TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: N2612– Elektrotechnika a informatika

Studijní obor: 1802T007 Informační technologie

Modulární systém pro vytváření a generování reportů z dat měřicích přístrojů

Modular system for design and generation of data reports from measuring instrument

Diplomová práce

Autor: **Pavel Vencl**

Vedoucí diplomové práce: **Ing. Jan Kraus**

Konzultant: **Ing. Tomáš Tobiška**

Liberec 20.5.2011

Zadání

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé DP a prohlašuji, že **s o u h l a s í m** s případným užitím mé diplomové práce (prodej, zapůjčení apod.).

Diplomové práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum:

Podpis:

Poděkování

Zde bych chtěl poděkovat svým rodičům, za podporu během celého studia. Mé přítelkyni za nekonečnou trpělivost, kterou se mnou měla v dobách tvorby této práce a panu Ing. Janu Krausovi za toto téma a pomoc s jeho vypracováním.

Abstrakt

Práce se zaměřuje na možnost tvorby modulární aplikace s využitím technologie NET Framework a jazyka C#. Dále je zaměřena na praktické využití tohoto modulárního systému při tvorbě reportů z údajů, naměřených přístroji společnosti KMB. Konkrétně na tvorbu manageru, který se stará o kompletní správu modulů. Také na způsob integrace tohoto manageru do základní aplikace, pojmenované ENVIS, tak aby bylo v aplikaci potřeba udělat co nejméně zásahů. A v neposlední řadě se práce zabývá samotnými ukázkovými moduly generující reporty.

Klíčová slova: plugin, plugin manager, report, rozhraní, reflexe

Abstrakt (en)

Focus of this thesis is given to making modular applications with NET Framework technology and C# programing language. Beside this, focus is also given to practical usage of this modular system for making of reports from data measured by KMB measuring instruments. Be specific for making of manager, which is taking care of administration of modules. Also to way of integration of this manager to basic application (called ENVIS) thus to make application without changes as much as possible. And last but not least this thesis put mind to examplary modules which are generatintg reports itself.

Key words: plugin, plugin manager, report, interface, reflection

Obsah

Seznam obrázků

Seznam kódů

1. Úvod

Důvodem vzniku této práce byl požadavek společnosti KMB. systems s.r.o. na vytvoření modulárního systému pro program ENVIS. Společnost KMB. systems s.r.o. se zabývá vývojem a výrobou měřících přístrojů, převážně se jedná o elektroměry používané ve výrobním odvětví. Data z přístrojů lze velice snadno stáhnout do počítače do připravené databáze a je možné s nimi dále pracovat, upravovat, zobrazovat různé grafy, tabulky a další data.

Tvořený modulární systém a následně tvořené moduly, mají v první řadě sloužit jako nástroj pro tvorbu reportů a jiných grafických zobrazení průběhu měřených dat. Uživatelé tak jednoduše získají údaje, které by jinak museli dohledávat v tabulkách a datech. Také získají možnost naměřené údaje zobrazit a vytisknou. Toho lze využít například, pokud chtějí změřené údaje doložit k dokumentům, archivovat, případně si do vytištěných dokumentů dopsat nějaké poznámky nebo zakreslit doplňující údaje.

Cílem práce je vytvořit modulární systém schopný práce s moduly pro generování reportů. Jeho integrace do prostředí programu ENVIS, tak aby nedocházelo k žádnému negativnímu ovlivnění základního programu. Ať už z hlediska funkčnosti nebo z hlediska omezování rychlosti. Následně je pak plánována tvorba několika ukázkových modulů pro názornou prezentaci vzhledu výsledných reportů.

Zamýšleny jsou tři ukázkové reporty. RetisType, což je report inspirovaný předchozí verzí softwaru. Je v něm soupis všech hlavních měřených hodnot, jejich maxim a minim a také maxim a minim klouzavých průměrů pro některé veličiny. Druhým reportem by měl být report, ve kterém budou zobrazeny události přepětí, podpětí a výpadků napětí. Posledním by měl být report, který vezme veškeré přístroje, které jsou uloženy v databázi a vypíše všechny konfigurace nastavení pro každý z přístrojů. Množství reportů se může podle časových možností rozrůst.

2. Teoretický rozbor zadaného úkolu

Úkolem bylo prostudovat databázi společnosti KMB systems s.r.o. a rozšířit software ENVIS o podporu modulů. Tyto moduly budou sloužit pro tvorbu reportu a usnadní tak práci s vyhodnocováním dat. Dalším z úkolů byla tvorba samotných ukázkových modulů, které tvoří reporty. Tyto reporty mají na starost vyhodnocování a prezentování měřených dat a tím usnadnit práci uživatelům vyhodnocujícím data. Ušetří se tak nejen spousta práce, která by byla vynaložena při počítání různých hodnot, ale i čas věnovaný této činnosti. Zvláště pokud by se vyhodnocování dat opakovalo nebo bylo vyhodnocováno více dat stejným způsobem.

K využití modulárního systému se přistoupilo i z důvodu distribuce přístrojů a softwaru více odběratelům. Každý z odběratelů chce svoje výsledky prezentovat jiným způsobem ať už formou nebo úpravou dat. Kdyby se měly veškeré požadavky odběratelů pokrýt v základní aplikaci, znamenalo by to neustálé doplňování aplikace o nové funkce vedoucí k zbytečně velkému softwarovému balíku. Následně by bylo nutno aplikaci znovu distribuovat a instalovat.

Nejtěžším krokem je navrhnout, jakým způsobem bude aplikace s moduly komunikovat, předávat jim data a doplňující parametry. Tento problém se rozdělil do tří částí. První část se stará o komunikaci s aplikací, druhá část má na starosti správu modulů a předávání parametrů získaných z aplikace. V poslední části jsou pak samotné moduly a jejich napojení.

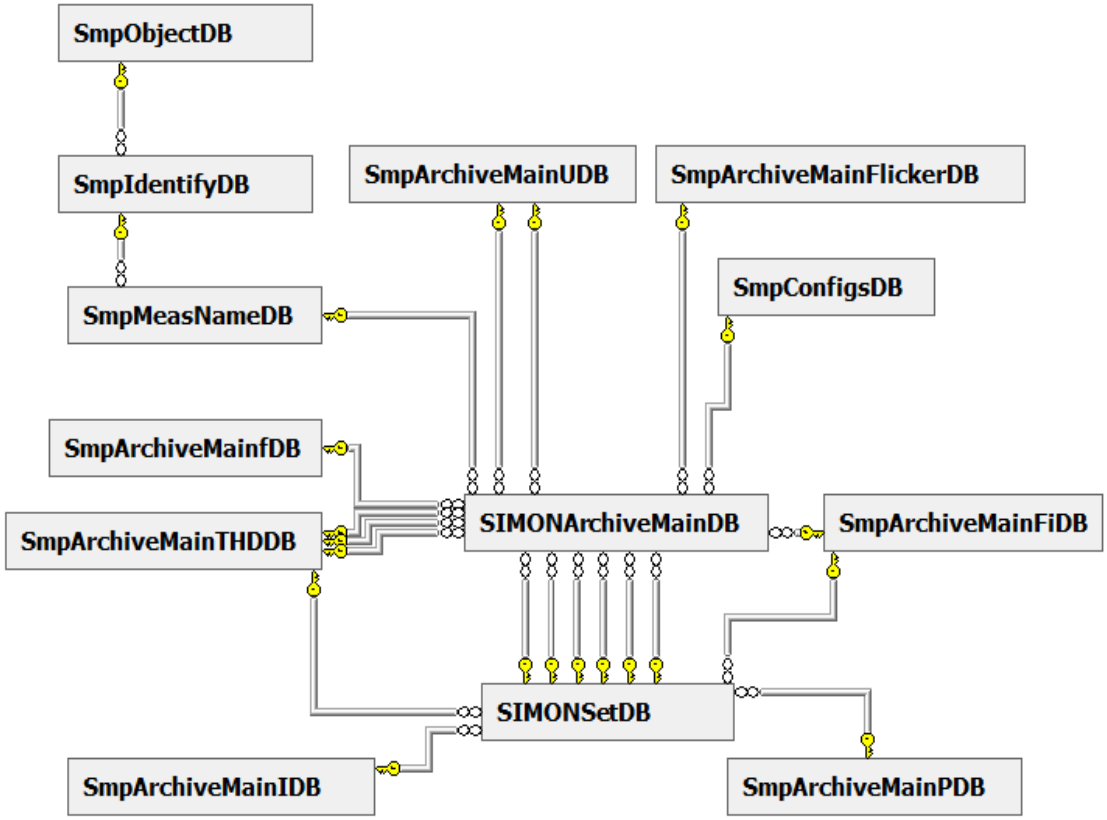
2.1. Databázová část aplikace

Jednou ze stěžejních oblastí byla práce s daty, moduly tvořené jako reporty vyžadují pro svoji práci vstupní data, která upravují podle svého účelu na požadovaný výstup. Bylo zapotřebí porozumět koncepci databáze a způsobu jak v ní jsou uložena.

2.1.1. Struktura databáze

Tato sekce se zabývá samotnou strukturou databáze, ta je pro všechny druhy přístrojů koncipována stejně. Pro jednotlivé přístroje se pak některé uložené události liší v závislosti na typu přístroje a jeho měřících schopnostech. Kompletní seznam tabulek

databáze najdeme v sekci 2.1.2. V databázi je struktura uložení zobrazená na obrázku 2. V tomto konkrétním případě se jedná o databázovou strukturu přístroje SIMON.



Obr. 1. Struktura databáze pro přístroj SIMON

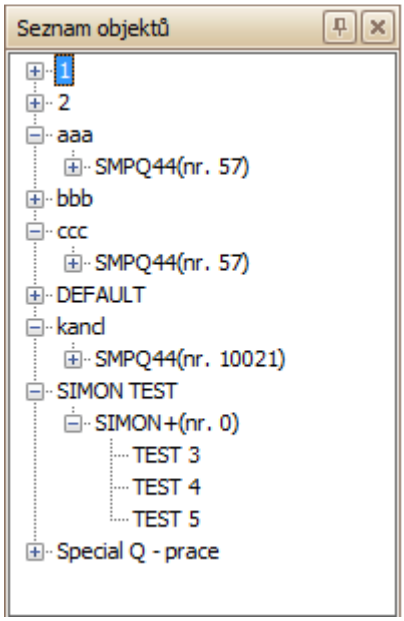
Zde je zobrazen hlavní vstupní bod, objekt tabulky *SmpObjectDB*, ten je relací 1:N spojen s objektem tabulky *SmpIdentifyDB* pomocí cizího klíče, jímž je primární klíč tabulky *SmpObjectDB*. *SmpIdentifyDB* označuje konkrétní instanci měřicího přístroje, *SmpMeasNameDB* je pak konkrétní uložené měření pro zvolený přístroj a je vázáno cizím klíčem k objektu tabulky *SmpIdentifyDB* opět relací 1:N. Tyto tři objekty tabulek jsou jediné položky databáze, které jsou viditelné ve stromě „Seznam objektu“ v ENVISu a které jsou pro všechny měřicí přístroje shodné. *SmpMeasNameDB* již umožňuje získat přístup k hlavnímu archívu daného typu přístroje.

SIMONArchiveMain je hlavním archivem přístrojů SIMON. Jak je vidět je spojen se všemi ostatními tabulkami příslušejícími k přístroji SIMON. Získávají se tak data o nastavení přístroje, měřených hodnotách, výpadcích a všech ostatních událostech, které přístroj zaznamenává.

Každý typ přístroje má svůj vlastní MainArchive. Údaje z přístroje jsou uloženy v několika tabulkách a každý typ záznamu o dané veličině má svůj vlastní Archiv. Tabulky jsou děleny do skupin podle uchovávaných údajů. Některé skupiny obsahují data o měřených hodnotách, jako jsou napětí *SmpArchiveMainUBD*, proud *SmpArchiveMainIBD*, výkon *SmpArchiveMainPBD* atp. Další skupiny tabulek udržují informace o nastavení přístroje v době měření (*SmpConfigsBD*).

V databázi jsou také uloženy různé procedury (storage procedures) pro usnadnění práce s databází. Pro vytváření záloh, mazání jednotlivých tabulek, odebírání celých objektů a podobně. Pro updatování informací nebo mazání je to bezpečnější způsob zásahu do databáze. Tímto se získá větší jistota, že budou dodržena integritní omezení. Aby bylo možno databázi používat jako zdroj dat, musí být zajištěno, aby se do ní nedostala data, která tam nepatří a zároveň se neztratila data, která v ní mají zůstat. K tomu slouží integritní omezení, zajišťující konzistentní stav databáze při úkonech jako je mazání nebo třeba úprava dat. Je tak zajištěno, že data v databázi jsou platná a lze s nimi pracovat.

Obr. 2. zobrazuje strukturu, tak jak jsou objekty databáze přístupné v programu ENVIS. Tato komponenta je součástí hlavního okna a je používána pro přístup k měřicím přístrojům uloženým v databázi a jejich měřeným výsledkům.



Obr. 2. Seznam objektu databáze z pohledu ENVISU

Základem celého stromu jsou objekty typu *SmpObjectDB*. To jsou uzly v kořenu stromu (1,2,aaa, atd.) Objekt může představovat jednu budovu nebo komplex budov jedné firmy. To znamená, že pod každým objektem může být uloženo více měřících přístrojů. Ty mohou měřit každý svůj objekt nebo více měřicích přístrojů měří jeden objekt na různé události. Po otevření kořenových objektů jsou vidět jednotlivé přístroje patřící k danému objektu. Jedná se o konkrétní fyzické měřicí přístroje. Každý z přístrojů má pak pod sebou několik různých měření podle toho, jak byly stahovány a ukládány do databáze.

2.1.2. Popis tabulek v databázi

CurveData, *CurveDataProfile*, *GrafUserProfile*, *GrafInfo* udržují informace o tom jaké křivky a jakým způsobem mají být vykresleny v grafu. *DatabaseUpdates* uchovává informace o updatech ENVISU. *InstrumentProfile* obsahuje informace s profily na načítání hodnot z přístrojů jiných společností. NovStateDB, NovConfigDB udržují konfigurační nastavení a data naměřená přístrojem NOVAR. SIMONArchiveMainDB, SIMONSetDB archivy přístroje SIMON, hlavní archiv udržující data a pomocný archiv, který udržuje informace o měřených proudech, ty se měří ve čtveřici. Těchto čtveřic může být zároveň měřeno až šest. SmlmnAllActDataDB, SmlmnConfigDB v těchto tabulkách jsou uložena konfigurační nastavení v době měření a naměřená data pro přístroje SML, SMM, SMN. SmyzArchiveDataDB, SmyzConfigDB, SmyzElmerActDataDB, SmyzStatusDB archivy měření napětí, proudů a elektroměrových údajů a také konfigurační nastavení pro přístroje SMY a SMZ.

Následující seznam tabulek patří k přístrojům SMP a SMQP. SmpArcConfigDB, SmpArchiveElmerDB, SmpArchiveLogDB, SmpArchiveMainDB, SmpArchiveMainfDB, SmpArchiveMainFiDB, SmpArchiveMainFlickerDB, SmpArchiveMainIDB, SmpArchiveMainPDB, SmpArchiveMainTHDDB, SmpArchiveMainUDB, SmpArchivePmaxDB, SmpArchivePQEventDB, SmpArchivePqEventTrendArchiveDB, SmpArchivePQMainDB, SmpArchivePQOscilogramDB, SmpArchiveSMProfileRecDB, SmpConfigDB, SmpConfigsDB, SmpDeviceUrl, SmpElectricityMeterConfigDB, SmpIdentifyDB, SmpInputConfigDB, SmpInstallConfigDB, SmpMeasNameDB, SmpObjectDB,

SmpOutputConfigDB, SmpOutputConfigUdalostDB, SmpOutputConfigVystupDB, SmpPQSettingsDB. Tyto typy přístrojů měří nejvíce parametrů, každý z parametrů má svůj main archive, ty se odlišují jmennou konvencí. Archivy jsou pojmenovány „SmpArchiveMain“ následováno typem měřeného údaje a koncovkou „DB“. Měří se napětí, proud, výkon, fáze, účiník, odběr elektrické energie a další parametry. Další archivy pak udržují údaje o nastavení přístroje, názvech měření atd.

2.2. Modulární systém

Modulární systém je systém obsahující nějaký základní objekt, který je rozšiřitelný pomocí dalších objektů. Tyto přídavné objekty se nazývají moduly neboli pluginy. Pomocí pluginů může být u aplikace upraven její vzhled nebo funkce. Protože lze modulární systém využít i pro rozšíření základních funkcí programu, vedla nakonec cesta k tvorbě univerzálního modulárního systému. Bylo tak potřeba několikrát přetvořit základní kámen systému, v původně navržené verzi nebyl modulární systém této funkčnosti schopen.

Na internetu je k nalezení několik odkazů, které se zabývají postupy jak vytvořit aplikaci s pluginy, pro inspiraci byl použit zdroj [1]. Po prostudování této tématiky bylo zjištěno, že tvorba modulárního systému musí probíhat vytvořením několika objektů. Je také nutno zabezpečit určitou stabilitu tvořeného systému pro správný běh aplikace bez kolizí s pluginy. Pokud by tato část byla zanedbána, mohlo by se stát, že by práce s pluginy mohla ohrozit stabilitu celé aplikace a vést k jejímu násilnému ukončení. To by se mohlo stát nejen při operacích s funkcemi pluginů, ale i při spravování a spouštění pluginů.

Pokud má být v aplikaci implementována modulární struktura, musí mít aplikace tři základní bloky. Prvním blokem je základní jádro pluginů. Může se jednat o interface nebo třídu. Pokud je použita třída, bývá to ve většině případů třída abstraktní. Použití třídy má oproti rozhraní jisté výhody, lze naimplementovat část kódu, o kterém víme, že bude pro všechny pluginy stejný, přímo do základní třídy.

Druhým blokem je část starající se o spravování pluginů. Úkolem takového správce je zjišťovat jaké pluginy jsou dostupné, jaké pluginy používáme, načítání pluginů ze souborů a jejich uvolňování z paměti, pokud není jejich služeb dále potřeba.

Musí mít nějaké jádro neustále kontrolující změny aplikace a pluginů, podle těchto událostí měnit chování pluginů, dodávat nová data pluginům a výsledky z pluginů předávat zpět programu. Musí umožňovat uživateli ukázat jaké pluginy jsou dostupné, jaké jsou aktivní a právě využívané. Zároveň musí uživateli dát možnost ovlivňovat, které pluginy chce využít a které mají být nečinné. Posledním blokem v tomto uspořádání jsou samotné pluginy. Ty se vytvářejí zvlášť jako knihovny a musí implementovat základní rozhraní nebo dědit od základní třídy. Přidávají se k aplikaci do předem stanoveného umístění, aby je mohl nalézt správce pluginů.

Struktura vytvářené aplikace se od prvotního uvažovaného návrhu trochu pozměnila a měla by vypadat přibližně tak, jak byla nyní představena. Po teoretické stránce byly představeny všechny aspekty problému a přichází tak na řadu praktické řešení.

3. Práce s NET Frameworkem

Celá práce byla tvořena v C# v prostředí NET Framework. V následujících kapitolách budou popsány některé z prvků NET Frameworku, které byly pro práci podstatné.

3.1. Třídy a rozhraní základní stavební kámen pluginu

Výše byla zmínka o použití rozhraní nebo třídy jako základního bloku pluginu. Nyní bude popsán rozdíl mezi jednotlivými prvky a začneme rozhraním. Rozhraní je šablona popisující jaké prvky musí objekt, toto rozhraní implementující, obsahovat. Těmito prvky mohou být metody, property apod. Pokud objekt rozhraní implementuje, musí obsahovat všechny prvky dané rozhraním bez výjimky. Rozhraní bylo zavedeno jako náhrada vícenásobné dědičnosti, ta je dostupná v jazyce C a C++, ale pod NET Frameworkem bude hledána marně. Pomocí rozhraní může být sjednocena funkce několika odlišných tříd, i když nemají, kromě implementace daného rozhraní, nic společného. Nemusí být známo, jak třída vypadá ani jaké má funkce, pokud ale implementuje metody z rozhraní je možno pomocí nich s třídou pracovat.

Druhý způsob tvorby pluginu předpokládá, že je děděno z nějaké základní třídy. Pro tyto účely se výborně hodí použit jako základní třídu, třídu abstraktní a ve většině případů tomu tak i je. Abstraktní a normální třída mezi sebou mají několik rozdílů. Tím největším je nemožnost vytvořit instanci abstraktní třídy. Abstraktní třídu lze použít pouze jako základní třídu pro dědění. V abstraktní třídě může být vytvořena normální metoda, navíc i metody abstraktní, což jsou metody, které je nutno v dědící třídě překrýt a to bez výjimky. Překrytí metody znamená implementaci kódu těla této metody v dědící třídě, překrývat lze i normální metody nejen abstraktní. V normální třídě nejsou abstraktní metody povoleny, je to z důvodu, že by v objektu třídy mohla být zavolána metoda, jejíž tělo by nemělo žádnou implementaci a to by vedlo k vyvolání výjimky.

Abstraktní třída a rozhraní, tvoří schéma metod a propert, které musí dědící třída implementovat. Oproti rozhraní má abstraktní třída výhodu, její neabstraktní metody mohou obsahovat kód provádějící nějakou činnost. Mezi třídou a rozhraním je ještě jeden veliký rozdíl. Zatímco třída může dědit od jedné jediné rodičovské třídy,

rozhraní může implementovat neomezeně. V tom je hlavní síla rozhraní a jistý způsob nahrazení vícenásobné dědičnosti. Pluginy jsou tvořeny buď poděděním třídy základní nebo implementací rozhraní a následným doplněním metod takovým kódem, aby plugin plnil funkci pro kterou je konstruován.

Třída může obsahovat soukromé a veřejné metody a proměnné. Rozdíl mezi nimi je v přístupu k těmto prvkům. Zatímco soukromé se uvozují klíčovým slovem *private* a mohou být volány pouze v rámci třídy, ve které jsou definované. Veřejné prvky uvozuje klíčové slovo *public* a lze je volat z vnějšku objektu pomocí tečkové notace. Existují ještě další uvozující slova. Mezi často používaná patří slovo *internal*, uvozující prvek, který je viditelný pro všechny objekty v rámci stejného namespace a *protected*, kterým jsou označeny prvky viditelné pouze v dědící třídě. Byl zmíněn název namespace, nyní bude trochu vysvětlen význam. Česky se namespace nazývá sestavení, je to oblast, ve které jsou seskupeny třídy a struktury s podobnými vlastnosti nebo podobnou oblastí působnosti. Tyto třídy ani nemusí být na jednom místě na disku, stačí pouze uvést stejný název namespace.

Stejně tak může namespace obsahovat jiný namespace, pro přístupy k jednotlivým namespace pak slouží opět tečková notace. Součásti NET Frameworku je několik namespace pomocí kterých můžeme v systému provádět různé operace, hlavním namespace je *Systém*. Vytvořit lze i vlastní namespace, nejlépe jako samotnou knihovnu, třeba pro nějaké univerzální třídy, u kterých je předpokládáno použití ve více aplikacích.

3.2. Net Framework a práce s databází

Databáze použita v ENVISU je typu Microsoft SQL. Různé formy této databáze (Microsoft SQL, MySQL atd.) jsou k nalezení v téměř každé větší firmě, patří totiž mezi nejpoužívanější. Databáze pochází ze sortimentu firmy Microsoft, stejně jako visual studio a Net Framework. Díky tomu je přímo v Net Frameworku implementováno několik užitečných tříd pro jednodušší práci s databází ve visual studiu.

Tyto třídy jsou k nalezení v namespace *System.Data.SqlClient*, již z názvu je poznat, že tento namespace je celý věnovaný práci s databází typu SQL. Pokud jsou

data, jež mají být získána, uložena v jiném typu databáze, je potřeba použít jiný způsob, případně dotáhnout komponenty pracující s daným typem databáze. Pro databázi oracle je například potřeba stáhnout a nainstalovat i klasického klienta pro oracle databáze. Aby bylo možno získat data z nějaké SQL databáze, stačí pro tuto činnost ve visuál studiu pouhé tři třídy z namespace *SqlClient*. Jedná se o třídy *SqlConnection*, *SqlCommand* a *SqlDataReader*.

Výpis jednoduchého kódu pro získání dat je vidět níže v popisu kódu 1. Hlavní třída pro práci s databází se nazývá *SqlConnection*. Je vytvořena její instance pojmenovaná *connection*. Instance *connection* bude udržovat veškeré informace o spojení a je potřeba po celou dobu práce s databází. Je tak perfektním adeptem na hlavní promněnnou sekce *using*. Jak je z kódu patrno při vytváření instance konstruktor přejímá jeden parametr typu *string*. Tento řetězec se nazývá *connection string* a obsahuje udaje o připojení. Udaje jsou zapsány ve formě „nazev parametru“ = „hodnota“. V *connection stringu* jsou pouze tři parametry *Data Source*, *Initial Catalog* a *Integrated Security*. První parametr uchovává hodnotu o zdroji dat, což je nějaký databázový server, v tomto případě pojmenovaný *SQLSERVER*, který se nachází na nějakém počítači, v tomto případě je pojmenován *PCSERVER*.

```
using (SqlConnection connection = new SqlConnection(@"Data
Source=PCSERVER\SQLSERVER; Initial Catalog=databasename; Integrated
Security=true;"))
{
      try
      {
            connection.Open();
            using (SqlCommand command = new SqlCommand("SELECT * FROM
      DataTable", connection))
            {
                  SqlDataReader reader = command.ExecuteReader();
                  while (reader.Read())
                  {
                        object param1 = reader[0];
                        object param2 = reader[1];
                        object param3 = reader[2];
                  }
            }
      }
      catch(Exception ex)
      {
            Console.WriteLine("Error at reading database: " +
      ex.ToString());
      }
}
```

Kód 1. Získání dat z databáze SQL

Jelikož databázový server může udržovat více databází, musí se předat serveru informace o jakou databázi se zajímáme. K tomu slouží druhý parametr *Initial Catalog* udávající z jaké databáze se mají zpřístupnit data pro spracování. Připojení k serveru je třeba také autentifikovat, na výběr je ze dvou možností volby buď uživatelské jméno a heslo pro připojení k databázovému serveru. Nebo jako v tomto případě parametr *Integrated Security* nastavíme na hodnotu true. Tím je předána serveru informace, že má být použito přihlašovacích údajů uživatele, aktuálně přihlášeného, neboli využít autentifikaci systému Windows.

Následně je provedeno připojení k serveru, využívá se k tomu metody *Open*. Pokud se nepodaří pokus o spojení se serverem, je vyhozena exception (výjimka). Z tohoto důvodu je celá další část obalena do sekce *try-catch-finally*. Jestliže bylo

připojení k databázi úspěšné, použije se instance třídy *SqlCommand* k vykonání SQL příkazu. Využívá se zde opět sekce *using*, ve které je vytvořena instance třídy pojmenovaná *command*. Konstruktor třídy přijímá dva parametry „SELECT * FROM DataTable“ je standardní SQL příkaz ve formátu jaký může být zadán například do SQL Management studia. Druhý parametr *connection* je typu *SqlConnection* a definuje, na kterém připojení se má příkaz provést.

Samotné provedení příkazu je zajištěno zavoláním metody *ExecuteReader* instance *command*. Tato metoda vrací instanci třídy *SqlDataReader*, v tomto případě pojmenovanou *reader*, ze které budeme načítat veškerá vrácená data. V readeru je zavolána nejprve metoda *Read* ta vrací true, pokud je další řádek dat k dispozici. Proto je tato metoda uzavřena v cyklu while, tím se zaručí postupné získání všech načtených řádků. Po provedení této metody může být přistoupeno k datům v aktuálním řádku tak, že se využije indexace u readeru jako u pole, jak je znázorněno na dalších řádcích. Místo indexů může být použíto i jmen sloupců, zde ale hrozí riziko chyby pokud v databázi sloupec někdo přejmenuje. Protože dopředu není známo, jakého typu budou uložená data, je vracen typ *object*.

3.3. Reflexe

Protože systém načítání pluginů využívá takzvanou reflexi a ta je využita u jednoho z pluginů, je na místě si o ní něco málo říci. Reflexe dovoluje za běhu zjistit potřebné informace o instanci nějakého objektu tím, že přečte jeho metadata. Každá třída vytvořená pod NET Frameworkem obsahuje metadata, což jsou data obsahující popis jejího typu, metod, propert a proměnných. U každého objektu existuje metoda *GetType*. Tato metoda vrací objekt typu *Type*, ve kterém je údaj o třídě, ze které je instance vytvořena a namespace, ve kterém se třída nachází. Toto je to nejzákladnější druh reflexe, který je obsažen přímo v namespace *System*. Všechny ostatní třídy, které se pro reflexi používají, jsou k nalezení v namespace *System.Reflection*. Jednou z obsažených tříd je *Assembly* s její pomocí je možno přistupovat k datům právě běžící aplikace, nebo vytvořit instanci nějakého objektu ze souboru knihovny na disku.

Pokud je pomocí metody *GetTyp* zjištěna třída, ze které je instance vytvořena, může být pomocí metod třídy *Type* dále zjištěno, jaké obsahuje property, pro popis property slouží třída *PropertyInfo*. Dále jaké obsahuje metody, pro jejich popis slouží

třída *MethodInfo* nebo třeba constructory, pro ně je zase použito třídy *ConstructorInfo* ,takto by bylo možno pokračovat se všemi možnými členy. Protože je postup pro získávání členů pro všechny možné typy stejný, bude na ukázku popsán postup pro získání propert.

K získání property slouží metoda *GetProperty*, ta přijímá jeden parametr typu *string* se jménem property a vrací typ *PropertyInfo*. Pokud není jméno známo, nebo je požadován seznam všech propert, využívá se metody *GetProperties*. Ta vrací pole instancí *PropertyInfo* a může být buď bez parametru, v takovém případě vrací všechny veřejné property, nebo s jedním parametrem typu *BindingFlags*. Parametr je možné kombinovat pomocí binárního operátoru. Díky tomuto parametru lze určit jaké property mají být vráceny, veřejné, privátní, metody této třídy, nebo metody této třídy a zároveň metody třídy, ze které je poděděno.

S instanci třídy *PropertyInfo* lze zjistit její jméno, to je uloženo v parametru Name. Nebo hodnotu této property v nějaké instanci dané třídy za pomocí metody *GetValue*. Ta přijímá jako parametr instanci třídy, ze které se zjišťuje hodnota property, jako druhý parametr pak lze zadat index, pokud je property pole, jinak se nastavuje hodnota *null*. Podobným způsobem lze pracovat se všemi elementy třídy.

3.4. Načítání modulů v NET Framework

Při hledání informací o práci s pluginy v prostření NET Framework bylo nejvíce inspirace čerpáno z odkazu [1]. O načítání pluginu se stará plugin manager a je rozděleno v několika metodách. Pluginy jsou k aplikaci přidány jako dll knihovny. Pro jejich načtení je pak potřeba znát jméno typu a namespace ve kterém tento typ v souboru nalezneme.

Původně měly všechny pluginy název hlavního typu pojmenován jednoduše *plugin*, takže při kontrole typu se zjišťovalo pouze, jestli jde o tento typ. Během práce jsme došli k problému s velkým množstvím souborů knihoven, pokud by byl pro každý plugin použit jeden soubor. Přitom by bylo možno pluginy s podobným zaměřením udržovat v jednom souboru a namespace. Bylo tak potřeba vyřešit problém jak rozlišit v jednom namespace typy pluginů od typů pomocných tříd a od sebe navzájem. Toho nakonec bylo docíleno pojmenováním každého typu, ze kterého se tvoří plugin,

začínajícím řetězcem „Plugin_“ následovaným názvem pluginu. Pro načítání knihoven za běhu je potřeba do projektu přidat reference na namespace *System.Reflection* získá se tak přístup ke třídě *Assembly*. Postup načítání pluginu ze souboru dll je sepsán v kódu 2, informace o pluginu jsou uloženy v instanci třídy *availablePlugins*, o ní bude více napsáno v sekci o plugin manageru.

```
try
{
      if (!File.Exists(AvPlugin.PluginFile)) return -1;
      Assembly asm = Assembly.LoadFile(AvPlugin.PluginFile);
      foreach (Type typ in asm.GetTypes())
      {
            if (typ.Namespace == "ENVIS.Plugins")
            {
                  if (typ.Name == AvPlugin.PluginTypeName)
                  {
                        PluginBaseClass plugin =
                        CreateInstancePlugin(typ, AvPlugin.PluginFile);
                        if (plugin != null)
                        {
                              usedPlugins.Add(plugin);
                              InicializeNewPlugin(plugin);
                              availablePlugins.Remove(AwPlugin);
                        }
                  }
            }
      }
      usedPlugins.Sort(SortByNamePluginBaseClass);
      return 0;
}
catch (Exception ex)
{
      MessageBox.Show("Error in loading plugin file " +
      AvPlugin.PluginFile, "Plugin manager", MessageBoxButtons.OK,
      MessageBoxIcon.Error);
      return -1;
}
```

Kód 2. Načítání modulů do aplikace při běhu programu

Pří práci se soubory se může snadno vyskytnout nějaká chyba, proto je celý kód uložen v sekci *try-catch*. Pokud se nějaká chyba vyskytne, je uživatel informován, že se vyskytla chyba při načítání pluginu s informací o který plugin se jedná.

Plugin je načítán ze souboru, proto je nejprve zjištěno, jestli soubor s daným jménem v úložišti pluginů existuje. Následně je vytvořena instanci třídy *assembly* pomocí statické metody této třídy *LoadFile*, metoda přijímá jeden parametr a to cestu k souboru, který chceme načíst. Vytvořená instance obsahuje všechny třídy, struktury a další prvky, které byli v souboru obsaženy. Seznam těchto prvků získáme pomocí metody *GetTypes*, ta vrací pole typů. Postupně jsou všechny typy prohledány a zjistí se, jestli se jedná o typ, který tvoří pluginy.

```
try
{
ConstructorInfo[] ConstructorInformation =
     Typ.GetConstructors(BindingFlags.Instance | BindingFlags.Public);
  PluginBaseClass plugin = null;
  if (ConstructorInformation.Length > 0)
  {
    plugin = (PluginBaseClass)Typ.Assembly.CreateInstance(Typ.FullName,
     true, BindingFlags.Instance | BindingFlags.Public, null, new
     object[] { FileName }, CultureInfo.CurrentCulture, null);
  }
  return plugin;
}
catch (Exception ex)
{
  MessageBox.Show("Error in creating instance plugin in plugin file " +
     FileName, "Plugin manager", MessageBoxButtons.OK,
     MessageBoxIcon.Error);
  return null;
}
```

Kód 3. Kód pro vytvoření instance nějaké třídy za běhu programu

Aby se jednalo o typ, který je pluginem musí být v namespace *ENVIS.Plugins*, ověření zajistí první *if* podmínka. Najde se jméno typu daného pluginu, pokud je v dané assembly nalezen, je vytvořena instance tohoto typu pomocí kódu v ukázce kód 3. Při

hledání dostupných pluginů není ve druhém *if* příkazu vyhledáváno jméno typu pluginu, ale všechny typy které začínají na „Plugin_“, tak jsou získány všechny typy pro tvorbu pluginu, i pokud je jich v jednom souboru více.

Tvoření instance je opět obaleno v bloku *try-catch*. Pomocí instance *ConstructorInformation* se zjistí, jestli je v daném typu constructor, který bude potřeba. Jak vidíme z vložených parametrů, hledáme instanční veřejný constructor. Pokud je počet nalezených constructor roven nule, není možné instanci vytvořit, protože nebyl nalezen vhodný constructor. Metoda *Typ.Assembly,CreateInstance* přebírá ve všech svých přetíženích jako parametr úplný název typu instance. Pokud by byla použita přetížení metody, které přijímá pouze tento jeden parametr, vytvořila by se instance daného typu za předpokladu, že typ obsahuje bezparametrový constructor, jinak by došlo k výjimce.

V práci je použit constructor, který jako parametr přebírá cestu k souboru, proto musí být zvolena složitější varianta tvorby instance. Prvním parametrem je zmiňovaný úplný název typu instance, druhým parametrem, jestli se mají ignorovat malá a velká písmena. Další parametr je kombinace parametrů hledaného typu constructoru. Následující parametr není podstatný proto je *null*, po něm je vloženo pole parametrů předávaných constructoru. Pole je typu object, ale parametry samotné, musí mít možnost přetypování na daný typ, který je v constructoru očekávám. Následujícím parametrem můžou být ovlivněny různé parametry podle zvyků dané nastavené kultury a poslední parametr je opět nepodstatný. Tím je instance vytvořena a vrácena jako výstupní parametr.

Zkontroluje se, jestli vytvoření instance proběhlo úspěšně. Pokud ano, je plugin přidán mezi používané moduly, zinicializuje se a odebere z dostupných pluginů. Seznam používaných pluginů je následně pro lepší orientaci abecedně seřazen. O používaných a dostupných pluginech bude více napsáno v sekci o plugin manageru.

4. Struktura vytvářené aplikace

Při vytváření bylo doplněno ke třem základním prvkům ještě report viewer a setup form. Každému z těchto prvků bude věnována samostatná kapitola. Také se podíváme na práci s databází a způsob získávání dat z ní.

4.1. Práce s databází

4.1.1. Databázové pojmy

Ještě než začne samotný popis databáze, bude vysvětleno několik použitých pojmů. Prvním pojmem je primární klíč, tak se označuje hodnota v řádku (sloupeček v tabulce), podle kterého jsou řádky jednoznačně identifikovatelné. Hodnota primárního klíče musí být pro každý řádek unikátní v celé tabulce. Nejčastěji se jako primární klíč používají hodnoty z oboru celých nezáporných čísel (int). Dalším pojmem je cizí klíč využívaný při relacích (spojení) tabulek. Jedna z tabulek obsahuje sloupeček, ve kterém jsou uloženy klíčové hodnoty z tabulky druhé (nejčastěji to bývá právě primární klíč). Snadno tak lze spojit řádek v jedné tabulce s řádkem v tabulce druhé, čímž získáme komplexnější informaci.

Posledním termínem jsou relace. Relace je vazba mezi dvěma tabulkami a existuje ve třech možných vazbách 1:1, 1:N a M:N. Relace umožňují všechny dnes běžné databázové systémy. Samozřejmě lze uložit i tabulky bez relací s jinými tabulkami, to může být využito třeba pro ukládání informací o databázi např., kdy došlo k poslednímu updatu dat.

První vazba 1:1 se moc často nepoužívá, jelikož lze všechna data uložit do jediného řádku tabulky. Využít ji lze při použití dvou tabulek. Druhá tabulka tak slouží pouze jako rozšíření první tabulky o informace, které nebudeme potřebovat tak často jako data v tabulce základní. Například pokud existují ve firmě zaměstnanci i z ciziny, budeme u nich potřebovat uchovávat údaje, které by u českých zaměstnanců neměli význam (země původu, číslo pasu atd.). Nejčastěji využívaná je vazba 1:N kdy řádek v jedné tabulce doplňuje několik řádků v tabulce druhé. Například jedna tabulka obsahuje děti a druhá rodiče. Na jeden primární klíč v tabulce rodičů, jednoho rodiče, může odkazovat několik cizích klíčů v tabulce druhé. Jinak řečeno jeden rodič může mít jedno dítě, ale

stejně tak nemusí mít dítě žádné nebo naopak více. Posledním druh vazby M:N označuje stav, kdy se na jeden řádek v tabulce první váže několik řádků v tabulce druhé a zároveň se tyto řádky váží ještě na jiný řádek z první tabulky. Pro tuto vazbu je většinou přidána ještě třetí tzv. vazební tabulka. Ve vazební tabulce jsou pak uchovány primární klíče z obou tabulek pro dané řádky, které spolu jsou v relaci. Na příkladě rodičů a dětí si to lze představit tak, že rodič může mít několik dětí, ale zároveň jedno dítě má ve většině případů dva rodiče.

4.1.2. Ukázka příkazu select pro získání dat z databáze

Získání dat z databáze je popsáno na následujícím příkladu. Jedná se o změřené údaje, které by ve stromu objektů byly k nalezení pod objektem DEFAULT, měřicí přístroj SMV44(nr. 10027) a názvem měření 2.

```
(SELECT avg_uLN,avg_iL FROM  SmpArchiveMainDB WHERE keymeasName =
(SELECT Id FROM  SmpMeasNameDB WHERE measName = '2' AND identifyDB
=
(SELECT Id FROM  SmpIdentifyDB WHERE DeviceNo = 10027 AND objekt =
(SELECT Id FROM  SmpObjectDB WHERE objekt = 'DEFAULT'))))
```

Kód 4. Získání dat z databáze pomoci vnořených selectů.

```
DECLARE @ObjectDB int
DECLARE @IdentifyDB int
DECLARE @MeasNameDB int
SET @ObjectDB = (SELECT Id FROM  SmpObjectDB WHERE objekt =
'DEFAULT')
SET @IdentifyDB = (SELECT Id FROM  SmpIdentifyDB WHERE DeviceNo =
10027 AND objekt = @ObjectDB)
SET @MeasNameDB = (SELECT Id FROM  SmpMeasNameDB WHERE measName =
'2' AND identifyDB = @IdentifyDB)
SELECT avg_uLN,avg_iL FROM  SmpArchiveMainDB WHERE keymeasName =
@MeasNameDB
```

Kód 5. Získání dat z databáze pomocí proměnných a oddělených selectů

Kód 4 a kód 5 zobrazují dva způsoby, jak lze získat z databáze hodnoty průměrných napětí a proudů na fázi. První způsob je napsání jednoho dlouhého příkazu. Ve druhém způsobu je napsáno několik kratších příkazů za sebou. Druhý způsob je přehlednější a bude použit k vysvětlení závislostí nutných pro získání konkrétních dat. První tři řádky deklarují proměnné, všechny jsou typu *int* protože je to typ hodnot primárních klíčů v tabulkách *SmpObjectDB, SmpIdentifyDB a SmpMeasNameDB.*

Kódem na druhém řádku je vybrán *Id (SELECT Id)* z tabulky *SmpObjectDB (FROM SmpObjectDB)* takové, že splňuje constraint (podmínka nebo omezení, která musí být dodržena, je tak omezeno množství vybraných řádků) nastavenou tak, aby hodnota prvku *objekt* v daném řádku byla rovna *DEFAULT (WHERE objekt = 'DEFAULT').* Tímto způsobem je vybrán z objektů v tabulce řádek, který obsahuje v položce objekt hodnotu DEFAULT. *Id* řádku, jenž je primárním klíčem určujícím daný řádek, se uloží do proměnné *@ObjectDB.* Podmínky můžou být nastaveny pro sloupce, řádky nebo pro celou tabulku.

Získání primárního klíče pro daný objekt umožňuje získat z databáze měřicí přístroje náležející k danému objektu. Následně pak k získání dat přístrojem naměřených. K tomu slouží druhý řádek, pomocí něhož je vybráno *Id* měřicího přístroje z tabulky *SmpIdentifyDB* tak, aby cizí klíč uložený v datech přístroje byl shodný s primárním klíčem vybraného objektu (*objekt = @ObjectDB).* Tím se zajistí, že vybraný přístroj náleží k našemu objektu. Pokud by ale zůstala constraint pouze v tomto tvaru, vrátilo by se více hodnot *Id* a to jedno pro každý z měřicích přístrojů náležících k tomuto hlavnímu objektu.

Aby byl získán jeden konkrétní měřicí přístroj, musí se constraint rozšířit pro tyto konkrétní potřeby takovýmto způsobem *WHERE DeviceNo = 10027 AND objekt = @ObjectDB.* Výsledkem jsou přístroje, jejichž DeviceNo je rovno 10027 a zároveň jejich cizí klíč *objekt* nabývá hodnoty primárního klíče zadaného objektu. Protože takový přístroj bude v databázi pouze jeden, je získáno jediné Id patřící hledanému přístroji. Když je známo Id měřicího přístroje, zbývá zjistit Id zadaného měření. To se získá za pomocí třetího řádku, opět se vybírá Id tentokrát z tabulky *SmpMeasNameDB,* které splňuje opět dvě constraint. První část constraint je cizí klíč shodný s primárním klíčem zvoleného měřicího přístroje. Druhá část říká, že to má být měření s názvem *2*.

Nyní je získán správný údaj, kvůli kterému byly podniknuty všechny předchozí kroky Id měření, z něhož mají být získány údaje. V posledním řádku je ukázka toho, jak lze získat průměrné napětí a průměrný proud na fázi. Vybírá se z tabulky *SmpArchiveMainDB*, která má cizí klíč shodný s primárním klíčem získaného měření. Tím se získají všechny hodnoty zadaných parametrů *avg_uLN,avg_iL*. Někdy se může stát, že nejsou potreba všechny hodnoty, ale jen část, například časově omezený úsek dat. Data z požadovaného úseku mohou být vybrána ručně a zbytek se nechá být. Je to však zbytečné plýtvání prostředky. Všechna data, která byla nalezena, máme uložena v paměti i když požadujeme jen část. Druhou, lepší variantou, je rozšířit constraint tak, aby byla vybrána jen data z hledaného časového úseku. Využívá se k tomu stejná podmínka, použitá na získání dat pouze rozšířená o hledaný časový úsek. Výsledek je vidět v kódu 6.

```
SELECT * FROM SmpArchiveMainDB WHERE keymeasName = @MeasNameDB AND
keyTime > CONVERT(datetime, '1.1.2010 9:43:0', 104) AND
keyTime < CONVERT(datetime, '5.1.2010 2:0:0', 104)
```

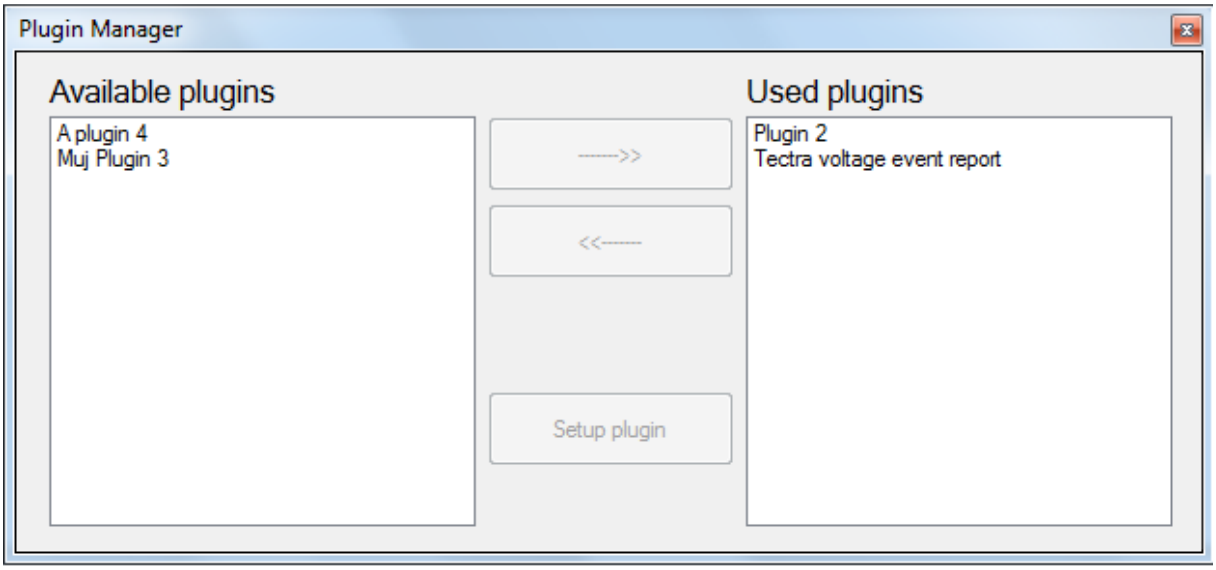
Kód 6. Data omezené časovým úsekem

Funkce *CONVERT* slouží k převodu vložené hodnoty v druhém parametru např. *'1.1.2010 9:43:0'* do správného formátu typu, který je uveden jako první parametr. V tomto případě je typem *datetime*, který se používá pro hodnoty týkající se času a data.

4.2. Plugin manager

Plugin manager je zmiňovaným správcem pluginů. Hlavní část se skrývá v třídě *PluginManagerClass,* kde jsou uloženy všechny metody pro práci s aplikací a pluginy. Pro interakci uživatele s plugin managerem je k dispozici třída *PluginManagerMainForm,* je to jediná viditelná komponenta v plugin manageru a zobrazena je na obrázku 4. Jelikož je funkce této komponenty jednoduchá, je popsána jako první, dále následuje samotného jádro.

4.2.1. Plugin manager form



Obr. 3. Plugin manager main form

Na obrázku 3. je zobrazen jednoduchý okenní formulář, na kterém jsou dva listboxy. V levém listboxu se zobrazují pluginy, které jsou dostupné, ale nejsou používány. V pravém listboxu se pak zobrazují pluginy, momentálně využívané. Mezi listboxy se nacházejí tři tlačítka. První tlačítko slouží pro přesouvání neaktivních pluginů do aktivních, to znamená k aktivaci pluginu a jeho načtení do paměti. Druhé pak aktivní pluginy deaktivuje, tím je uvolníme z paměti a šetří se tak systémové prostředky vypínáním nevyužívaných pluginů.

Poslední tlačítko je používané jen pro některé pluginy. Aby byl k tomuto tlačítku získán přístup, je potřeba aby plugin, který je aktuálně vybraný, byl aktivní a měl nějaká vlastní nastavení, která je možno upravit. Plugin,který bude mít možnost individuálního nastavení, se musí sám postarat o interakci s uživatelem. Toho bude ve většině případů dosaženo použitím nějakého okenního formuláře, který zpřístupní prvky a hodnoty, jejichž parametry bude možno ovlivňovat. Pro tyto případy není možno vytvořit formulář univerzální, jelikož nastavované parametry budou vždy závislé na konkrétním pluginu a jeho funkcích.

4.2.2. Plugin manager hlavní třída

Jedná se o standardní třídu v namespace s názvem Envis.PluginManager. Hlavní třída obsahuje metody a proměnné vyobrazené v kódu 7. Nalézá se zde constructor a destructor třídy plugin manager, constructor je metoda, která se zavolá jen při vytvoření nové instance. Přebírá jeden parametr typu *BarSubItem*. Původně byly pluginy zamýšleny jen jako výstupní reporty přístupné z menu, každý plugin tak musí zaregistrovat svoji položku do menu. Na tuto registraci potřebuje vědět do jakého menu se má plugin zaregistrovat a právě toto menu je předáno jako vstupní parametr. V constructoru se také nastaví všechny proměnné, ty jsou zobrazeny dále. První z nich je *pluginsPath,* jedná se o textový typ *string*, v kterém je uložena cesta, kde má plugin manager hledat pluginy. *ManegerConfigFile,* je také typu *string* a je v něm uložena cesta k souboru s konfiguračními nastaveními pro plugin manager. V současné době zatím soubor obsahuje pouze seznam pluginů, spouštějících se po spuštění aplikace. Opakem constructoru je destructor třídy, ten se zavolá při uvolnění třídy z paměti, používá se k uvolnění dodatečných prostředků. V tomto případě je v něm volána metodu *Dispose,* ve které se volá metoda *Dispose* pro každý z používaných pluginů, aby i ony uvolnily používané prostředky. Následně jsou uvolněny všechny seznamy a objekty.

LoadedObject je typu list, což je seznam objektů určitého daného typu, takzvaný generický list. V našem případě je to seznam položek typu *object*. Tento seznam využíváme pro ukládání objektů, které potřebují pluginy ke svoji práci. Seznam je naplňován v hlavní aplikaci. Typ *object* byl vybrán, protože je možné jakýkoliv objekt, používaný v NET Frameworku, přetypovat na typ *object* a zpět. Seznam tak je možno naplnit čísly, textovými řetězci nebo i instancemi používaných objektů. Další tři proměnné jsou také typu list, ale mají jiné typy ukládaných položek. Proměnná *availablePlugins* využívá typ *AvailablePlugin*, tento typ byl speciálně vytvořen právě pro tuto proměnnou a bude popsán v jedné z dalších kapitol.

```
Constructor a destruktor
public PluginManagerClass(BarSubItem PluginsMenuItem)
~PluginManagerClass()

Proměnné
string pluginsPath;
string managerConfigFile;
List<object> loadedObject;
List<AvailablePlugin> availablePlugins;
List<PluginBaseClass> usedPlugins;
List<PluginBaseClass> alwaysUsePlugins;

Ostatní funkce
private List<string> GetSaveUsePlugin()
private void SaveUsedPlugins()
private void SetPluginManagerItem()
private void pluginManagerMenuClick(object sender, ItemClickEventArgs e)
public void Dispose()
```

Kód 7. Metody a proměnné hlavní třídy plugin manageru

Zbývající dva seznamy jsou typu *PluginBaseClass*, to je základní třída všech pluginů, bude o ní dále celá jedna kapitola a tak zde nebude probírána. Původně byl i seznam *availablePlugins* typu *PluginBaseClass*, díky tomu bylo snadnější spravovat pluginy a jejich aktivaci a deaktivaci. Zjednodušeně lze říci, že stačilo jen přesunout plugin z jednoho seznamu do druhého. Z programátorského hlediska jednodušší, ale neefektivní co se systémových prostředků týká.

Je pravděpodobné, že bude vytvořeno velké množství pluginů, pro jednoduchost si lze představit například počet 1000 pluginů, z nichž uživatel v současné době využívá jen 10 pluginů. Velikost paměti využívaná každým z pluginů pro svoji práci může být například 100 kB. Pokud by zůstalo u starého způsobu spravování, udržovaly by se v paměti všechny pluginy, to znamená 1000 pluginů krát 100 kB, což je 100 MB obsazené paměti. Při novém způsobu zpravování se udržuje v paměti jen používaných 10 pluginů, čili 10 * 100 kB, obsazujeme tak jen 1 MB paměti. K tomu drobné množství informací o neaktivních pluginech, ale velikost objemu těchto dat lze zanedbat. Může tak ve výsledku dojít k velikému ušetření systémových prostředků.

UsedPlugins obsahuje pluginy, které jsou v současné době používané, neboli ty, které se v plugin manager form objevují v pravém okně. *AlwaysUsePlugin* je seznam, jehož existence v původním plánu nebyla zamýšlena a byl doplněný až v průběhu práce. Obsahuje pluginy, které budou použity vždy, to znamená, že se inicializují se startem aplikace a končí až s koncem aplikace. Tyto pluginy se nezobrazují v plugin manageru, uživatel tak nemá možnost s nimi manipulovat. Mohlo by se totiž stát, že plugin bude životně důležitý pro aplikaci a pokud by ho uživatel nevědomky vypnul, mohla by aplikace spadnout.

V constructoru jsou dále volány tři metody, *SetPluginManagerItem*, *GetAllAvailablePlugins* a *SetUseAllUsedPlugins*. Jsou to některé z metod pro práci s pluginy, jejichž seznam je níže v kódu 8. První zmiňovaná metoda zaregistruje do menu položku pro plugin manager. Při každém kliknutí na tuto položku se spustí metoda *pluginManagerMenuClick*, ta má za úkol zobrazení plugin manager form a v zápětí po jeho zavření se uloží seznam používaných pluginů, pro případ nějakých provedených změn. Druhá z metod, *GetAllAvailablePlugins* vrátí seznam všech dostupných pluginů. Poslední zmiňovaná metoda *SetUseAllUsedPlugins* jako první věc zavolá metodu *GetSaveUsePlugin*. Tato metoda z konfiguračního souboru zjistí všechny pluginy, které byli aktivované při posledním vypnutí ENVISU. Následně postupně zaktivuje všechny pluginy jeden po druhém metodou *ReplaceAwailablePluginsToUsePlugins*.

Metoda *ReplaceAwailablePluginsToUsePlugins* přijímá jako parametr *AvPlugin* typu *AvailablePlugin*, který představuje objekt ze seznamu dostupných pluginů, jenž se má převést do pluginů používaných. Nejprve se zkontroluje, jestli je soubor, v kterém se plugin nachází, dostupný. Poté je vytvořena instance daného pluginu a následně odebrána položka z dostupných pluginů a přidána instanci pluginu do používaných pluginů. K metodě *ReplaceAwailablePluginsToUsePlugins* je ekvivalentní metoda *ReplaceUsePluginsToAwailablePlugins*, ta naopak převádí plugin z používaných do dostupných pluginů. Jako vstupní parametr přejímá *PluginBaseClass* typu *UsePlugin*.

```
Metody třídy pro práci s pluginy
private void SetUseAllUsedPlugins()
private bool IsPluginUse(AvailablePlugin Plugin)
internal int ReplaceUsePluginsToAwailablePlugins(PluginBaseClass
UsePlugin)
internal int ReplaceAwailablePluginsToUsePlugins(AvailablePlugin
AvPlugin)
private void GetAllAvailablePlugins()
private PluginBaseClass CreateInstancePlugin(Type Typ, string FileName)
private void AddPluginToAvailable(Type Typ, string FileName)
private void InicializeNewPlugin(PluginBaseClass Plugin)
private void LoadNewObjectToPlugins(object PluginObject)
public void AddNewObject(object ObjectForPlugins)
```

Kód 8. Metody plugin manageru pro práci s pluginy

4.2.3. Třída AvailablePlugin

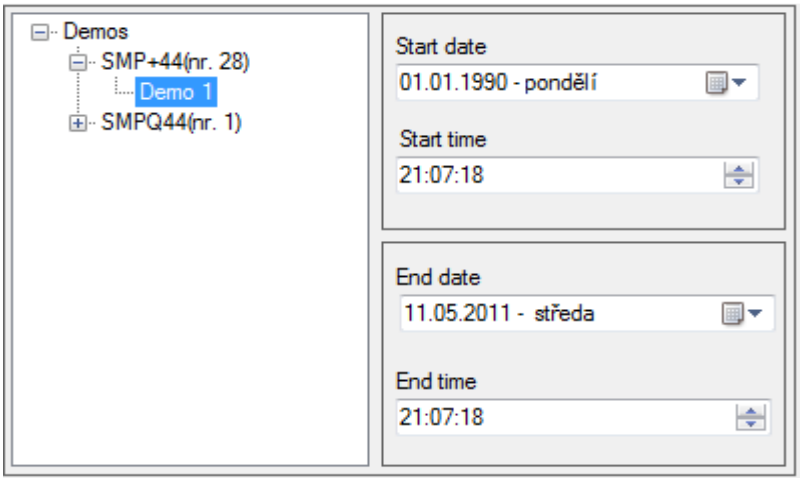
Je zobrazena v kódu 9, jsou zde tři proměnné typu *string*, těmi jsou *PluginName* v ní je uloženo jméno pluginu, *PluginFile* to je cesta k souboru, kde je plugin k nalezení. *PluginTypeName* obsahuje název typu třídy, jejíž instanci má tento plugin při běhu představovat. Třetí zmiňovaná položka původně v třídě nebyla, během práce byl ale předělán základ takovým způsobem, že je možno mít více pluginů v jedné *dll* knihovně. Bylo tak potřeba odlišit jaký plugin ze souboru konkrétní instance zastupuje.

```
public class AvailablePlugin
{
  public string PluginName { get; set; }
  public string PluginTypeName { get; set; }
  public string PluginFile { get; set; }
}
```

Kód 9. Seznam položek třídy available plugin

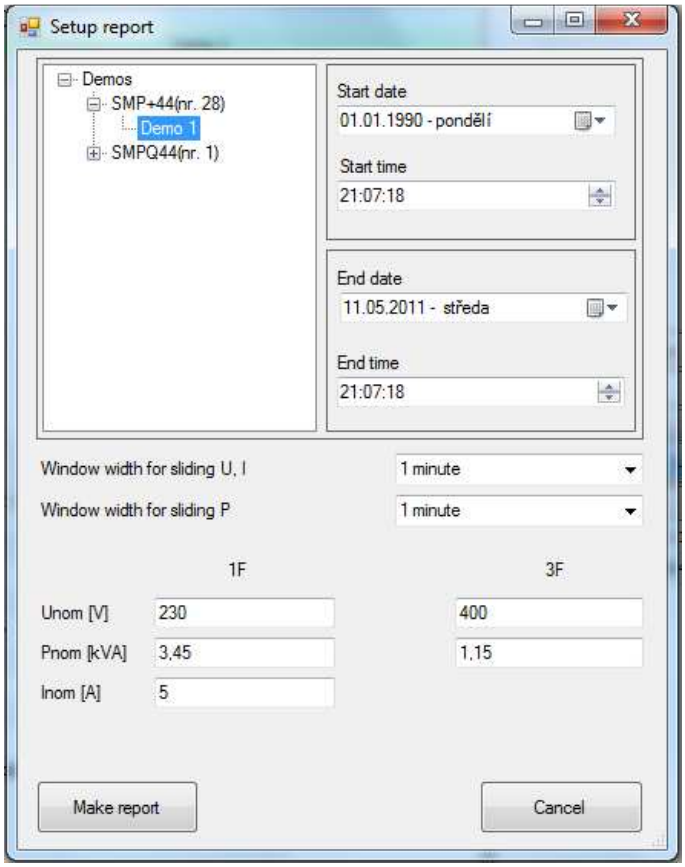
4.3. Setup form

Pro využití funkce pluginu je třeba mu předat vstupní informace. Těmi jsou objekt, přístroj a měření, s kterými má plugin pracovat. Také je potřeba omezit oblast zpracovávaných dat, která mají být získána na rozsah od kdy do kdy. Tyto informace bude nutno předat každému pluginu, byla tak pro tyto účely vytvořena komponenta typu *UserComponent*, která obsahuje všechny potřebné prvky pro získání požadovaných údajů. Samotná komponenta je zobrazena na obrázku.



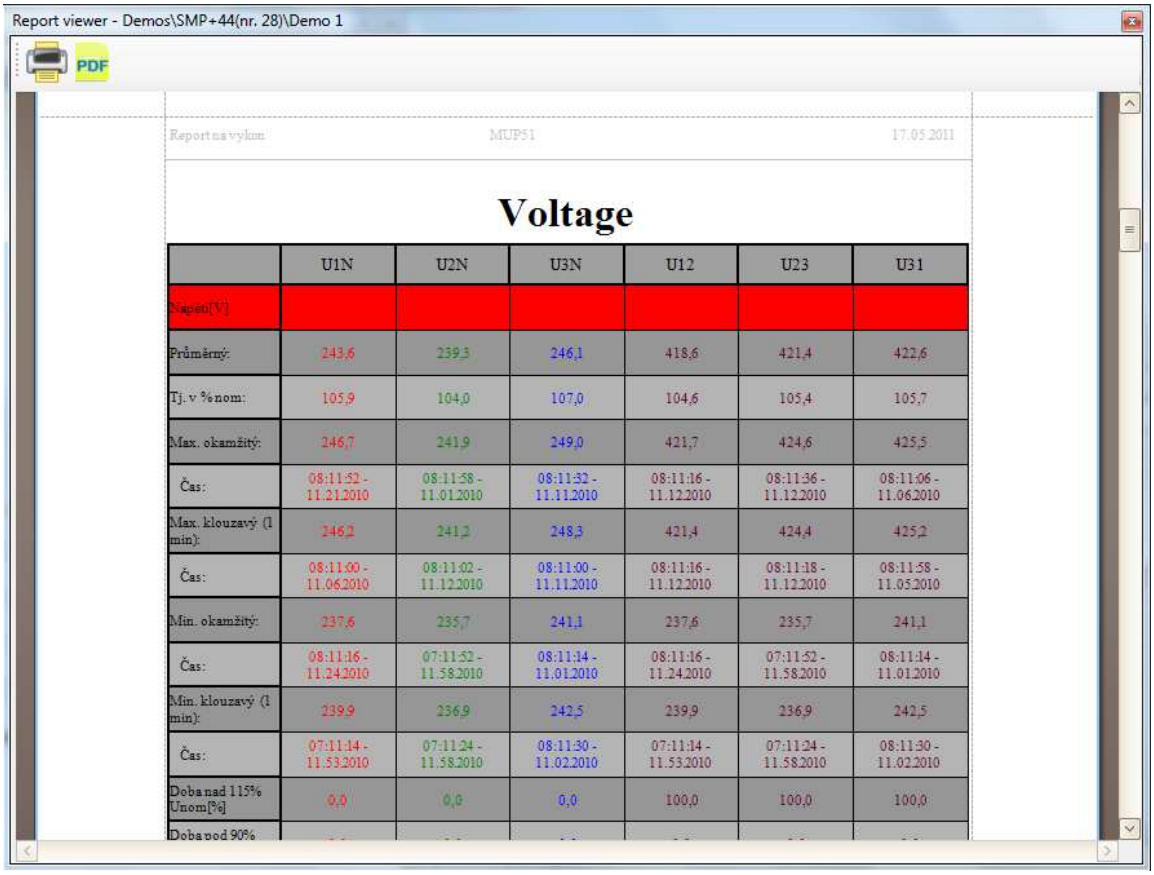
Obr. 4. Komponenta pro výběr měření a rozsahu data

Vidíme na ní treeview prvek shodný se základním prvkem ENVISu pro získání měření jednotlivých přístrojů a objektů. Následně pak 4 prvky typu *DateTimePicker*, použity pro získání počátečního a koncového data a času. Setup form jako celek je pak součástí knihovny pluginu. Jednotlivé pluginy mají další různé parametry na základě, kterých provádějí operace, pro které byly navrženy. Není tak možné vytvořit jeden univerzální setup form. Na obrázku 5 je vyobrazen setup form pro *RetysTypePlugin*.



Obr. 5. Ukázkový Setup form pro retis type report

4.4. Report viewer



	U1N	U2N	U3N	U12	U23	U31
Napětí[V]						
Průměrný:	243,6	239,3	246,1	418,6	421,4	422,6
Tj. v %nom:	105,9	104,0	107,0	104,6	105,4	105,7
Max. okamžitý:	246,7	241,9	249,0	421,7	424,6	425,5
Čas:	08:11:52 - 11.21.2010	08:11:58 - 11.01.2010	08:11:32 - 11.11.2010	08:11:16 - 11.12.2010	08:11:36 - 11.12.2010	08:11:06 - 11.06.2010
Max. klouzavý (1 min):	246,2	241,2	248,3	421,4	424,4	425,2
Čas:	08:11:00 - 11.06.2010	08:11:02 - 11.12.2010	08:11:00 - 11.11.2010	08:11:16 - 11.12.2010	08:11:18 - 11.12.2010	08:11:58 - 11.05.2010
Min. okamžitý:	237,6	235,7	241,1	237,6	235,7	241,1
Čas:	08:11:16 - 11.24.2010	07:11:52 - 11.58.2010	08:11:14 - 11.01.2010	08:11:16 - 11.24.2010	07:11:52 - 11.58.2010	08:11:14 - 11.01.2010
Min. klouzavý (1 min):	239,9	236,9	242,5	239,9	236,9	242,5
Čas:	07:11:14 - 11.53.2010	07:11:24 - 11.58.2010	08:11:30 - 11.02.2010	07:11:14 - 11.53.2010	07:11:24 - 11.58.2010	08:11:30 - 11.02.2010
Doba nad 115% Unom[%]	0,0	0,0	0,0	100,0	100,0	100,0
Doba pod 90%						

Obr. 6. Zobrazení formuláře report viewer

Tento okenní formulář slouží k zobrazování vytvořených reportů. Lze je dále vytisknout nebo uložit jako PDF soubor pro zálohování v elektronické podobě. Při zobrazení je mu pouze předána instanci reportu, který má zobrazit, v tomto případě je to *RetisTypePlugin*.

5. Pluginy

5.1. PluginBase

Jako základ pro pluginy byla nakonec zvolena abstraktní třída a *PluginBaseclass* je výsledkem. Abstraktní třída byla vybrána vzhledem k již zmiňovaným výhodám, jako je možnost napsání společného kódu v základní třídě. Ve třídě jsou tři proměnné a pět objektů označovaných jako property vyobrazené v kódu 10.

```
protected EnvisFacade Facade;
protected IDataSource dataSource;
string fileName;
public string FileName { get { return fileName; } }
public abstract string Name { get; }
public abstract string Description { get; }
public abstract bool IsSetupAwailable { get; }
public abstract bool AlwaysStart  { get; }
```

Kód 10. Proměnné a property ve třídě PluginBase

Proměnná *Facade* udržuje instanci třídy *EnvisFacade*, která je přechodem mezi datovou a uživatelskou částí softwaru. *DataSource* je typu *IDataSource* a odkazuje na objekt datového zdroje, se kterým se aktuálně pracuje. Nejedná se o objekt konkrétního měření, ale o databázi nebo datový soubor, z kterého se načítájí data.

Proměnná *FileName* typu *string* uchovává cestu ke knihovnímu souboru, z něhož pochází instance pluginu. Cestu je potřeba uchovat kvůli převodu pluginu z používaných do dostupných a naopak. Aby bylo možno se k cestě dostat zvenčí třídy, je zde příslušná property *FileName*.

Poslední 4 property jsou označeny slovem *abstrakt*, takto označené property nemají žádný kód a je potřeba jej doplnit v dědící třídě. První dvě property jsou typu *string* a obsahují název a popis pluginu. Název musí být vložen u každého pluginu a měl by být jednoznačný, popis je pak doplněním toho co plugin obsahuje a provádí za operace.

Poslední dvě property jsou typu *bool* ten může nabývat pouze hodnot *true* a *false*. První pojmenovaná *IsSetupAvailable* vrací hodnotu podle toho, jestli má plugin dostupný setup nebo ne. Podle této property v se *PluginManageru* určuje, zdali je tlačítko *Setup plugin* dostupné (*IsSetupAvailable* je nastaven na hodnotu true) nebo ne.

Hodnota property *AlwaysStart* určuje, jestli má plugin být spustitelný volitelně pokud je hodnota *false*, to znamená, že plugin je vidět v PluginManageru a uživatel ho může zapnout nebo vypnout. Nebo naopak plugin není vidět a je součástí seznamu *AlwaysUsePlugin*, který obsahu seznam pluginů jež běží automaticky po celou dobu běhu aplikace.

V následující ukázce kódu s číslem 11 je seznam metod obsažených ve třídě *PluginBase*, některé mají část kódu implementovanou, jiné je třeba překrýt v dědící třídě.

```
public PluginBaseClass(string FileName)
public void SetDataSource(IDataSource DataSource)
public virtual void InicializePlugin(object Facade)
public override string ToString()
public abstract int LoadData(object Data);
public abstract void Dispose();
public abstract void PluginSetup();
```

Kód 11. Metody ve třídě PluginBase

První metoda s názvem *PluginBaseClass* je contructor třídy, přijímá jeden parametr a tím je cesta k souboru knihovny, ze kterého je plugin inicializován. Další metodou *SetDataSource* se nastavuje pluginům zdroj dat, jenž májí použít pro své operace.

Pomocí metody *InicializePlugin* je provedena ve správný čas inicializace pluginu, pokud jsou nějaké potřeba. Metoda přijímá pouze jeden parametr ten je typu *object*, přestože se očekává, že daný objekt bude typu *EnvisFacade*. Je to z důvodu, že daná metoda může být přetížena v dědící třídě, a bude požadovat jako vstupní parametr objekt jiného typu. Jak již bylo dříve řečeno objekt jakéhokoli typu, lze přetypovat na *object* a zpět proto metoda v tomto formátu předchází případným pozdějším problémům. Uvnitř těla metody pak již jen stačí zjistit, jestli je objekt daného typu,

který je požadován a přetypovat ho zpět na původní typ. K označení, že je funkci možno v dědící třídě přetížit slouží další z klíčových slov, o kterém zde ještě nepadla zmínka, slovo *virtual*. Metoda *ToString* vrací textový řetězec, kterým identifikujeme plugin, v tomto případě vrací hodnotu parametru *name*. Je u ní napsáno dalšího klíčové slovíčko *override*, takto označená metoda nám říká, že překrývá metodu z rodičovské třídy.

Další tři metody jsou všechny abstraktní, takže jejich tělo stejně jako u property musí být vytvořeno v dědící třídě. První z nich se používá pro nahrání různých pomocných objektů pro plugin, opět přejímá parametr typu *object*, takže je schopna přijmout jakákoli data. Druhá metoda se volá při ukončení práce s objektem a musí v ní být naimplementován kód, který se postará o úklid všech zdrojů, které plugin používal. Poslední metoda *PluginSetup* je dostupná jen pokud má plugin možnost vlastních nastavení a spouští se pomocí tlačítka *Setup plugin* na Windows form okně *PluginManageru*. Tato třída je základní pro úplně všechny typu pluginů.

5.2. ReportPluginBase

ReportPluginBase je třídou, od které jsou poděděny všechny pluginy, které slouží jako výstupní reporty. Opět se jedná o třídu abstraktní a sama je poděděna od třídy *PluginBase*. Lze tak říci, že přidává funkčnost k základní třídě, kterou musí poděděné pluginy provádět. Sama také již určitou funkčnost implementuje. Přehled proměnných a property třídy *ReportPluginBase* máme sepsány v ukázce kódu 12, přehled metod pak v kódu 13.

```
bool disposed;
BarSubItem pluginsMenuItem;
public abstract string MenuName { get; }
```

Kód 12. Proměnné, property třídy ReportPluginBase

Třída obsahuje pouze dvě proměnné jednu *disposed* typu bool, ta je využita v destructoru třídy ke zjištění, jestli již byla zavolána metoda *Dispose* nebo, zdali je nutno ji zavolat nyní. Druhá proměnná je pak typu *BarSubItem*, tento typ se používá

jako položka v okenním menu. Do menu je třeba přidat jednotlivé pluginy, aby bylo možno je nějakým způsobem spustit a vytvořit si tak report, který je zrovna aktuálně potřebný.

V třídě je také jedna property *MenuName*, je typu *string* a obsahuje jméno pluginu pro registraci do menu. To je nutné protože jmeno pluginu normální se může shodovat se jménem jiného pluginu, pokud by byly oba pluginy aktivní zároveň, mohlo by dojít ke komplikacím, kterým se takto vyhneme.

```
public ReportPluginBase(string FileName)
~ReportPluginBase()
private void AddPluginToMenu()
private void RemovePluginFromMenu(
public virtual void InicializePlugin(BarSubItem PluginsMenuItem, object
       Facade)
public abstract void MenuClickFunction(object sender,
       ItemClickEventArgs e);
public override void Dispose()
```

Kód 13. Metody třídy ReportPluginBase

Jako první je opět constructor třídy, který přejímá jeden parametr a tím je cesta k souboru. Neimplementuje žádný kód v těle, pouze předá cestu rodičovskému pluginu (*PluginBase*). Další metoda, destructor je poslední metoda třídy, která se spustí před jejím uvolněním z paměti. V její implementaci se pouze testuje, jestli už byla zavolána metoda *Dispose*, případně se zavolá.

Metoda AddPluginToMenu se stará o správné přidání položky pluginu do položek v menu, aby bylo možno k němu přistoupit z prostředí aplikace. Další metodou *RemovePluginFromMenu* se naopak plugin stará o správné odebrání z menu. Přidávání a odebírání z menu si hlídá každý plugin sám, kvůli možnosti vypínat pluginy za běhu. Bylo by sice možno tuto činnost obstarávat pomocí plugin manageru, ale tento způsob je jednodušší a přehlednější.

Metodou, která nám inicializuje plugin předáme pouze objekt menu, do kterého se plugin zaregistruje a objekt typu Facade jako v rodičovské třídě. *MenuClickFunction*

se zaregistruje pro danou položku v menu jako metoda, která se má spustit, pokud uživatel na položku klikne myší. Pomocí této metody je proveden celý pracovní kód pluginu a bude probrána později na konkrétním příkladu. Poslední metodou je opět Dispose, ta má funkci jako v ostatních třídách.

5.3. RetisType plugin

5.3.1. Původní report

RetisTypePlugin je založen na reportu, který byl v předchozím programu pro práci s přístroji. Tento software nesl název Retis. Report sloužil jako obecný přehled o měřených veličinách, konkrétně se jednalo o napětí, proud, výkon, napěťovou nesouměrnost, účiník a spotřebu elektrické energie. Report se vytvářel za nějaké časové období, o kterém bylo potřeba zobrazit údaje. U napětí, proudu a výkonu se zobrazovaly údaje o průměrné hodnotě, maximální a minimální hodnotě, dále maximální a minimální hodnotě, co by klouzavého průměru za zadaný časový úsek. Doby, kterou měřená veličina dosahovala hodnot nad nebo pod hranice určené hodnoty, tato hodnota byla určena v procentech nominální hodnoty. Zobrazen je i počet a doba přerušení dodávky energie. Hodnoty, které se týkají doby trvání nějakého měření, jsou udávány v procentech rozpětí času, ze kterého je report tvořen.

U napětí a proudu se zobrazuje i harmonické zkreslení udávané pomocí činitele harmonického zkreslení THD, opět se jedná o hodnoty průměru, maximální hodnoty a maximální klouzavé hodnoty. THD by mělo být co nejmenší, určuje procentuální podíl vyšších harmonických oproti celému signálu.

Výkon byl rozdělen do 5 sekcí. První dvě zobrazovaly import a export činné složky, další dvě pak import a export jalové složky. Poslední sekce zobrazovala údaje o zdánlivém výkonu.

5.3.2. Nově vytvořený report

Protože se jedná o report, je třída, ze které je plugin děděn, typu *ReportPluginBase*. Při tvorbě byla snaha dodržet stejné grafické rozvržení, jako měl původní report. Při prvních pokusech tvorby tohoto reportu bylo zjištěno, že jeho tvorba

trvá poněkud déle a zamrzá tak celý program. Takový problém je ovšem nepřípustný a bylo potřeba ho odstranit, na to bylo využito systému vláken. Takovýmto způsobem hlavní vlákno zpracovává chod programu a vedlejší se stará o tvorbu reportu. Nakonec bylo toto řešení rozšířeno ještě dále, tak aby se hlavní plugin staral o chod vláken, ve kterých se vytvářejí vlastní pluginy. Tím se dosáhlo možnosti tvořit několik reportů stejného typu, které se tvoří paralelně a nezávisle.

Tvorba reportu se spouští v menu reportů, po stisku na položku menu se jménem reportu je spuštěna metoda *MenuClickFunction*. Zde se nejprve ověří, zdali je fronta vláken vytvořena, v tomto případě ji zastupuje generická kolekce list typu *Thread*. Pokud ne, znamená to tvorbu prvního reportu tohoto typu, v takovém případě se fronta vytvoří. Pokud již existuje, zjistí se, jestli neobsahuje nějaká vlákna, která už provedla svoji činnost, jestliže se tu nějaké takové vyskytují, jsou z fronty odstraněna. Následně je spuštěna metoda *ExecutePlugin*, metoda bude pro většinu reportů shodná a je vyobrazena v kódu 14, v této metodě se nejprve ověří, jestli máme v aplikaci nějaká načtená data, ze kterých je možno plugin vytvořit.

```
private void ExecutePlugin()
{
      SmpObjectDB[] objectDB;
      try
      {
            objectDB = dataSource.GetObjects();
      }
      catch (Exception ex)
      {
            MessageBox.Show("No records to show", Name,
      MessageBoxButtons.OK, MessageBoxIcon.Warning);
            return;
      }
      if ((objectDB == null) || (objectDB.Length == 0))
      {
            MessageBox.Show("No records to show", Name,
      MessageBoxButtons.OK, MessageBoxIcon.Warning);
            return;
      }
      SetupFormData setupData = new SetupFormData(base.dataSource);
      SetupFormPluginTypeRetis setFrm = new
                        SetupFormPluginTypeRetis(ref setupData);
      setFrm.ShowDialog();
      if (!setupData.MakeReport) return;
      Thread workThread = new Thread(new
                        ParameterizedThreadStart(WorkThreadFunctions));
      workThread.SetApartmentState(ApartmentState.STA);
      workThread.Priority = ThreadPriority.Lowest;
      workThread.Start(setupData);
      workingThreads.Add(workThread);
      workThread = null;
}
```

Kód 14. Zobrazení metody execute plugin

Dále se vytvoří nové vlákno, které bude vytvářet report. Toto vlákno je spuštěno s jedním parametrem, kterým je instance typu *SetupFormData*. Tato třída je pro každý plugin jiná, uchovává informace o pomocných proměnných, které jsou potřeba ke tvorbě reportu. Třída dědí od základní třídy *SetupFormDataBase*, ta obsahuje property nutné pro tvorbu reportu, jejich seznam je vidět v kódu 15.

```
public IDataSource DataSource { get; set; }
public SmpMeasNameDB MeasNameDB { get; set; }
public SmpObjectDB ObjectDB { get; set; }
public SmpIdentifyDB IdentifyDB { get; set; }
public bool MakeReport { get; set; }
public DateTime StartDateTime { get; set; }
public DateTime EndDateTime { get; set; }
```

Kód 15. Zobrazení seznamu property pro třídu SetupFormDataBase

Kód 15. zobrazuje property *IDataSource*, ve které je uchován odkaz na instanci aktuálního zdroje dat. Dále property *MeasNameDB*, *ObjectDB* a *IdentifyDB* s jejichž pomocí je možné ve zdroji dat nalézt konkrétní měření. *MakeReport* je použita při nastavování v úvodním formuláři při spuštění pluginu, pokud je nastavena na *true* proběhlo nastavení reportu korektně s platnými daty a může tak být vytvořen. Poslední dvě property *StartDateTime* a *EndDateTime* pak uchovávají údaje o časovém rozpětí, od kdy do kdy má být report vytvořen. V kódu 16, jsou zobrazeny property, o které *SetupFormData* rozšiřuje základ pro tento konkrétní případ.

```
public ESlideWindow USliding { get; set; }
public ESlideWindow PSliding { get; set; }
public float UNom1F { get; set; }
public float UNom3F { get; set; }
public float PNom1F { get; set; }
public float PNom3F { get; set; }
public float INom1F { get; set; }
```

Kód 16. Zobrazení properte ve třídě SetupFormData pro RetisTypePlugin

Udržují se zde hodnoty *USliding* a *PSliding* což je čas, který udává velikost okna pro tvorbu klouzavých průměrů napětí, proudů a výkonů. Napětí a proud mají stejně velkou dobu klouzavého průměru. Další property pak udržují nominální hodnoty pro napětí, proud a výkon, jak pro jednofázové tak třífázové hodnoty. Ty se využívají na dopočítávání procentuálních hodnot nominální hodnoty měřených veličin a také na zjišťování času, po kterou byla měřená veličina pod, nebo nad určitou procentuální hodnotu nominální veličiny. Touto metodou končí veškerá obsluha tvorby pluginu hlavním vláknem, které se vrací k obsluze událostí od uživatele.

Tvorbu reportu přebírá metoda vlákna pro tvorbu reportu i tato metoda bude pro většinu pluginů shodná a proto je zobrazena v kódu 17.

```
private void WorkThreadFunctions(object SetupData)
{
      SetupFormData setupData = (SetupFormData)SetupData;
      ReportDataSet rd = new ReportDataSet();
      RetisReport report;
      try
      {
            report = GetFilledReportData(setupData);
      }
      catch (Exception ex)
      {
            MessageBox.Show("Error in creating report", "Error",
                        MessageBoxButtons.OK, MessageBoxIcon.Error);
            return;
      }
      string name = string.Format("{0}\\{1}\\{2}",
                        setupData.ObjectDB.objekt.ToString(),
                        setupData.IdentifyDB.ToString(),
                        setupData.MeasNameDB.measName.ToString());
      ReportViewerForm reportViewer = new ReportViewerForm(report,
                        name);
            reportViewer.ShowDialog();
        }
```

Kód 17. Hlavní metoda vlákna pro tvorbu reportu

Protože do metody vlákna lze předat pouze parametry typu object, musí se na začátku přetypovat vstupní parametr na správný typ. Následně je zavolána metoda *GetFilledReportData*, která přijímá parametr *SetupFormData* a vrací vytvořený report. Tělo této funkce je v kódu 18. Zde je vytvořena instance *rd* typu *ReportDataSet*, což je *DataSet* pro přenesení změřených dat z výpočetní části vlákna do reportu. Každé měření má připravenu jednu tabulku, do které se uloží data a následně v reportu přepíší do tabulek. Reporty XtraReport používané v práci mají vestavěnu podporu nejen pro databáze, ale i pro typ *DataSet* díky tomu vytvoří tabulku samy z dat, která vložený *DataSet* obsahuje. Poté jsou všechny tabulky v datesetu vyplněny a předány

constructoru třídy *RetisReport*. Jako druhý parametr je předán seznam barev pro řádky a sloupce, v tomto případě je použit defaultní typ. Všechny tabulky jsou plněny pomocí metod třídy *FilledClass* ve které dochází k získání všech dat.

```
private RetisReport GetFilledReportData(SetupFormData SetupData)
{
      ReportDataSet rd = new ReportDataSet();
      FilledClass.SetupData = SetupData;
      DataTable table = rd.Tables["VoltageTable"];
      FilledClass.FillVoltageTable(ref table);
      table = rd.Tables["CurrentTable"];
      FilledClass.FillCurrentTable(ref table);
      table = rd.Tables["ElectricWork"];
      FilledClass.FillElectricWorkTable(ref table);
      table = rd.Tables["Power"];
      FilledClass.FillPowerTable(ref table);
      table = rd.Tables["PowerFactor"];
      FilledClass.FillPowerFactorTable(ref table);
      table = rd.Tables["VoltageUnbalance"];
      FilledClass.FillVoltageUnbalanceTable(ref table);
      table = rd.Tables["StringTable"];
      FillString(ref table);
      return new RetisReport(rd, new RetisReport.Colors());
}
```

Kód 18. Tělo funkce GetFilledReportData

5.3.3. FilledClass

Získání dat pro *RetisTypePlugin* report zajišťuje tato třída. Na pomoc je vytvořena třída *PhaseData*. V té jsou udržovány hodnoty pro jednotlivé fáze, počítá se, kolik vzorků bylo doposud zahrnuto, dobu a počet přerušení, dobu přepětí, podpětí. Od této třídy jsou dále děděny dvě nové třídy. *PhaseDataFloat* využívanou na klasické hodnoty, například pro výpočet průměrné hodnoty, maximální a minimální okamžité hodnoty. U těchto hodnot se dále ukládá čas kdy nastaly, ten je v reportu také zobrazen. Druhá děděná třída nese název *PhaseDataSliding*, je uzpůsobena pro výpočet klouzavých maxim a minim. Toho je docíleno pomocí generického listu, ve kterém je uchováno několik posledních měřených hodnot i s jejich časy. Pokaždé když čas mezi

první a poslední událostí překročí čas určený pro velikost okénka pro klouzavý průměr, je první událost odmazána, takto se počítá skrz celý rozsah naměřených hodnot. Také v tomto případě se ukládá čas začátku okna, které má maximální nebo minimální hodnotu. Obě třídy mají několik metod, všechna měřená data jsou uložena v události typu *SmpArchiveMainDB.* Data jsou k nalezení v různých propertach např. pro napětí jsou to property avg_uLN, avg_uLL pro průměrné napětí za časovou jednotku události, max_uLN a max_uLL pro maximální napětí v dané události a min_uLN a min_uLL pro minimální napětí. Pro každou z měřených veličin je obsaženo několik propert, ve kterých jsou uložena data pro každou z fází. Metody v pomocných třídách je proto potřeba přetížit tolikrát kolik typů hodnot má změřit, tak aby se vzala správná data z měřených událostí.

Navíc zde existuje ještě třída *TableNameClass*, v ní je pro každý řádek uchován text náležející hlavičce, barva pozadí daného řádku, barva textu v daném řádku a informace o tom, jestli je řádek prázdný nebo plný. Tato informace je uchovávána kvůli řádkům, které slouží jako nadpis další sekce tabulky a nemají tak v jednotlivých sloupcích žádné hodnoty. Pro každou tabulku je zde pole instancí této třídy a hodnoty z nich jsou vkládány při tvorbě datasetu do tabulky, ke které náleží. Naplnění těchto instancí daty je prováděno v metodách *FillVoltageName, FillCurrentName, FillEletricWorkName, FillPowerSeparateName, FillPowerCompleteName, FillPowerFactorName, FillVoltageUnbalanceName*. Příslušná metoda je spuštěna na začátku metody s příslušným měřením.

Protože se všechny metody měření podobají, jsou rozdílné jen v přístupu k datům a v počtu řádků, které jsou zapsány do reportu, je zde detailně uvedena pouze jedna z nich. Pro ukázku je zvolena tabulka s údaji napětí, kde jsou použity obě pomocné třídy a měří se průměrné, minimální i maximální hodnoty. Metoda se jmenuje *FillVoltageTable* a přijímá jeden parametr typu *DataTable*, což je tabulka, která má být naplněna daty. Metoda je v kódu 19, není zde zobrazen celý kód, zabral by totiž několik stránek. Zobrazuje se od všech podstatných událostí minimálně jeden výskyt, na kterém je vysvětleno o co se jedná.

Na začátku se pustí metoda *FillVoltageName,* která přijímá parametr typu int, který udává velikost klouzavého okénka v minutách. Následně se nadeklarují pomocné proměnné. Instance arch třídy *ArchDescriptionDB*, obsahuje údaj o archivu, který má

být získán. Další položka *col* pak všechna měření daného archivu, která jsou uvnitř hledaného časového úseku, v zadaném zdroji typu *IDataSource*. Následně je zobrazeno nastavení okénka pro klouzavý průměr u instance *minimalniKlouz*, která je typu *PhaseDataSliding*. V další části se postupně procházejí všechny naměřené záznamy. Nejprve se do dočasných proměnných uloží, maximální a minimální hodnota, buď z hodnoty maxim a minim, jsou-li dostupné, jinak se nastaví z průměrné hodnoty. Následně je vidět přehled metod pro vkládání hodnot různých typů do instancí pomocných tříd *PhaseDataFloat* a *PhaseDataSliding*.

```
FillVoltageName((int)SetupData.USliding
.
.
ArchDescriptionDB arch = new ArchDescriptionDB(KMB.Structures.SMP.Sm
      pArchiveTypes.ARCH_MAIN);
RowCollection col = setupData.DataSource.GetRows(SetupData.MeasNameD
      B, arch,
      new DateRange(SetupData.StartDateTime, SetupData.EndDateTime),
      setupData.DataSource.DefaultSession,
      numberRecordsToGetFromDB, false);
.
.
 minimalniKlouz.SetSlideWindow(new TimeSpan(0, (int)setupData.USlidi
ng, 0));
.
.
foreach (SmpArchiveMainDB evnt in col)
{
      maxULN = evnt.max_uLN == null? evnt.avg_uLN:evnt.max_uLN;
.
.
      average.AddValue(evnt.avg_uLN, evnt.avg_uLL);
      maximalni.SetIsMaxValue(maxULN, maxULL, evnt.key.TimeLocal);
      minimalni.SetIsMinValue(minULN, minULL, evnt.key.TimeLocal);
      maximalniKlouz.CountMaxSlidingValue(maxULN, maxULL, evnt.key.T
                imeLocal);
      minimalniKlouz.CountMinSlidingValue(minULN, minULL, evnt.key.T
                imeLocal);
      nad115Unom.OverValue(evnt.avg_uLN, evnt.avg_uLL, SetupData.UNo
                m1F * 1.15f, evnt.timelength);
      interuptCount.CheckInterupt(evnt.avg_uLN, evnt.avg_uLL, evnt.t
                imelength);
      thdAverage.AddValue(evnt.avg_uTHD);
}
```

Kód 19. Ukázka z metody získání dat pro měřená napětí

```
average.CreateAverage();
.
.
nad115Unom.UnderOverPercentCount(wholeTime);
.
.
actualTableName = voltageName;

int index = 0;
TableAddDataValue(ref Table, ref index, null, average);
TableAddDataValue(ref Table, ref index, null, PhaseDataFloat.CreateN
     ominalPerception(average, SetupData.UNom1F, SetupData.UNom3F));
```

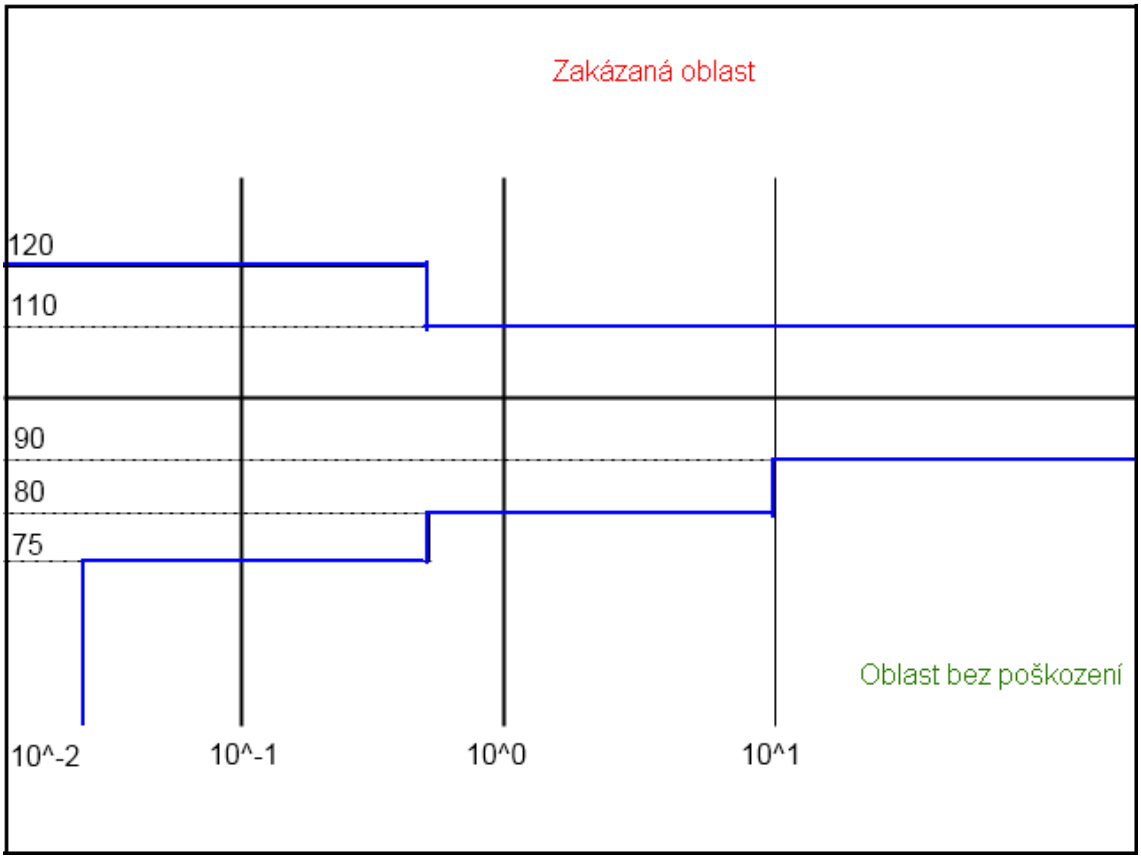
Kód 20. Dokončení ukázky metody získání dat pro měřená napětí

V kódu 19 je dokončení metody. Před zadáním hodnot do tabulky se musí provést ještě několik úprav, hodnoty, které mají být průměrné, se zprůměrují. Hodnotám, které jsou závislé na celkové době měření, předáme celkový čas. Poté je nastaveno pole, ve kterém jsou uloženy texty pro hlavičky řádků, pro měřenou tabulku. Následně je pomocí metody *TableAddDataValue* přidána změřená veličina jako jeden řádek tabulky. Přijímá parametr *Table*, kterým je tabulka z datasetu určená pro uchování dat daného typu. Také přijímá parametr *index* typu *int* uchovávající index řádku tabulky přiřazení správného textu hlavičky k řádku. Tento parametr je předám jako reference, důvodem je fakt že aktuální řádek, na který odkazuje, může být prázdný s popisem sekce. Uvnitř metody tak vložíme prázdný řádek, *index* zvětšíme o jedna, vložíme řádek s daty a opět o jedničku inkrementujeme. Je potřeba, aby se případné dvojité inkrementování promítlo i v metodě *FillVoltageTable* kvůli dalším řádkům tabulky.

Po vyplnění všech tabulek daty, je dataset předán reportu jako zdroj dat, ze kterého vygeneruje report. Kde má jaká data zobrazit se nastavuje při sestavování reportu pomocí designeru, zde se určuje jednotlivým sloupečkům tabulek, z jaké tabulky datasetu a jakého sloupce mají čerpat data. Tabulka může mít i méně sloupečků než má daná tabulka v datasetu a zobrazovat tak jen některé údaje. Vygenerovaný ukázkový report je přiložen na CD.

5.4. Power quality event report

Další z ukázkových reportů se zaměřuje na zobrazování třech typů událostí. Těmito událostmi jsou přepětí, podpětí a výpadky napětí. Tyto události jsou sledovány kvůli jejich negativnímu vlivu na elektrické přístroje. Každý přístroj má dané určité nominální napětí a je schopen správně fungovat pouze v určitém rozpětí kolem tohoto nominálního napětí. Hodnoty, které jsou pod nebo nad touto hranicí, nezaručují správnou činnost přístrojů, což může v některých případech vést i k úrazům. Hlavním problémem ale zůstává činnost přístrojů nad maximální hranicí, která se nazývá zakázaná oblast. Pokud přístroj pracuje při napětích, v této oblasti dochází k jeho poškození. Na Obr. 7 je znázornění oblastí působení. Na ose x je zobrazen časový údaj v sekundách, na ose y jsou procenta nominálního napětí. Hodnoty nad horní modrou linkou jsou v zakázané oblasti, pod dolní modrou linkou pak v oblasti bez poškození. Pokud se hodnota pohybuje mezi modrými linkami, je vše v pořádku.



Obr. 7. Zobrazení oblastí pro PQ eventy

Protože je tělo pluginu z většiny stejné jako předcházející plugin, bude zde popsána pouze část, které se zabývá získáním dat a tvorbou reportu. Tvorba reportu začíná v metodě *GetFilledReportData*, ta opět přebírá parametr *SetupFormData*, ten má některé parametry jiné než u *RetisTypePlugin*. Základní parametry opět dědí od *SetupFormDataBase*, a přidává k nim 3 hodnoty, které nám tvoří 4 časové úseky. Tělo metody *GetFilledReportData* je zobrazeno v kódu 21. Na začátku vytvoříme instanci *arch* typu *ArchDescriptionDB* tentokrát vytvořené s parametrem udávajícím, že vrácené měření mají být z archivu PQEvents. Získáme seznam měřených událostí. Pokud je seznam *null* není buď měření v tomto přístroji obsaženo, nebo nebyla žádná data změřena, proto vyhodíme výjimku. Vytvoříme si instance *phasesVoltageSwell*, *phasesVoltageDip*, *phasesVoltageFailure* typu *PhaseEventData*, tato třída má 4 hodnoty a to počet událostí pro každý z časových úseků. Následně se prochází skrz seznam uložených událostí, pro jistotu kontrolujeme, jestli je správně uvedeno číslo fáze a jakého typu je událost. Typ události s číslem 1 jsou přepětí, s číslem 2 podpětí a s číslem 3 se jedná o výpadky. V metode *CreatePhaseData* z třídy *Utils*, zjístíme do jakého časovéhu useku událost spadá a pomocí metody *AddEventCount* ze třídy *PhaseEventData* přičteme hodnotu k již stávajícím hodnotám.

```
ArchDescriptionDB arch = new
ArchDescriptionDB(KMB.Structures.SMP.SmpArchiveTypes.ARCH_PQEVENTS);

RowCollection col =
setupData.DataSource.GetRows(setupData.MeasNameDB, arch,
new DateRange(setupData.StartDateTime, setupData.EndDateTime),
      setupData.DataSource.DefaultSession, 0, false);
if (col == null)
throw new Plugin_PQEventPlugin.BadDataException("Bad device");
PhaseEventData[] phasesVoltageSwell = new PhaseEventData[4];
.
.
foreach (SmpArchivePQEventDB evnt in col)
{
  if ((evnt.phaseN > 0) && (evnt.phaseN < 5) && (evnt.type == 1))
      phasesVoltageSwell[evnt.phaseN -
            1].AddEventCount(Utils.CreatePhaseData(evnt.Trvani,
            setupData));
.
.
}
return new PQEventReport(setupData, phasesVoltageSwell,
      phasesVoltageDip, phasesVoltageFailure);
```

Kód 21. Tělo metody GetFillReportData v PQ Evnt plugin

Nakonec je vytvořen samotný report, v tomto případě není použito třídy dataset, ale předávají se přímo instance *SetupFormData* a pro každý typ události jedna instance typu *PhaseEventData*. V reportu je v tomto případě v designeru při vytváření vzhledu místo tabulky vloženo několik labelů, ty jsou následně naplněny při vytvoření reportu daty z vložených instancí. Je tak ukázáno, že data lze předat i jinak než jen pomocí datasetu. I tento report je přiložen na CD.

5.5. Device list report

Posledním z ukázkových pluginů je device report list. Tento report vypíše seznam vše přístrojů, které jsou uloženy v databázi a ke každému z nich vypíše pro všechny configy všechny dostupné hodnoty nastavení. Jedna se o *SmpConfig*, v něm

jsou uloženy všechny hlavní nastavení přístroje. *SmpArcConfig*, ve kterém jsou uloženy nastavení a pravidla pro archivaci záznamů. *SmpInstallConfig* obsahuje informace o instalaci přístroje. *SmpElmerConfig*, zde jsou uloženy informace pro elektroměr a jeho tarify. Nakonec pak *SmpPQSettings*, kde jsou nastavení a pravidla pro ukládání power quality událostí. I pro tento plugin platí, že je z většiny stejný jako předcházející pluginy. Proto je zde opět zmíněna pouze metoda *GetFilledReportData* které je v kódu 22.

Pro předání dat do reportu je použito *DeviceListDataSet*, jenž je typu *DataSet*. V tomto datasetu máme jednu tabulku pro přístroje, ta obsahuje sloupečky *DeviceID*, *ObjectName*, *DeviceName*, *DeviceNumber*, jenž je sériové číslo přístroje a *DeviceTyp*, ve kterém je uveden typ přístroje s příslušenstvím. Následně je zde pět tabulek pro každý typ konfigu jedna, každá z nich má čtyři sloupečky *PropertyID*, *DeviceID*, podle toho je určeno, ke kterému přístroji tato položka patří. Dále pak název položky *PropertyName* a nakonec její hodnota *PropertyValue*.

Nejprve se inicializuje instance pomocné třída *ReportData*, ta v sobě uchovává instanci zmiňovaného datasetu a dále obsahuje dvě metody, pomocí *AddDevice* přidáme přístroj do tabulky přístrojů. Metoda přijímá pět parametrů, které jsou stejné jako sloupečky v tabulce datasetu. Druhá metoda AddConfigProperty přijímá parametry čtyři, prvním je jméno tabulky, která je vyhrazena pro druh vkládaných dat. Druhý parametr je id přístroje, ke kterému záznam patří, třetí parametr jméno proměnné a čtvrtý její hodnota.

Do dočasné proměnné si vložíme id pro první přístroj. Následně v cyklu projdeme všechny objekty, které máme v databázi uloženy a pro každý z objektů projdeme všechny přístroje, které obsahuje. Každý z nalezených přístrojů je přidán do dat pro report. Dále jsou prohledána všechna měření a pro každé měření je získána poslední konfigurace všech configů, z těchto konfigurací je nakonec vybrána ta, která byla na přístroji jako úplně poslední, je pravděpodobné, že je stále ještě na přístroji uložena.

```
ReportData reportData = new ReportData();
int deviceID = 1;
foreach (SmpObjectDB objectDB in dataSource.GetObjects())
{
foreach (SmpIdentifyDB identidfyDB in dataSource.GetIdents(objectDB))
{
      reportData.AddDevice(deviceID, objectDB.ToString(),
            identidfyDB.ToString(), identidfyDB.DeviceNo.ToString(),
                  identidfyDB.Identifycation.ToString());
      ConfigsClass configs = null;
            ArchDescriptionDB[] archives =
            ArchDescriptionDB.GetArchiveListByDeviceClass(identidfyDB.
            Identifycation);
      foreach (SmpMeasNameDB measName in
            dataSource.GetRecords(identidfyDB))
      {
         ConfigsClass tmpConfigs = GetLastConfig(measName, archives);
         if (configs == null) configs = tmpConfigs;
         if (configs.Time < tmpConfigs.Time) configs = tmpConfigs;
      }
      PropertyInfo[] propInfo =
          configs.Configs.arcConfig.GetType().BaseType.GetProperties(B
          indingFlags.Public | BindingFlags.Instance |
          BindingFlags.DeclaredOnly);
      foreach (PropertyInfo property in propInfo)
            reportData.AddConfigProperty("SmpArcConfig", deviceID,
                 property.Name,
                 property.GetValue(configs.Configs.arcConfig,
                 null).ToString());
.
.
      deviceID++;
}
}
return new DeviceListReport(reportData);
```

Kód 22. Kod třídy GetFilledReportData pro plugin device list

Pro každý z configů je pomocí reflexe získán seznam všech proměnných, které obsahuje. Tyto proměnné, jsou pak postupně zapsány do příslušných tabulek. V kódu je

zobrazeno přidání propert do tabulky *SmpArcConfig*. Pro ostatní tabulky je kód obdobný. Mění se pouze název configu, ze kterého jsou získány property a název tabulky, do které jsou následně vkládány. Tento plugin byl tvořen jako poslední a v době dokončení této práce zobrazuje jména a hodnoty propert tak jak jsou uloženy ve třídě daného configu. V takovém tvaru jsou tyto daje pro uživatele nesrozumitelné a je třeba převést jak název proměnné, tak její hodnotu na srozumitelný výraz. Protože je těchto hodnot velké množství, bude tato úprava vyžadovat nějaký čas. Report s nepřeloženými názvy a hodnotami je také přiložen na CD.

6. Závěr

Pracnost tématu byla nakonec o něco složitější, než bylo předpokládáno, naštěstí se podařilo hlavní plánované bloky s úspěchem vytvořit. Plugin manager velice dobře odděluje část pluginů od části základního softwaru velice dobře, nepřicházejí spolu fyzicky do žádného kontaktu. Správu pluginů zvládá také velmi dobře a díky změnám provedeným během práce ve spravování šetří maximum systémových prostředků.

Během testů se nevyskytl ani jeden problém související s přidáváním nebo odebíráním pluginu za chodu ani při jejich okamžitém následnému spuštění. Pokud došlo k nějakému pádu pluginu při zpracování dat, tyto pády byly postupně odladěny, aby nebyla hlavní aplikace vůbec nijak zasažena. Prokázalo se tak, i když neúmyslně, že oddělení pluginů od aplikace je opravdu dobře provedené. Díky běhu pluginů ve vlastních vláknech není chod aplikace nijak omezen a ta tudíž nezamrzává. Vytvořené reporty je možno zobrazit a následně vytisknout nebo převést do formátu PDF a uložit přímo z formuláře pro zobrazení reportu.

Jelikož nebyla grafika vytvářena zkušeným profesionálem, je grafické zpracování bohužel na nižší úrovni, než by bylo vhodné. Pro názorné ukázky jsou ale reporty dostatečné. Je v nich prezentována hlavně možnost zobrazit opravdu jakýkoli typ dat z databáze.

Použitá Literatura

[1] Stránka o tvorbě pluginů [online], http://programujte.com/?akce=clanek&cl=2006041802-c

[2] Miroslav Virius: C# - Hotová řešení. Computer Press 2006

[3] Christian Nagel, Bill Evjen, Jay Glynn, Morgan Skinner, Karli Watson, Allen Jones: C# 2005 Programujeme profesionálně. Computer Press 2006

[4] Knihovna MSDN od Microsoft. [online], http://msdn.microsoft.com/cs-cz/default.aspx

[5] Informační server o programování. [online], http://www.builder.cz/

[6] Stránky společnosti Developer Express. [online], http://www.devexpress.com/