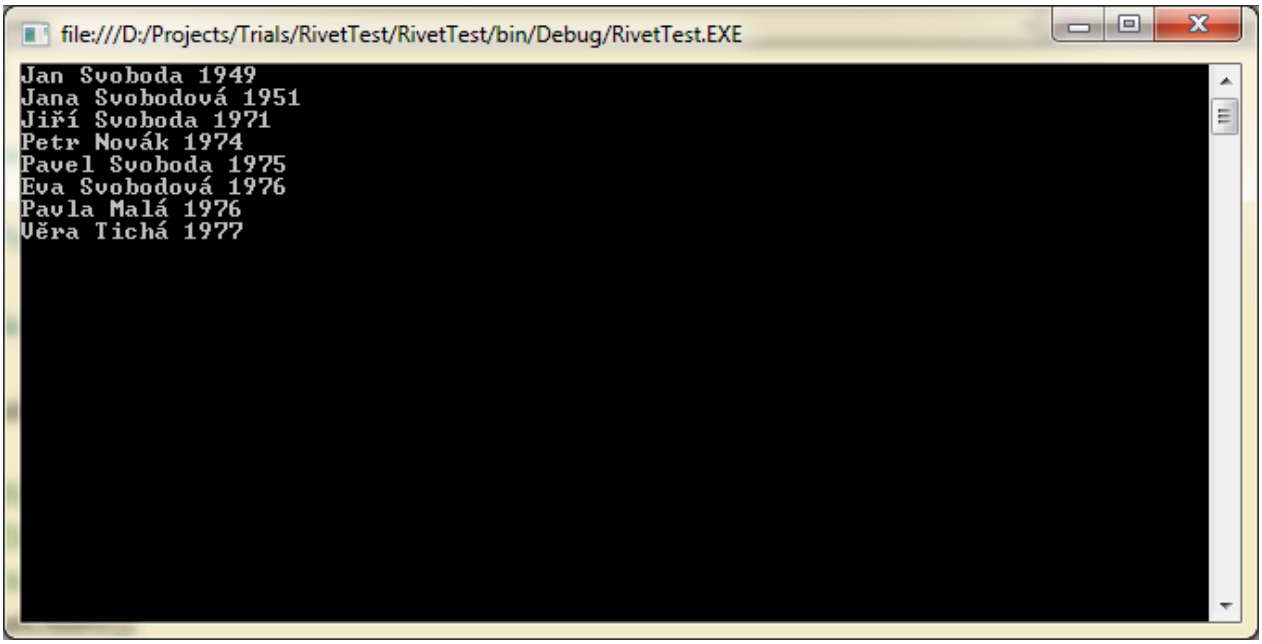
Z metody Zobrazit pro přehlednost opět odstraníme cyklus pro zobrazování dětí a necháme pouze základní smyčku:

```
        foreach (Osoba osoba in osoby.Cast<Osoba>())
                Console.WriteLine(osoba.ToString());    //vypíšeme osobu...
```

V hlavním programu pak zavoláme takovouto posloupnost:

```
static void Main(string[] args)
{
        Init();

        PridatPartnery ();
        Zobrazit ();
}
```

A dostaneme:



Nyní budeme potřebovat metodu pro načtení osoby, třeba podle zadaného jména. Nazvěme jí **„GetProxy“**:

- argument metody **„Query“** můžeme psát obdobně jako u metody string.Format(...)

```
private static Proxy GetProxy(Context ctx, string jmeno, string prijmeni)

{
        ProxyList list =
                ctx.GetProxyList()
                // argument metody Query můžeme psát obdobně jako u metody string.Format(...)
                .Query("select OID from Osoba where Jmeno='{0}' and Prijmeni='{1}'", jmeno, prijmeni);

        return list.Count > 0 ? list[0] : null;
}
```

Partnerství vytvoříme v metodě **„Oddej“**. Zde nastavíme ženě nové příjmení (pokud při volání zadáme nepovinný parametr „**novePrijmeni**“) a přiřadíme jí partnera metodou **„AddPartner“**:

```
private static void Oddej(Muz muz, Zena zena, string novePrijmeni = “”)
{
        if (novePrijmeni != “”)
                zena.Prijmeni = novePrijmeni;
        zena.AddPartner(muz);

        zena.Update();
}
```

V následující metodě **„OddatVsechny“** potom vytvoříme jak partnerství rodičů, tak i partnerství jejich tří dětí s nově přidanými osobami:

```
private static void OddatVsechny()
{
        using (ContextTran ctx = Rivet.Instance.GetContextTran())
        {
                //rodiče...
                Muz janSvoboda = GetProxy(ctx, "Jan", "Svoboda").Cast<Muz>();
                Zena janaSvobodova = GetProxy(ctx, "Jana", "Svobodová").Cast<Zena>();
                Oddat(ctx, janSvoboda, janaSvobodova);
                //syn...
                Muz jiriSvoboda = GetProxy(ctx, "Jiří", "Svoboda").Cast<Muz>();
                Zena veraTicha = GetProxy(ctx, "Věra", "Tichá").Cast<Zena>();
                Oddat(ctx, jiriSvoboda, veraTicha, "Svobodová");
                //syn...
                Muz pavelSvoboda = GetProxy(ctx, "Pavel", "Svoboda").Cast<Muz>();
                Zena pavlaMala = GetProxy(ctx, "Pavla", "Malá").Cast<Zena>();
                Oddat(ctx, pavelSvoboda, pavlaMala, "Svobodová");
                //dcera...
                Zena evaSvobodova = GetProxy(ctx, "Eva", "Svobodová").Cast<Zena>();
                Muz petrNovak = GetProxy(ctx, "Petr", "Novák").Cast<Muz>();
                Oddat(ctx, petrNovak, evaSvobodova, "Nováková");

                ctx.Commit();
        }
}
```

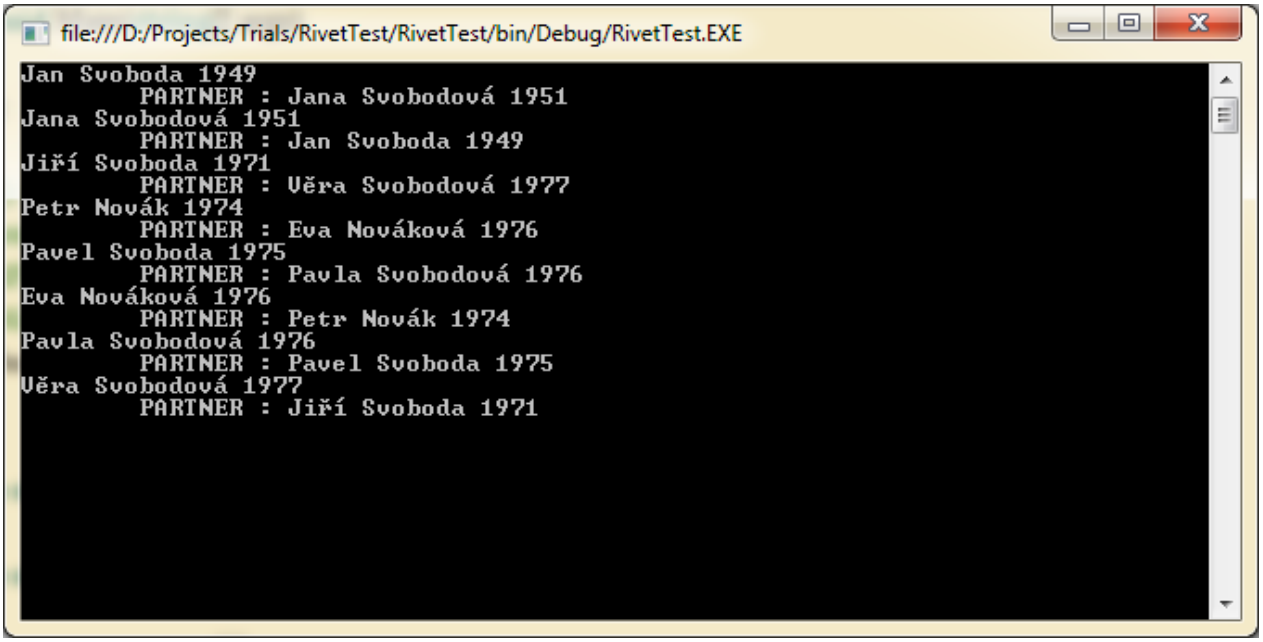
Ještě upravíme metodu **Zobrazit** tak, aby nám ke každé osobě vypsala všechny partnery:

```
private static void Zobrazit()
{
        ProxyList osoby =
                Rivet.Instance.Query("select OID from Osoba")
                .Sort<Osoba>(SortDirection.Asc, a => a.DatNarozeni);

        foreach (Osoba osoba in osoby.Cast<Osoba>())
        {
                Console.WriteLine(osoba.ToString());

                if (osoba is Muz)
                        foreach (Zena zena in osoba.LightCast<Muz>().GetPartneri().Cast<Zena>())
                                Console.WriteLine(string.Format("\tPARTNER : {0}", zena));

                if (osoba is Zena)
                        foreach (Muz muz in osoba.LightCast<Zena>().GetPartneri().Cast<Muz>())
                                Console.WriteLine(string.Format("\tPARTNER : {0}", muz));
        }

        Console.ReadLine();
}
```

V hlavním programu pak zavoláme tuto posloupnost:

```
static void Main(string[] args)
{
        Init();
        OddatVsechny ();
        Zobrazit ();
}
```

A dostaneme:



Pokud bychom chtěli partnerství smazat, mohli bychom použít metodu **„RozvestVsechny“**:

```
private static void RozvestVsechny()
{
        using (ContextTran ctx = Rivet.Instance.GetContextTran())
        {
                ProxyList muzi = ctx.GetProxyList().Query("select OID from Muz");

                foreach (Muz muz in muzi.Cast<Muz>())
                        foreach (Zena zena in muz.GetPartneri().Cast<Zena>())
                                muz.RemovePartner(zena);

                ctx.Commit();
        }
}
```

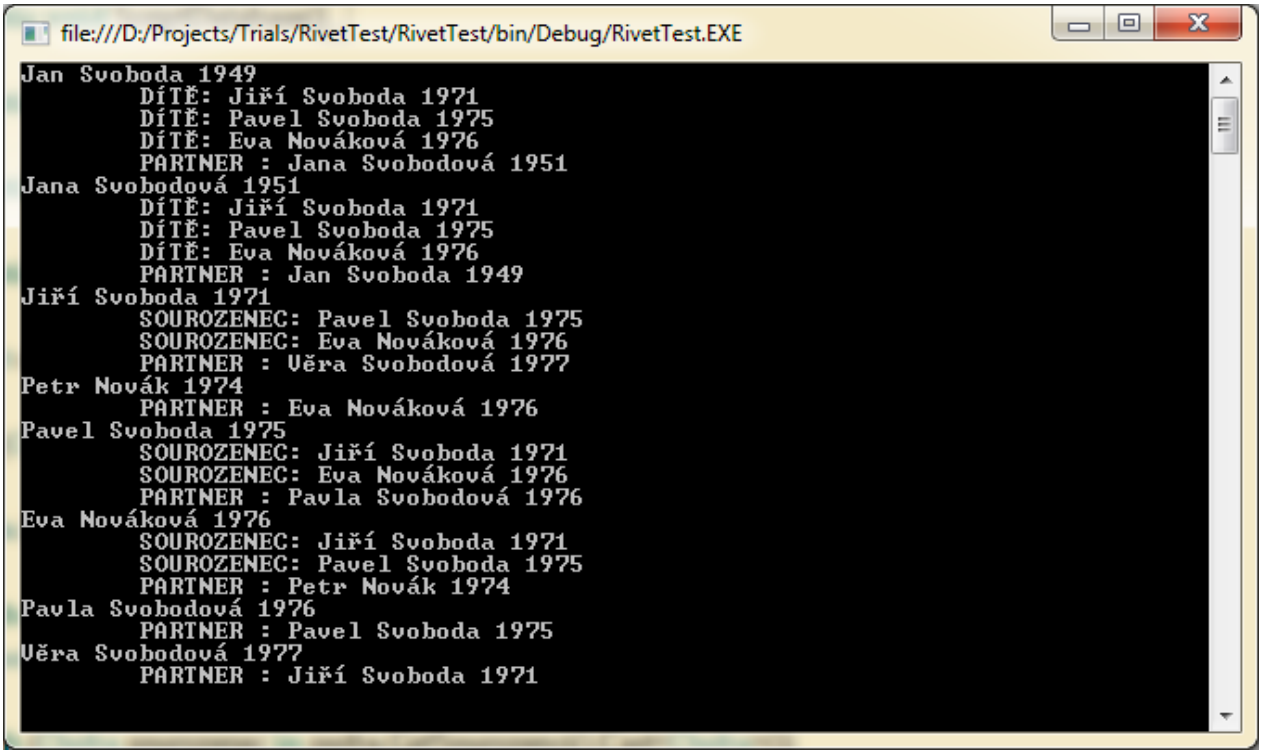
Pozn.:
Partnerství je symetrický vztah, čili je jedno, ze které strany jej přidáváme nebo odebíráme.

Celkové přehledné zobrazení

Upravme metodu **Zobrazit** takto:

```
private static void Zobrazit()
{
        ProxyList osoby =
                Rivet.Instance.Query("select OID from Osoba")
                .Sort<Osoba>(SortDirection.Asc, a => a.DatNarozeni);

        foreach (Osoba osoba in osoby.Cast<Osoba>())
        {
                Console.WriteLine(osoba.ToString());

                foreach (Osoba dite in osoba.GetDeti().Cast<Osoba>())
                        Console.WriteLine(string.Format("\tDÍTĚ: {0}", dite));
                foreach (Osoba sourozenec in osoba.GetSourozenci().Cast<Osoba>())
                        Console.WriteLine(string.Format("\tSOUROZENEC: {0}", sourozenec));
                if (osoba is Muz)
                        foreach (Zena zena in osoba.LightCast<Muz>().GetPartneri().Cast<Zena>())
                                Console.WriteLine(string.Format("\tPARTNER : {0}", zena));
                if (osoba is Zena)
                        foreach (Muz muz in osoba.LightCast<Zena>().GetPartneri().Cast<Muz>())
                                Console.WriteLine(string.Format("\tPARTNER : {0}", muz));
        }

        Console.ReadLine();
}
```

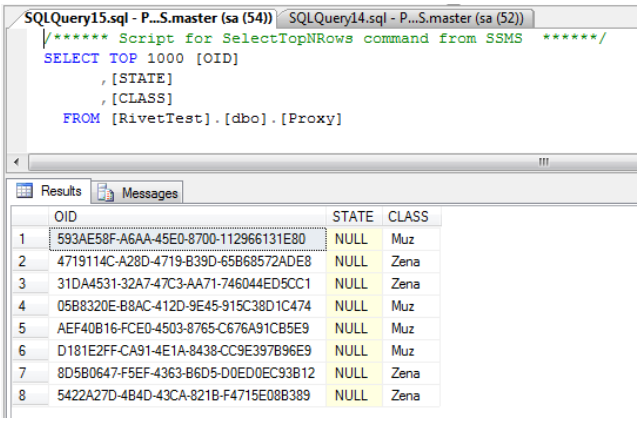
A dostaneme:



Jak to vypadá v databázi…

Tabulka Proxy:

- Sloupec **OID** představuje „**primární klíč**“ a pomocí něho je provedeno hierarchické propojení tabulek (**dědičnost**). Jednoznačně identifikuje objekt.
- Vidíme, že sloupec **STATE** není vyplněn. Je to proto, že tento sloupec je vyplněn vždy pouze v té tabulce, která odpovídá typu objektu.
- Sloupec **CLASS** obsahuje název třídy objektu. Pokud tedy máme k dispozici pouze objekt „**Proxy**“ a neznáme jeho skutečný typ, lze jej odsud zjistit. To je také primárním posláním této tabulky. Nic nám ovšem nebrání vytvořit objekt přímo typu „**Proxy**“. Jeho záznam by potom ležel pouze v této tabulce.



SQLQuery15.sql - P...S.master (sa (54)) | SQLQuery14.sql - P...S.master (sa (52))

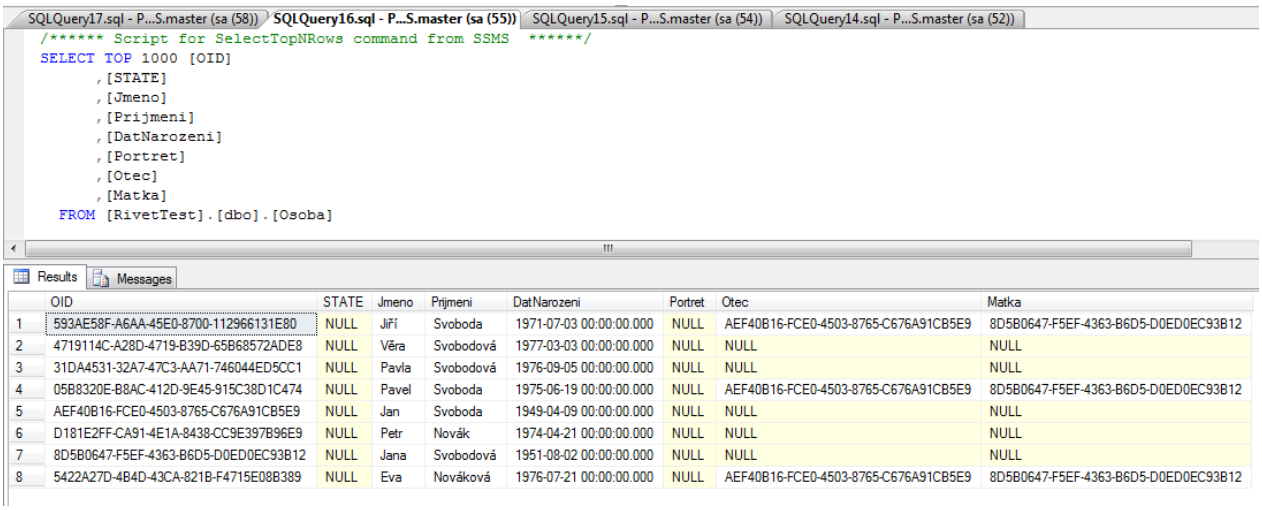
```
/****** Script for SelectTopNRows command from SSMS  ******/
SELECT TOP 1000 [OID]
      ,[STATE]
      ,[CLASS]
  FROM [RivetTest].[dbo].[Proxy]
```

Results | Messages

	OID	STATE	CLASS
1	593AE58F-A6AA-45E0-8700-112966131E80	NULL	Muz
2	4719114C-A28D-4719-B39D-65B68572ADE8	NULL	Zena
3	31DA4531-32A7-47C3-AA71-746044ED5CC1	NULL	Zena
4	05B8320E-B8AC-412D-9E45-915C38D1C474	NULL	Muz
5	AEF40B16-FCE0-4503-8765-C676A91CB5E9	NULL	Muz
6	D181E2FF-CA91-4E1A-8438-CC9E397B96E9	NULL	Muz
7	8D5B0647-F5EF-4363-B6D5-D0ED0EC93B12	NULL	Zena
8	5422A27D-4B4D-43CA-821B-F4715E08B389	NULL	Zena

Tabulka Osoba:

- Kromě sloupců **OID** a **STATE** zde vidíme sloupce, odpovídající vlastnostem definovaným ve třídě „**Osoba**“.
- Ani zde není sloupec **STATE** vyplněn – žádný objekt nebyl vytvořen jako typ „**Osoba**“.
- Vlastnost „**Portret**“ jsme v našem příkladu nepoužívali.
- Vidíme, že sloupce „**Otec**“ a „**Matka**“, odkazují na příslušné objekty prostřednictvím jejich **OID** („**cizí klíče**“).



SQLQuery17.sql - P...S.master (sa (58)) | SQLQuery16.sql - P...S.master (sa (55)) | SQLQuery15.sql - P...S.master (sa (54)) | SQLQuery14.sql - P...S.master (sa (52))

```
/****** Script for SelectTopNRows command from SSMS  ******/
SELECT TOP 1000 [OID]
      ,[STATE]
      ,[Jmeno]
      ,[Prijmeni]
      ,[DatNarozeni]
      ,[Portret]
      ,[Otec]
      ,[Matka]
  FROM [RivetTest].[dbo].[Osoba]
```

Results | Messages

	OID	STATE	Jmeno	Prijmeni	DatNarozeni	Portret	Otec	Matka
1	593AE58F-A6AA-45E0-8700-112966131E80	NULL	Jiří	Svoboda	1971-07-03 00:00:00.000	NULL	AEF40B16-FCE0-4503-8765-C676A91CB5E9	8D5B0647-F5EF-4363-B6D5-D0ED0EC93B12
2	4719114C-A28D-4719-B39D-65B68572ADE8	NULL	Věra	Svobodová	1977-03-03 00:00:00.000	NULL	NULL	NULL
3	31DA4531-32A7-47C3-AA71-746044ED5CC1	NULL	Pavla	Svobodová	1976-09-05 00:00:00.000	NULL	NULL	NULL
4	05B8320E-B8AC-412D-9E45-915C38D1C474	NULL	Pavel	Svoboda	1975-06-19 00:00:00.000	NULL	AEF40B16-FCE0-4503-8765-C676A91CB5E9	8D5B0647-F5EF-4363-B6D5-D0ED0EC93B12
5	AEF40B16-FCE0-4503-8765-C676A91CB5E9	NULL	Jan	Svoboda	1949-04-09 00:00:00.000	NULL	NULL	NULL
6	D181E2FF-CA91-4E1A-8438-CC9E397B96E9	NULL	Petr	Novák	1974-04-21 00:00:00.000	NULL	NULL	NULL
7	8D5B0647-F5EF-4363-B6D5-D0ED0EC93B12	NULL	Jana	Svobodová	1951-08-02 00:00:00.000	NULL	NULL	NULL
8	5422A27D-4B4D-43CA-821B-F4715E08B389	NULL	Eva	Nováková	1976-07-21 00:00:00.000	NULL	AEF40B16-FCE0-4503-8765-C676A91CB5E9	8D5B0647-F5EF-4363-B6D5-D0ED0EC93B12

Tabulka Muz:

- V této tabulce jsou uvedeny pouze ty objekty, které byly vytvořeny jako typ „**Muž**“.
- Vidíme, že je zde vyplněn sloupec **STATE**. Tento sloupec obsahuje **GUID**, který je změněn na nový pokaždé, když je na objektu provedena metoda „**Update**“. Zde se při čtení objektu zjistí, jestli je v **Cache** aktuální záznam (hodnota STATE v **Cache** a v databázi se shoduje). Pokud je záznam v **Cache** aktuální, použije se (z databáze se načte pouze záznam z této tabulky – právě kvůli zjištění hodnoty **STATE**) . Pokud záznam v **Cache** aktuální není, načte se celá hierarchie propojených tabulek dle dědičnosti a **Cache** se aktualizuje.
- V této tabulce jsou sloupce, odpovídající vlastnostem, kterými třída „**Muz**“ rozšiřuje třídu „**Osoba**“. Jsou to sloupce „**Povolani**“, „**Vousy**“ a „**Plat**“, které jsme v našem testu zatím nepoužili.

SQLQuery17.sql - P...S.master (sa (58)) | SQLQuery16.sql - P...S.master (sa (55)) | SQLQuery15.sql - P...S.master (sa (54)

```
/****** Script for SelectTopNRows command from SSMS  ******/
SELECT TOP 1000 [OID]
      ,[STATE]
      ,[Povolani]
      ,[Vousy]
      ,[Plat]
  FROM [RivetTest].[dbo].[Muz]
```

Results | Messages

	OID	STATE	Povolani	Vousy	Plat
1	593AE58F-A6AA-45E0-8700-112966131E80	12A3A7D2-7A88-4381-B782-A2D5972EBB43	NULL	0	NULL
2	05B8320E-B8AC-412D-9E45-915C38D1C474	E2D84139-3E4E-4683-AF93-4517F418609C	NULL	0	NULL
3	AEF40B16-FCE0-4503-8765-C676A91CB5E9	5C5CD995-97FF-4956-A947-5E75BA7219F0	NULL	0	NULL
4	D181E2FF-CA91-4E1A-8438-CC9E397B96E9	5374F0AB-0855-4EBD-9F7C-99FA468889A2	NULL	0	NULL

Tabulka Zena:

- V této tabulce jsou uvedeny pouze ty objekty, které byly vytvořeny jako typ „**Zena**“.
- Vidíme, že i zde je vyplněn sloupec **STATE** (viz. výše).
- V této tabulce jsou sloupce, odpovídající vlastnostem, kterými třída „**Zena**“ rozšiřuje třídu „**Osoba**“. Jsou to sloupce „**Rozena**“ a „**ObvodPasu**“, které jsme v našem testu zatím nepoužili.

SQLQuery18.sql - P...S.master (sa (59)) | SQLQuery17.sql - P...S.master (sa (58)) | SQLQuery16.sql - P...S.master (sa (5

```
/****** Script for SelectTopNRows command from SSMS  ******/
SELECT TOP 1000 [OID]
      ,[STATE]
      ,[Rozena]
      ,[ObvodPasu]
  FROM [RivetTest].[dbo].[Zena]
```

Results | Messages

	OID	STATE	Rozena	ObvodPasu
1	4719114C-A28D-4719-B39D-65B68572ADE8	D5E04ACE-96C7-427B-B9CA-E151417CE93F	NULL	0
2	31DA4531-32A7-47C3-AA71-746044ED5CC1	E0F06D65-83F6-46E3-B035-EFDB32E6DA06	NULL	0
3	8D5B0647-F5EF-4363-B6D5-D0ED0EC93B12	B6C82829-E97F-46C2-889D-7D5812CE2E73	NULL	0
4	5422A27D-4B4D-43CA-821B-F4715E08B389	970B4E7F-808C-41EA-B066-2F00D7F41BC8	NULL	0

Tabulka Partnerstvi:

- Tabulka „**Partnerstvi**“ je vazební a obsahuje pouze dva sloupce, odkazující na **OID** příslušných svázaných objektů.

SQLQuery19.sql - P...S.master (sa (60)) | SQLQuery18.sql - P...S.master (sa (59)) | SQLQuery1

```
/****** Script for SelectTopNRows command from SSMS  ******/
SELECT TOP 1000 [Muz]
      ,[Zena]
  FROM [RivetTest].[dbo].[Partnerstvi]
```

Results | Messages

	Muz	Zena
1	AEF40B16-FCE0-4503-8765-C676A91CB5E9	8D5B0647-F5EF-4363-B6D5-D0ED0EC93B12
2	593AE58F-A6AA-45E0-8700-112966131E80	4719114C-A28D-4719-B39D-65B68572ADE8
3	05B8320E-B8AC-412D-9E45-915C38D1C474	31DA4531-32A7-47C3-AA71-746044ED5CC1
4	D181E2FF-CA91-4E1A-8438-CC9E397B96E9	5422A27D-4B4D-43CA-821B-F4715E08B389

Co v příkladu nebylo...

Atributy

Již známe atributy **„MaxLength“** a **„ForeignKey“**.
Existuje ještě atribut „**NotPersistent“**. Tímto atributem můžeme označit pole, které nechceme uchovávat v databázi a má zůstat pouze jako paměťové.

Cache

Systém používá **Cache** – v paměti si udržuje jednou načtené objekty. Aby byly vlastnosti v **Cache** vždy aktuální, používá se mechanismus, popsaný výše.
Cache používá automatický systém odmazávání, nastavitelný pomocí dvou vlastností:

- **„CacheLatency“** – doba v sekundách, po kterou je záznam v **Cache** „chráněn“. Měří se od posledního okamžiku, kdy byl záznam použit. Defaultně je **600** (10min). Hodnota 0 odmazávání vypne.
- **„CacheItemsToClear“** – počet záznamů v **Cache**, po jehož překročení se odmažou „nechráněné“ záznamy. Defaultně je **1000**.

Defaultní hodnoty lze změnit při inicializaci (zde jsou v textové podobě), např.:

```
Rivet.InitialCacheLatency = "60";
Rivet.InitialCacheItemsToClear = "500";
```

Dále je lze měnit kdykoli na instanci třídy **„Rivet“**, např.:

```
Rivet.Instance.CacheLatency = 60;
Rivet. Instance.CacheItemsToClear = 500;
```

Proxy

Pokud známe **OID**, můžeme objekt načíst třeba takto:

```
Guid OID = new Guid("593AE58F-A6AA-45E0-8700-112966131E80");
Proxy proxy = new Proxy(OID);
using (Context ctx = Rivet.Instance.GetContext())
{
        proxy.Bind(ctx);
        Muz muz = proxy.Cast<Muz>();
        ...
}
```

Zkráceně:

```
using (Context ctx = Rivet.Instance.GetContext())
{
        Muz muz =
                new Proxy(new Guid("593AE58F-A6AA-45E0-8700-112966131E80"))
                .Bind(ctx).Cast<Muz>();
        ...
}
```

Třída **„Proxy“** má pro zjednodušení práce také tyto statické metody:

- **„New“** – tuto metodu můžeme s výhodou použít pro přidávání nových objektů, např.:

```
using (ContextTran ctx = Rivet.Instance.GetContextTran())
{
        Muz muz = Proxy.New<Muz>(ctx);
        muz.Jmeno = "Aleš";
        muz.Prijmeni = "Novák";
        muz.DatNarozeni = new DateTime(1977, 3, 8);
        muz.Update();
        ctx.Commit();
}
```

- **„Cast“** – tuto metodu můžeme s výhodou použít pro update existujících objektů, např.:

```
using (ContextTran ctx = Rivet.Instance.GetContextTran())
{
        Guid OID = new Guid("889409E4-3517-4044-8E1A-A21CB537B7C6");
        Muz muz = Proxy.Cast<Muz>(OID, ctx);
        muz.Jmeno = "Alois";
        muz.Update();
        ctx.Commit();
}
```

  Všiměme si, že touto metodou lze nahradit načítání objektů pomocí konstruktoru třídy **„Proxy“**, jak to bylo popsáno výše na této stránce.

- **„Delete“** – tuto metodu můžeme s výhodou použít pro smazání objektů, jejichž **OID** známe, např.:

```
using (ContextTran ctx = Rivet.Instance.GetContextTran())
{
        Guid OID = new Guid("889409E4-3517-4044-8E1A-A21CB537B7C6");
        Proxy.Delete(OID, ctx);
        ctx.Commit();
}
```

Ekvivalence objektů:

Třída **„Proxy“** má přetížené operátory rovnosti a nerovnosti (také metody **„Equals“** a **„GetHashCode“**) tak, že dva objekty jsou vyhodnoceny jako shodné právě tehdy, rovnají-li se jejich **OID** – tedy jedná-li se o shodný záznam. Ekvivalence se nevyhodnocuje na základě totožnosti instancí paměťového objektu.

ProxyList

Třída **„ProxyList“** je asi nejzajímavější a také nabízí nejvíce funkčností, které nám výše uvedený příklad neukázal.
Výše jsme použili kód:

```
ProxyList list =
        ctx.GetProxyList()
        .Query("select OID from Osoba where Jmeno='{0}' and Prijmeni='{1}'", jmeno, prijmeni);
Proxy proxy = list.Count > 0 ? list[0] : null;
```

Třída **„ProxyList“** nám nabízí použití **parametrů**:

```
ProxyList list = ctx.GetProxyList();
list.Sql = "select OID from Osoba where Jmeno=@Jmeno and Prijmeni=@Prijmeni";
list.AddParameter("@Jmeno", jmeno, SqlDbType.VarChar);
list.AddParameter("@Prijmeni", prijmeni, SqlDbType.VarChar);
list.Query();
Proxy proxy = list.Count > 0 ? list[0] : null;
```

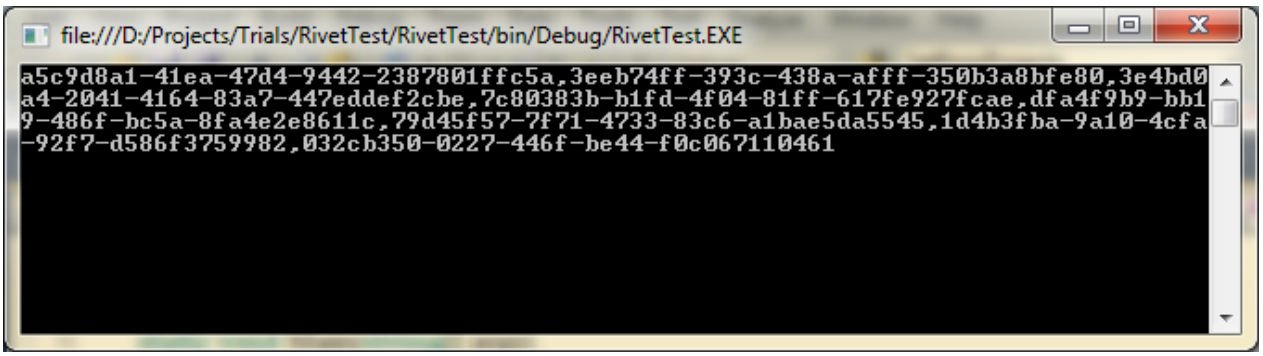
Seznam parametrů lze smazat metodou **„ClearParameters“**:

```
list.ClearParameters();
```

U kratších seznamů se může hodit možnost „odložit si jejich obsah“ do textové proměnné prostřednictvím metody **„ToString“**:

```
static void Main(string[] args)
{
        Init();

        ProxyList list = Rivet.Instance.Query("select OID from Osoba");
        string oids = list.ToString();
        Console.WriteLine(oids);
        Console.ReadLine();
}
```

Vypíše seznam hodnot **OID**, odpovídající načteným objektům, oddělených čárkami:



Naopak, z textového seznamu hodnot **OID** oddělených čárkami, lze vytvořit objekt **„ProxyList“** pomocí statické metody **„Create“**:

```
static void Main(string[] args)
{
        Init();

        ProxyList list1 = Rivet.Instance.Query("select OID from Osoba");
        string oids = list1.ToString();

        using (Context ctx = Rivet.Instance.GetContext())
        {
                ProxyList list2 = ProxyList.Create(ctx, oids);
                foreach (Osoba osoba in list2.Cast<Osoba>())
                        Console.WriteLine(osoba.ToString());
        }
        Console.ReadLine();
}
```

Ačkoli seznam „**list2“** vznikl pouze na základě stringu **„oids**, načetl se seznam odpovídajících osob:

Množinové operace s objekty „ProxyList“

Výše jsme se již setkali s operacemi **„Union“** a **„Subtract“**.

- **Union**(ProxyList list) – sjednocení – k objektům instance přidá objekty parametru
- **Add**(Proxy proxy) – přidání - k objektům instance přidá objekt parametru
- **Subtract**(ProxyList list) – odebrání - ze seznamu objektů instance odebere objekty parametru
- **Subtract**(Proxy proxy) – odebrání - ze seznamu objektů instance odebere objekt parametru
- **Intersection**(ProxyList list) – průnik - v seznamu objektů instance ponechá jen ty objekty, které obsahuje i parametr

Pozn.:

1. Tyto operace probíhají „v odpojeném stavu“, tzn. že se v nich v tomto okamžiku nekontroluje skutečná existence objektů v databázi.
2. Při těchto operacích nevzniká duplicita objektů.

Poznatky:

- jádrem třídy **„ProxyList“** je prostý seznam hodnot **OID** objektů => jednoduchost, odpojenost
- pokud používáme **stránkování**, seznam obsahuje **OID** pouze objektů z aktuální stránky

Stránkování

Třída **„ProxyList“** obsahuje sadu metod a vlastností, podporujících snadné **stránkování**. Pokud máme seznam stránkovat, musíme ho nějak **setřídit**. Proto má metoda „**Page**“ argumenty pro třídění.
Příklad stránkovaného seznamu (stránkovací metody a vlastnosti jsou zvýrazněny):

```
private static void StrankovanySeznam()

{
        ProxyList list = Rivet.Instance.Query("select OID from Osoba");

        list.PageLength = 3;
        list.ToFirstPage();
        while (true)
        {
                Console.WriteLine(string.Format("---{0}/{1}---", list.PageIndex + 1, list.PageCount));

                ProxyList page = list.Page<Osoba>(SortDirection.Asc, a => a.DatNarozeni);
                foreach (Osoba osoba in page.Cast<Osoba>())
                        Console.WriteLine(osoba.ToString());

                if (list.IsLastPage)
                        break;

                list.ToNextPage();
        }

        Console.ReadLine();
}
```

Dostaneme toto:

Filtrování – třída FilterHelper

Pro snažší filtrování je implementována třída „**FilterHelper“**. Umožňuje jednoduše provádět základní filtrování.

Příklad:

```
private static void ZobrazitFiltrovane()
{
        //filtrujeme nad seznamem osob...
        FilterHelper filter = new FilterHelper("select OID from Osoba");
        //osoby, jejichž jméno začíná na „P“...
        filter.AddString<Osoba>(FilterType.StartsWith, "P", a => a.Jmeno);
        //...a datum narození je starší než rok 1976
        filter.AddDateTime<Osoba>(FilterType.Less, new DateTime(1976, 1, 1), a => a.DatNarozeni);

        ProxyList osoby = filter.GetProxyList();
        osoby.Query();

        foreach (Osoba osoba in osoby.Cast<Osoba>())
                Console.WriteLine(osoba.ToString());

        Console.ReadLine();
}
```

Metody:

- **Add**
- **AddClass**
- **AddString**
- **AddDateTime**
- **AddInt**
- **AddDouble**
- **AddDecimal**

Typy filtrace:

- **Neutral** – nedělá nic
- **Equal** – rovnost
- **NotEqual** – nerovnost
- **Less** – menší než
- **LessOrEqual** – menší nebo rovno
- **Greater** – větší než
- **GreaterOrEqual** – větší nebo rovno
- **StartsWith** – začíná na (má smysl pro stringy)
- **EndsWith** – končí na (má smysl pro stringy)
- **Like** – obsahuje na (má smysl pro stringy)
- **Class** – vybírá pouze objekty daného typu (viz. níže)

Kdybychom chtěli filtrovat třeba přes vlastnost „**Plat**“, která se ale nachází v tabulce „**Muz**“, nikoli tabulce „**Osoba**“, došlo by k chybě, přestože i objekty typu „**Muz**“ jsou osobami.
Abychom mohli filtrovat přes libovolné vlastnosti objektu (které mohou ležet v různých propojených tabulkách), nabízí třída **„Rivet“** statickou generickou metodu „**GetQuery“**.

Příklad:

```
var sql = Rivet.GetQuery<Osoba>();
```

Tato metoda za nás vytvoří kompletní dotaz **SELECT** s propojením tabulek. V našem případě třídy „**Osoba**“ to bude:

```
select Osoba.OID from Osoba
    left join Muz on Muz.OID=Osoba.OID
    left join Zena on Zena.OID=Osoba.OID
```

Tato metoda nám tedy vždy poskytne základ, nad nímž může třída **„FilterHelper“** pracovat v plném rozsahu.
Pro zjednodušení pak třída **„FilterHelper“** nabízí statickou generickou metodu **„Create“**, která vytvoří instanci filtru s již takto vytvořeným selektem.

Příklad:

```
private static void ZobrazitFiltrovane()
{
    FilterHelper filter = FilterHelper.Create<Osoba>();
    filter.AddString<Osoba>(FilterType.StartsWith, "P", a => a.Jmeno);
    filter.AddDateTime<Osoba>(FilterType.Less, new DateTime(1975, 12, 31), a => a.DatNarozeni);
    //...a plat muže je vyšší než 10 000...
    filter.AddDecimal<Muz>(FilterType.Greater, 10000, a => a.Plat);

    ProxyList osoby = filter.GetProxyList();
    osoby.Query();

    foreach (Osoba osoba in osoby.Cast<Osoba>())
        Console.WriteLine(osoba.ToString());

    Console.ReadLine();
}
```

Filtrování přes typ objektu (zde vyfiltruje pouze ženy):

```
private static void ZobrazitFiltrovane()
{
    FilterHelper filter = FilterHelper.Create<Osoba>();
    filter.AddClass<Zena>();
    //je totéž jako
    //filter.Add(typeof(Zena), FilterType.Class, true, string.Empty);
    ProxyList osoby = filter.GetProxyList();
    osoby.Query();
    foreach (Osoba osoba in osoby.Cast<Osoba>())
        Console.WriteLine(osoba.ToString());

    Console.ReadLine();
}
```