VYSOKÁ ŠKOLA POLYTECHNICKÁ JIHLAVA

Katedra elektrotechniky a informatiky

Obor Počítačové systémy

Aplikace pro evidenci majetku

bakalářská práce

Autor: Radomír Šimek

Vedoucí práce: Ing. Marek Musil

Jihlava 2015

Vysoká škola polytechnická Jihlava

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Autor práce:	**Radomír Šimek**
Studijní program:	Elektrotechnika a informatika
Obor:	Počítačové systémy
Název práce:	**Aplikace pro evidenci majetku**
Cíl práce:	Cílem práce je vytvořit desktopovou aplikaci umožňující evidovat majetek a uchovávat jeho pozici a historii umístění. Aplikace bude dále umožňovat plánování nákupu nového majetku podle finančního rozpočtu a hlídání rozpočtu pro finanční rok, dále bude umožňovat tisk objednávek pro nákup nového majetku a tisk předávacího protokolu. Předávací protokol bude možné tisknout v případě, že majetek bude přesunut na nové umístění. Aplikace bude vytvořena v prostředí .NET/C#.

Ing. Marek Musil
vedoucí bakalářské práce

Ing. Bc. Michal Vopálenský, Ph.D.
vedoucí katedry
Katedra elektrotechniky a informatiky

Abstrakt

Práce provází postupem tvorby aplikace pro správu a evidenci majetku. Aplikace bude mít za úkol sjednocovat podrobné informace o jednotlivém majetku a kontrolovat jeho pohyb ve firmě. Uživatel bude mít přehled o aktuální pozici majetku a zároveň si bude moci zobrazit historii, kde již byl majetek používán. Součástí aplikace budou funkce, které usnadní práci při vyhledávání, hlídání finančního rozpočtu nebo převádění majetku mezi jednotlivými místy. Převádění majetku a pořizování majetku je v aplikaci kontrolováno systémem předávacích protokolů a objednávek.

Klíčová slova

.NET Framework, C#, Evidence, majetek, MSSQL, rozpočet, správa

Abstract

This work shows the procedure of creating application for asset administration. The main purpose of application is hold detail information about asset in the company and its move. User will have an overview of the current position of the asset and at the same time will be able to view the history of where the asset was used. Application components are features that facilitate the work for the discovery asset, monitoring financial budget or transfers of assets between locations. Movement of assets is controlled by system of transfer and order protocol.

Key words

.NET Framework, administration, Asset, C#, budget, registration, MSSQL

Prohlašuji, že předložená bakalářská práce je původní a zpracoval/a jsem ji samostatně. Prohlašuji, že citace použitých pramenů je úplná, že jsem v práci neporušil/a autorská práva (ve smyslu zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů, v platném znění, dále též „**AZ**“).

Souhlasím s umístěním bakalářské práce v knihovně VŠPJ a s jejím užitím k výuce nebo k vlastní vnitřní potřebě VŠPJ.

Byl/a jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje **AZ**, zejména § 60 (školní dílo).

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

Jsem si vědom/a toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem VŠPJ, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených vysokou školou na vytvoření díla (až do jejich skutečné výše), z výdělku dosaženého v souvislosti s užitím díla či poskytnutí licence.

V Jihlavě dne

.....

Podpis

Poděkování

Na tomto místě bych rád poděkoval svému vedoucímu práce Ing. Marku Musilovi za poskytnutí podpory pro vlastní téma a možnosti vytvářet práci pod jeho vedením.

Obsah

1 Úvod

Jsem pracovníkem firmy GLS na pozici správce počítačové sítě. Jako jeden z členů IT oddělení mám na starost správu veškerého IT majetku. Ke každému nově příchozímu IT majetku musíme vytvořit záznam o značce, typu a sériovém čísle daného zařízení. Dále je nutné zaznamenat datum nákupu, vytvořit jedinečné registrační číslo a zapsat, kde bude majetek umístěn. V současné době registrujeme více než 1200 kusů IT majetku, který je umístěn buď ve firmě, nebo je půjčen našim subdodavatelům a zákazníkům. Kontrola takového množství majetku již není jednoduchá a vyžaduje nemalé úsilí a investici do času zaměstnance. Abychom snížili náklady na čas zaměstnance, který je nucen trávit průměrně 30 minut denně evidencí, či přesunem majetku, rozhodl jsem se, najít nějaký softwarový produkt, který by nám usnadnil práci a zároveň zajistil lepší kontrolu.

Nový softwarový produkt by měl umět majetek evidovat, umisťovat ho do konkrétních pozic a zároveň mezi pozicemi přesouvat. Další nadstavbové funkce aplikace budou vyhledávání majetku podle zadaných kritérií, evidence objednávek nebo kontrola finančního rozpočtu.

2 Současný stav

V dnešní době snad neexistuje firma, která by ke své práci nepotřebovala podpůrná zařízení jako počítač, tiskárnu nebo kopírovací stroj. V případě malé firmy není třeba speciální kontrola nad všemi těmito zařízeními. Pro kontrolu data nákupu nebo sériového čísla jednotlivých zařízení nám postačí obyčejný Excel. Ale pokud se firma rozroste, tak je již nutné zvolit nějaký komplexní nástroj na evidenci těchto zařízení.

Na internetu nalezneme mnoho softwarových produktů, které jsou schopné vést inventuru a administraci majetku. Bohužel většina nalezených produktů je robustní a pořizovací náklady nejsou nízké, anebo nesplňují požadavky naší firmy. Typickým příkladem velkého produktu je produkt SAP, který má modul pro inventarizaci. Ale toto řešení je velmi drahé a práce s ním není jednoduchá.

Mezi nástroje, které jsou dostupné zdarma, patří například zajímavý program GLPI. Tato aplikace je určena pro IT správce a umožňuje podrobnou evidenci IT majetku. Ale je spíše specializován na přehled prostředků, které má IT oddělení pod svojí správou. Např. u počítače lze zadávat podrobné informace o nainstalovaném software a podobně. Software GLPI je postaven na PHP a My SQL.

Vzhledem k tomu, že velmi často poskytujeme majetek našim subdodavatelům, tak je přikládán velký důraz na evidenci toho, co a kdy bylo půjčeno. Při každé půjčce je vyžadován podepsaný předávací protokol od subdodavatele nebo zákazníka. Výše zmíněný program GLPI, nemá implementovanou podporu tisku předávacího protokolu v takové formě, kterou bychom požadovali.

Dalším důvodem pro vytvoření nové aplikace je důraz na snadné a jednoduché ovládání. Aplikace musí být napsána tak, aby co nejméně zaměstnávala pracovníka IT oddělení (je kladen velký důraz na rychlost převodu položek a tisk předávacího protokolu). Proto by měla aplikace umět efektivně přesouvat položky mezi pozicemi a zároveň při tomto převodu vytvářet předávací protokol, který bude obsahovat informace jak o předávajícím, tak i o přebírajícím.

Stávající program používaný v naší firmě je pouze nadstavba nad mdb-databází. Tento program bohužel nedokáže efektivně vyhledávat informace v databázi, přesouvat více položek najednou a sledovat IT rozpočet. Reporty se nedělají přímo v programu, ale pouze přes nástroje MS Access.

Tento postup není příliš efektivní a díky nekontrolovanému přístupu do databáze[1] může dojít ke smazání údajů buď z důvodu uživatelské chyby, nebo mohou být data odstraněna záměrně.

Tisk předávacího protokolu není také triviální. Existují určité šablony, do kterých musí uživatel vyplnit ručně jméno předávajícího a příjemce. Dále je nutné zadat jednotlivé položky, které se předávají a k nim informaci o počtu předávaného majetku. Tento protokol je po vytištění umístěn na síťový disk pod určitým názvem.

V případě, že chce uživatel získat seznam položek na konkrétním umístění, musí upravit dotaz do mdb databáze a nakonec vyexportovat do excelového sešitu. Tato složitá procedura se děje při jakémkoliv speciálním dotazu do databáze.

Nicméně, ale i toto řešení bylo zlepšením, protože dříve se veškerý IT majetek evidoval v pouze v Excelu. Nová aplikace by měla odstranit výše zmíněné neduhy současného řešení a zprůhlednit tok majetku. Přemístěním databáze ze síťového disku pod SQL server omezíme přístup k datům, která jsou v databázi umístěna. Zároveň zaručíme centralizaci databáze a znemožníme duplikaci MDB databáze, která nyní může nastat chybou uživatele.

Při evidenci nového majetku se vygeneruje nové jedinečné číslo, pod kterým je majetek zaevidován. Toto číslo, index, se používá při zakládání majetku do šanonu. Každý majetek má svoji speciální složku, která je označena právě tímto indexem. V případě, že je majetek přesunut do nového umístění, tak se podepsaný předávací protokol přidá do složky majetku. Tato složka slouží i k dalším dokumentům jako záruční nebo dodací list.

Z důvodu zpětné kompatibility bude nutné uchovat způsob papírové dokumentace a generování indexu pro nový majetek.

[1] Provádění některých úkonů přímo v MS Access.

3 Analýza požadavků

Kapitola shrnuje požadavky na aplikaci jak z pohledu uživatele, tak i z pohledu programátora. Nejdříve je nutné určit dostupné technologické prostředky, které budou pro vývoj aplikace použity. Dále je nutné definovat požadavky na aplikaci ze strany uživatele a nakonec provést rozbor jednotlivých funkcí programu.

3.1 Prostředí implementace a použité technologie

Aplikace bude vytvořena v prostředí *Microsoft .NET* ve verzi 4.5.1, v programovacím jazyce C#. Jako databázové uložiště bude použit MS SQL Server ve verzi 2008.

Microsoft .NET Framework je nejrozšířenější platforma pro osobní počítače nebo chytré telefony s operačním systémem Windows. Toto prostředí se snaží ulehčit vývojářovu práci a nabízí mnoho programových knihoven a spouštěcí prostředí.

Jazyk C# byl navržen právě pro platformu .NET a na rozdíl od ostatních jazyků, které .Net využívají, neobsahuje svoje vlastní knihovny. Ale právě proto, že byl C# vyvíjen společně s .NET, dokáže využít vlastnosti tohoto frameworku v plné šíři.

Framework .NET, i když je produktem Microsoftu, je nezávislý na hardwarové platformě. Zdrojový kód je překládán do mezi-jazyka IL (bajtového kódu virtuálního stroje) a teprve před spuštěním se překládá do strojového kódu cílového počítače. Jediné, co vyžaduje, je platforma .NET. V současné době existuje i open source projekt MONO, který se snaží implementovat jazyk C# i na operační systémy Unixového typu.

Další výhodou .NET je spolupráce více jazyků při programování. Například knihovnu z Visual C++ je možné využít v projektu, který je napsán pomocí programovacího jazyka C#. Právě nesporné výhody platformy .NET a C#, stály za rozhodnutím, využít tento programovací jazyk při mé bakalářské práci.

3.2 Požadavky na aplikaci

Názvosloví použité v textu

Položka je nejčastěji hmotný IT majetek, který bude evidován podle registračního čísla. Nicméně aplikace může být využita pro obyčejný hmotný majetek nebo i pro evidenci spotřebního zboží.

Umístění nebo pozice se neomezuje pouze na fyzický prostor, kde je položka umístěna (např. kancelář), ale umístěním může být i uživatel, který má danou položku ve správě.

Dodavatel je typicky firma, která dodává majetek (položky).

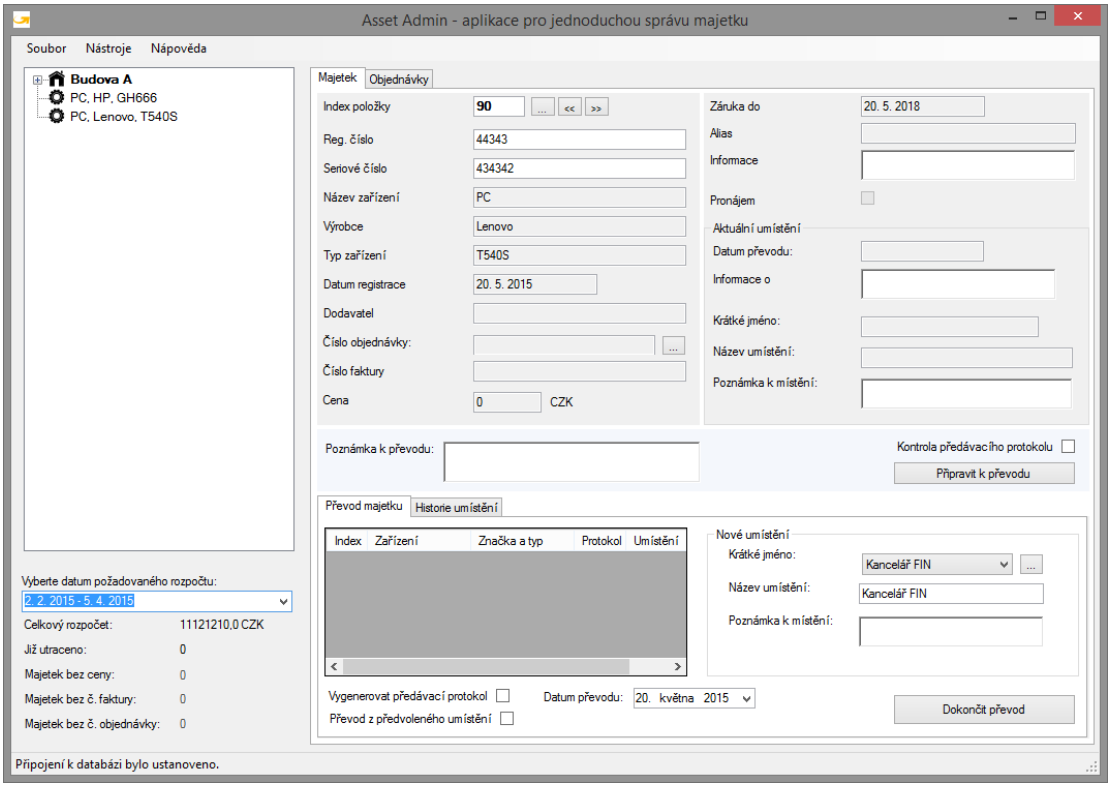
Objednávka je závazný dokument k objednání zboží. Každá objednávka má své originální číslo. Toto číslo bude později využito při párování s číslem faktury.

Historie umístění seznam míst, kde byl jednotlivý majetek používán za určité období.

Rozpočet udává celkovou sumu, která může být použita na nákup jednotlivého majetku.

Zadání požadavků z pohledu uživatele

Aplikaci bude moci provozovat na osobním počítači s operačním systémem Windows 7 nebo vyšším. Data budou ukládána do databáze MS SQL. Aplikace bude v základu postavena jako jednouživatelská a nebude třeba žádné ověření uživatele vůči autentizační službě.



Obrázek 1. Základní okno aplikace

Evidence a editace majetku

Základní vlastností aplikace bude udržení uceleného přehledu nad hmotným majetkem. Uživatel bude moci majetek jak přidávat, tak editovat. Každý nově příchozí majetek dostane originální číslo. Toto číslo bude použito jako reference pro zakládání papírových podkladů jako předávací protokol nebo záruční list. Při vkládání nové položky nebo editaci stávající, bude moci uživatel vyplnit pole určující vlastnosti a typ majetku. U nového majetku bude moci uživatel vybrat inicializační umístění. Typicky tímto inicializačním místem bývá sklad. V případě editace majetku již toto umístění nebude možné měnit.

Vlastnosti určující typ majetku:

- Sériové číslo.
- Registrační číslo.
- Název zařízení.
- Výrobce.
- Typ zařízení.
- Datum registrace.
- Dodavatel – program bude mít vlastní databázi dodavatelů.
- Číslo objednávky.
- Číslo faktury.
- Cena.
- Záruka.
- Dodatečné informace.
- Pronájem.

Evidence umístění

Umístění je fyzické místo nebo uživatel. Umístění může seskupovat jednotlivé položky nebo další umístění do stromové struktury.

Příkladem umístění může být místo *Finanční oddělení* vnořeno do umístění *Budova A.*

Základní vlastnosti umístění jsou:

- Krátké jméno – slouží pro vyhledávání.
- Jméno – je celé jméno.
- Adresa.
- IČ a DIČ.
- Poznámka.
- Je zákazníkem – použije se v případě, že umístění je naším zákazníkem.

Evidence dodavatelů

Dodavatel je použit při evidence majetku a tvorbě objednávky. Dodavatele bude moci přidávat a editovat.

Dodavatel je reprezentován:

- Jménem.
- Adresou.
- IČ, DIČ.

Evidence objednávek

Pokud chceme ve firmě objednat nové zařízení, je třeba vygenerovat objednávku, která obsahuje seznam položek k nákupu. Každá objednávka musí být evidována pod jednoznačným identifikátorem. Při nákupu se tento identifikátor předává dodavateli a dodavatel je povinen ho vložit i do faktury. Na základě této informace můžeme párovat schválené objednávky se skutečnými fakturami a tak se vyhnout neschváleným nákupům.

Objednávka by měla obsahovat:

- Dodavatele, u kterého je majetek nakupován.
- Osobu, která objednávku tvořila.
- Seznam položek k objednání.
- Název objednávky.

Položka v objednávce by měla obsahovat:

- Počet kusů.
- Název položky.
- Cenu za jeden kus.

Funkce vyhledávání jednotlivého majetku

Tato funkce bude moci vyhledat jednotlivé položky na základě zadaného výrazu. Vyhledávání nemusí být fulltextové. Vyhledané výsledky bude moci uživatel vyexportovat do *csv*[2] souboru.

Zvolené sloupce, podle kterých bude moci uživatel hledat:

- Číslo objednávky.
- Číslo faktury.
- Datum posledního převodu.

[2] CSV soubor, je textový soubor, kde hodnoty v jednotlivých řádcích jsou odděleny speciálním znakem např. ";".

- Datum registrace.
- Názvu zařízení.
- Registračního čísla.
- Sériového čísla.
- Dodavatele.
- Typu zařízení.
- Umístění.
- Výrobce.
- Záruky.

U datumových položek je možné vybrat časový interval.

Funkce převodu majetku do nového umístění

Tato funkce je nejdůležitější částí programu. Uživatel bude moci převést vybraný majetek do nového umístnění. Převádět se může pouze vybraná položka, pokud uživatel převádí více položek najednou, je nutné si je připravit k převodu a přidat do dočasného seznamu.

Po dokončení převodu program vygeneruje předávací protokol. Zakázání generování protokolu může uživatel ovlivnit pomocí zaškrtávacího tlačítka. Při převodu se volí datum převodu a poznámka k převodu.

Funkce zobrazení historie majetku

O každém přesunu majetku je vytvořen jednoduchý záznam. Tento záznam obsahuje informace o časovém úseku, po který byl majetek na určité pozici. Seřazené záznamy podle data o přesunu jedné položky, bude uživatel moci zobrazit v jednoduché tabulce.

Funkce hlídaní rozpočtu

Další užitečnou vlastností programu bude funkce hlídání rozpočtu, která má na starost zobrazit informace o majetku, který byl zaregistrován v určitém období. Uživatel si tak bude moci navolit časové úseky a program zobrazí sumarizované informace ze zadaného období.

Příkladem může být, kolik bylo již vyčerpáno nebo kolik položek je zadáno bez pořizovací ceny.

3.3 Rozbor a návrh řešení

ERA model

Na základě požadavků od uživatele bylo nutné rozdělit jednotlivé objekty na menší objekty a udělat hrubý návrh. Tedy z textu plyne, že v aplikaci budeme používat následující objekty:

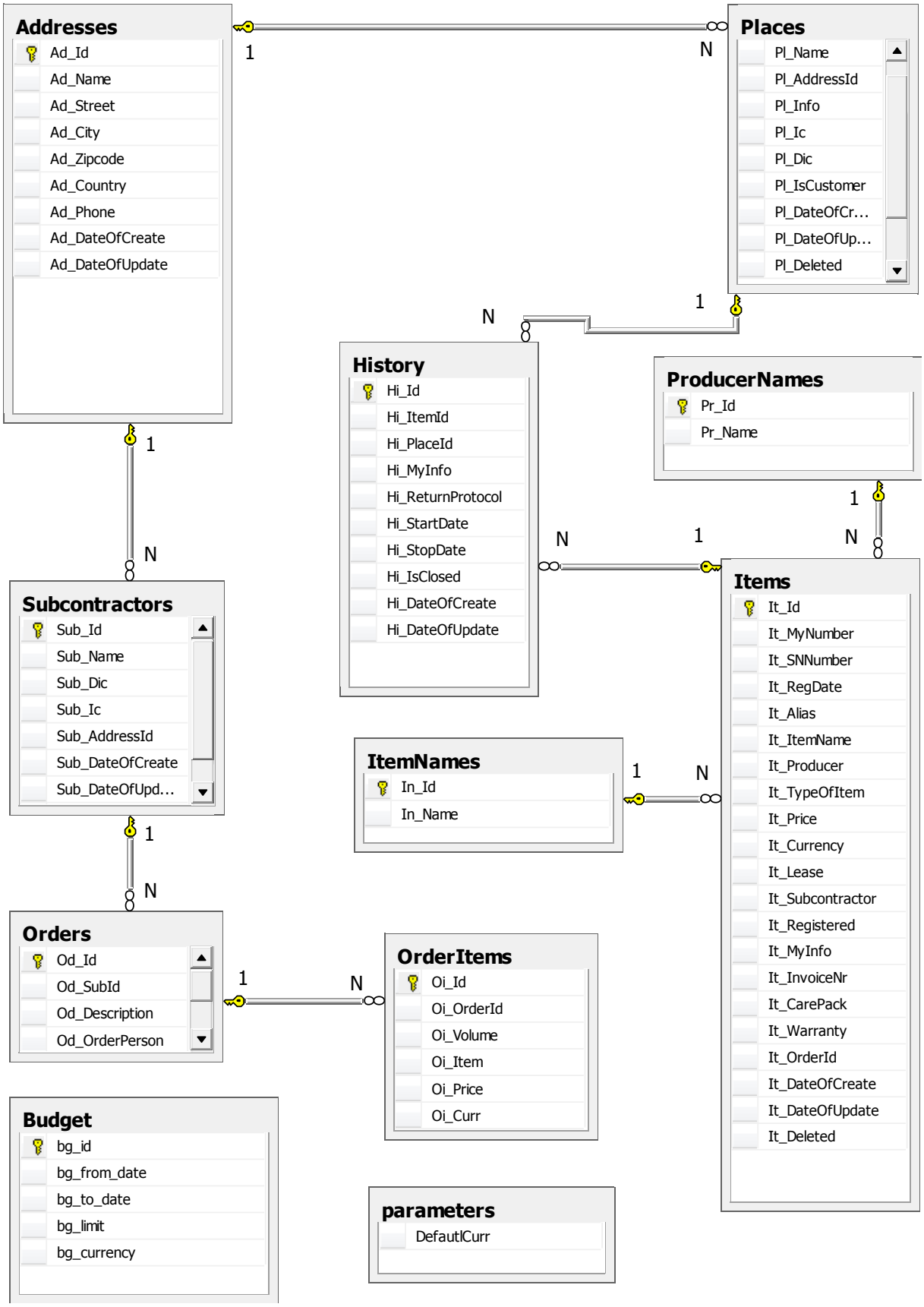
- majetek,
- dodavatele,
- umístění,
- rozpočet,
- objednávka,

Tyto položky můžeme dále rozložit.

Majetek bude mít vždy opakující „název zařízení“ a „výrobce“. U majetku bude nutné definovat ukazatel do tabulky dodavatelů. Pro název zařízení a výrobce vytvoříme další dvě tabulky.
Dodavatelé a **Umístění** budou mít položku adresa. Proto si pro adresu můžeme přidat tabulku. A zároveň z Dodavatele a Umístění definovat odkaz (cizí klíč) do tabulky adres. **Rozpočet** je jednoduchá tabulka a není nutné definovat spojení.

Další tabulka bude vytvořena pro **Objednávky**, ale objednávka může obsahovat více položek, proto si pro položky vytvoříme tabulku a nastavíme spojení.

Podle požadavků na aplikaci by mělo být možné zobrazovat historii, kde a po jakou dobu byla daná položka umístěna. A zároveň musíme získat aktuální polohu položky. Oba údaje můžeme vložit do jedné tabulky, kde určíme, zda položka byla uzavřena a převedena na novou pozici. Nebo je stále otevřená.



Obrázek 2. Era model

Tabulka Items

Obsahuje informace týkající se jednotlivého majetku.

Název sloupce	Typ	Popis
It_Id	Int	- Index tabulky a privátní klíč, při vložení se přičítá automaticky.
It_MyNumber	nvarchar(50)	- Interní číslo pro interní použití.
It_SNNumber	nvarchar(50)	- Sériové číslo položky.
It_RegDate	Datetime	- Datum registrace, datum evidence majetku.
It_Alias	nvarchar(50)	- Alias pro majetek, zatím nevyužito.
It_ItemName	Int	- Název zařízení, cizí klíč do tabulky ItemNames.
It_Producer	Int	- Výrobce, cizí klíč do tabulky ProducerNames.
It_TypeOfItem	nvarchar(50)	- Typ zařízení.
It_Price	float	- Pořizovací cena.
It_Currency	nvarchar(50)	- Měna pořizovací ceny.
It_Lease	bit	- Pokud je 1, tak majetek je pouze v pronájmu.
It_Subcontractor	int	- Cizí klíč do tabulky SubContractors.
It_Registered	bit	
It_MyInfo	nvarchar(255)	- Dodatečné informace týkající se majetku.
It_InvoiceNr	nvarchar(255)	- Číslo faktury.
It_CarePack	date	- Zatím nevyužito.
It_Warranty	date	- Záruka do.
It_OrderId	varchar(20)	- Číslo objednávky, nemá klíč z do tabulky Orders.
It_DateOfCreate	datetime	- Datum vytvoření.
It_DateOfUpdate	datetime	- Datum aktualizace.
It_Deleted	datetime	- Datum vymazání, nevyužito.

Tabulka Places

Tabulka obsahuje nezbytné informace týkající se umístění majetku.

Název sloupce	Typ	Popis
Pl_Id	int	- Index tabulky a privátní klíč, při vložení se přičítá číslo automaticky.
Pl_BelongTo	int	- Určuje nadřazenou pozici.
Pl_ShortName	nvarchar(255)	- Krátké jméno.
Pl_Name	nvarchar(255)	- Celé jméno.

Pl_AddressId	int	- Reference do tabulky adresses.
Pl_Info	nvarchar(255)	- Dodatečná informace.
Pl_Ic	nvarchar(20)	- IČ.
Pl_Dic	nvarchar(50)	- DIČ.
Pl_IsCustomer	bit	- Určuje, zda položka byla půjčena zákazníkovi.
Pl_DateOfCreate	datetime	- Datum vytvoření.
Pl_DateOfUpdate	datetime	- Datum aktualizace.
Pl_Deleted	datetime	- Datum smazání.

Tabulka Addresses

V tabulce je informace o adrese. Sloupec *Ad_Id* je současně i cizím klíčem pro tabulky Subcontractors a Places.

Název sloupce	**Typ**	**Popis**
Ad_Id	int	- Index tabulky a privátní klíč, při vložení se přičítá číslo automaticky.
Ad_Name	nvarchar(255)	- Jméno nebo firma.
Ad_Street	nvarchar(255)	- Ulice.
Ad_City	nvarchar(255)	- Město.
Ad_Zipcode	nvarchar(10)	- PSČ.
Ad_Country	nvarchar(50)	- Stát.
Ad_Phone	nvarchar(50)	- Telefon.
Ad_DateOfCreate	date	- Datum vytvoření záznamu.
Ad_DateOfUpdate	date	- Datum aktualizace záznamu, realizováno pomocí triggeru.

Tabulka Subcontractors

Obsahuje informace o subdodavatelích. Pole tabulky a jejich popis:

Název sloupce	**Typ**	**Popis**
Sub_Id	int	- Index tabulky a privátní klíč
Sub_Name	varchar(255)	- Jméno subdodavatele.
Sub_Dic	nchar(20)	- DIČ subdodavatele.
Sub_Ic	varchar(20)	- IČ subdodavatele.
Sub_AddressId	int	- Cizí klíč z tabulky Addresses.
Sub_DateOfCreate	datetime	- Datum vytvoření záznamu.
Sub_DateOfUpdate	datetime	- Datum změny záznamu, realizováno pomocí triggeru.

Sub_Deleted	datetime	- Datum smazání.

Tabulka Orders

Tabulka se záznamy objednávek.

Název sloupce	Typ	Popis
Od_Id	varchar(20)	- Index tabulky a privátní klíč.
Od_SubId	Int	- Cizí klíč tabulky Subcontractors.
Od_Description	nvarchar(MAX)	- Krátký popis objednávky.
Od_OrderPerson	varchar(255)	- Osoba vyplňující objednávku.
Od_Datum	Date	- Datum objednávky.

Tabulka OrderItems

Tabulka obsahuje seznam položek určených k objednání. Ke každé položce je zapsána přibližná cena a měna.

Název sloupce	Typ	Typ
Oi_Id	Int	- Index tabulky a privátní klíč.
Oi_OrderId	varchar(20)	- Cizí klíč z tabulky Orders.
Oi_Volume	Int	- Počet objednaných kusů.
Oi_Item	varchar(MAX)	- Název položky k objednání.
Oi_Price	Float	- Cena položky.
Oi_Curr	varchar(4)	- Měna u ceny položky.

Tabulka History

Tabulka ukládá veškerý pohyb majetku. V tabulce je uložena jak aktuální pozice, tak historie dané položky. Aktuální stav určuje současnou pozici majetku. Uzavřený stav znamená, že majetek byl umístěn na určité pozici. Stav aktuální od stavu uzavřeného bude rozlišen sloupcem *Hi_IsClosed*. Pokud bude *Hi_IsClosed* nastaven na 1, tak se jedná o uzavřenou historii. V opačném případě se jedná o aktuální pozici majetku.

Název sloupce	Typ	Typ
Hi_Id	int	- Index tabulky a privátní klíč. Toto číslo bude použito jako číslo převodu.
Hi_ItemId	int	- Reference do tabulky Items, zobrazuje informaci o položce.

Hi_PlaceId	int	- Reference do tabulky Places. Určuje umístění majetku.
Hi_MyInfo	nvarchar(500)	- Dodatečná informace přidaná během převodu majetku.
Hi_ReturnProtocol	bit	- Označuje, zda se vrátil podepsaný předávací protokol od zákazníka či subdodavatele. Pokud bude nastaven na 1, tak program bude uživatele upozorňovat na kontrolu předávacího protokolu.
Hi_StartDate	date	- Nastavuje datum počátku výpůjčky.
Hi_StopDate	date	- Nastavuje datum konce výpůjčky.
Hi_IsClosed	bit	- Určuje, zda je historie uzavřená.
Hi_DateOfCreate	datetime	- Datum vytvoření záznamu.
Hi_DateOfUpdate	datetime	- Datum poslední změny záznamu.

Tabulka Budget

Tabulka obsahuje seznam kontrolovaných časových úseků. Pro každý časový úsek je definovaný limit finančních prostředků k vyčerpání.

Název sloupce	Typ	Popis
bg_id	int	- Primární klíč.
bg_from_date	date	- Počáteční datum kontrolního období.
bg_to_date	date	- Konečné datum kontrolovaného období.
bg_limit	float	- Finanční prostředky pro kontrolované období.
bg_currency	nvarchar(5)	- Měna.

Při tvorbě databáze jsem současně vytvořil i několik *triggerů*. Tyto *triggery* například upravují sloupec, který zaznamenává poslední změnu záznamu.

Příklad *triggeru*, který vkládá aktualizované datum do sloupce *It_DateOfUpdate* v tabulce *Items*. Tento *trigger* se spouští hned po aktualizaci záznamu v tabulce *Items*.

```
USE [AssetAdmin]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER TRIGGER [dbo].[Items_updatedate]
    ON  [dbo].[Items]
AFTER UPDATE
AS
BEGIN
  SET NOCOUNT ON;
```

```
    UPDATE Items SET It_DateOfUpdate=GETDATE() WHERE It_Id= (SELECT
    DISTINCT It_Id FROM inserted)
    -- Insert statements for trigger here

END
```

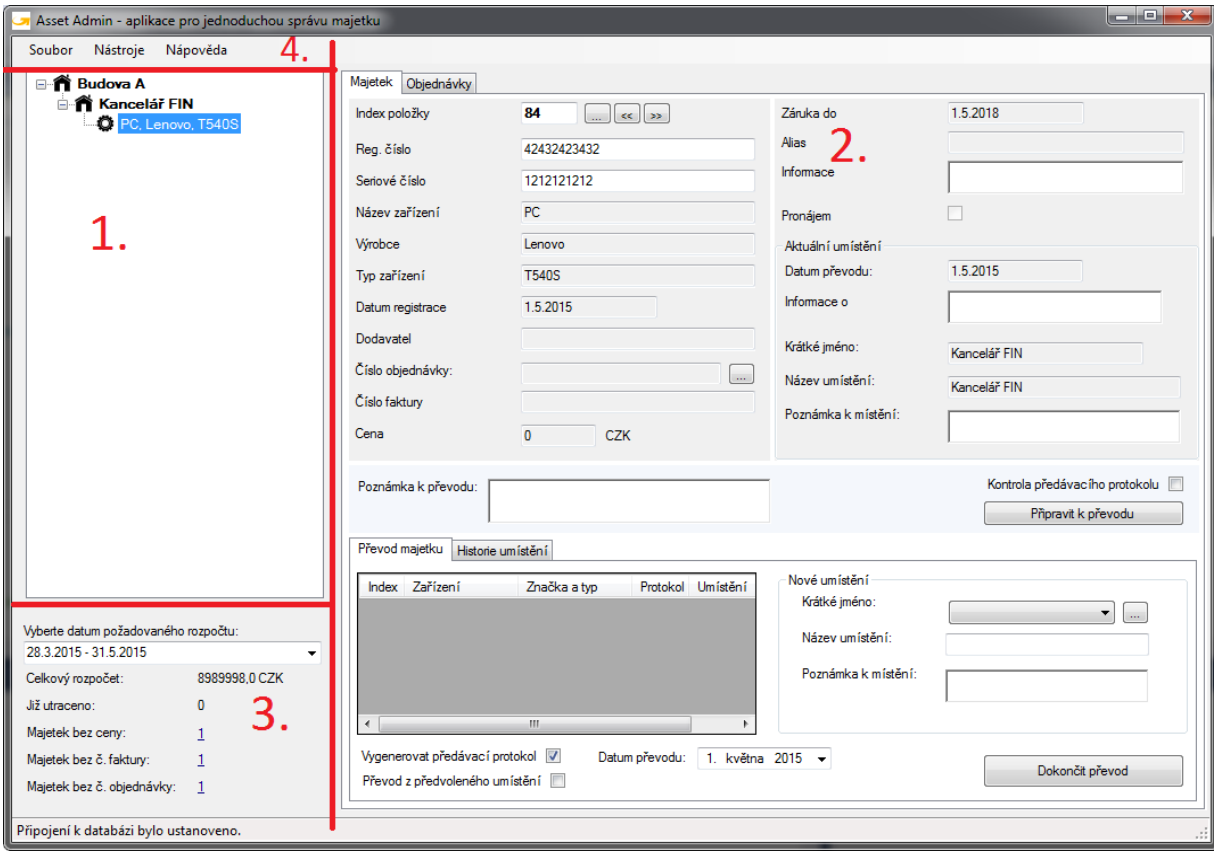
3.3.1 Uživatelské rozhraní

Pro tvorbu uživatelského rozhraní nám .NET nabízí několik knihoven. Mezi nejvíce používané patří *Windows.Forms* a *WPF*. V současné době začíná *WPF* vytlačovat *Windows Forms* a „dere se“ na první příčku. Tato knihovna je také technologicky dále, na rozdíl od *Windows Forms*. Nicméně, nejvíce tutoriálů a dokumentace je stále k *Windows Forms* a na začátku, kdy jsem začal tvořit tuto aplikaci, tak jsem o *WPF* neměl ponětí, proto jsem logicky zvolil *Windows Forms*.

Všechny grafické prvky použité v této práci bude z prostoru *System.Windows.Forms*, nebo budou odvozené z této třídy.

Základní hlavní okno

Základní okno je rozděleno na základní 4 částí.



Obrázek 3. Hlavní základní okno

1. Seznam všech umístění ve stromové struktuře. Kliknutím na určitou položku (místo) se rozbalí podseznam s podřadnými místy či seznam majetku, který je k danému místu evidován. Uživatel tak má přehled o počtu majetku na daném místě. Vnořování jednotlivých pozic do nadřazených položek bude zajištěno pomocí záznamu v tabulce Places, kde je definováno, která pozice je nadřazená.

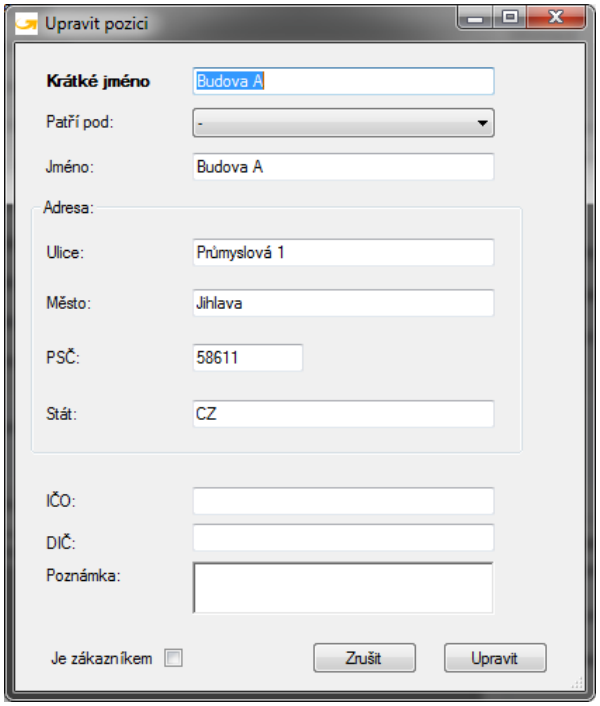
2. V této části je umístěn tabulátor, který má dvě stránky. První stránka slouží pro konkrétní informace o vybrané položce (index, sériové číslo, interní číslo, výrobce, název zařízení, typ majetku, datum registrace, záruku a aktuální pozici). Na první stránce také provádíme přesun majetku a zároveň můžeme zkontrolovat historii vybrané položky. Druhá stránka slouží pro tvorbu objednávek.

3. Tato část je oblastí rozpočtu. Uživateli je zobrazeno kolik již utratil a jaký je rozpočet.

4. Hlavní menu obsahuje odkazy na nastavení a úpravu dodavatelů, majetku a pozic.

Okno pro přidání a editaci pozice

Toto okno se zobrazí zavoláním pomocí příslušného tlačítka z menu.



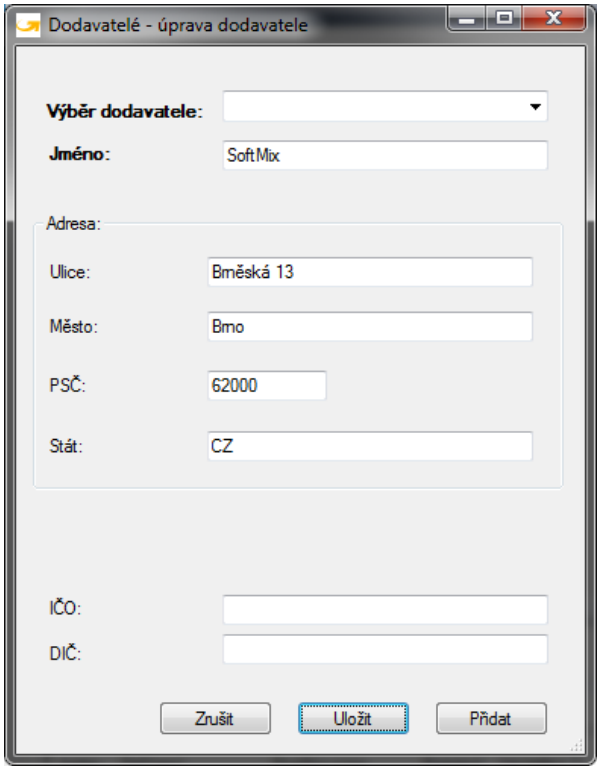
Obrázek 4. Okno pro přidání a editaci pozice

Požadované informace o umístění:

- Název umístění (krátké jméno).
- Název umístění (celé jméno). Např. jméno uživatele, firmy nebo oddělení.
- IČO, DIČ
- Poznámka pro další informace
- Políčko „Je zákazníkem“ – je využito v případě, že majetek je půjčen zákazníkovi.

Okno pro přidání a editaci subdodavatele

Toto okno se zobrazí zavoláním pomocí příslušného tlačítka z menu.



Obrázek 5. Okno editace dodavatelů

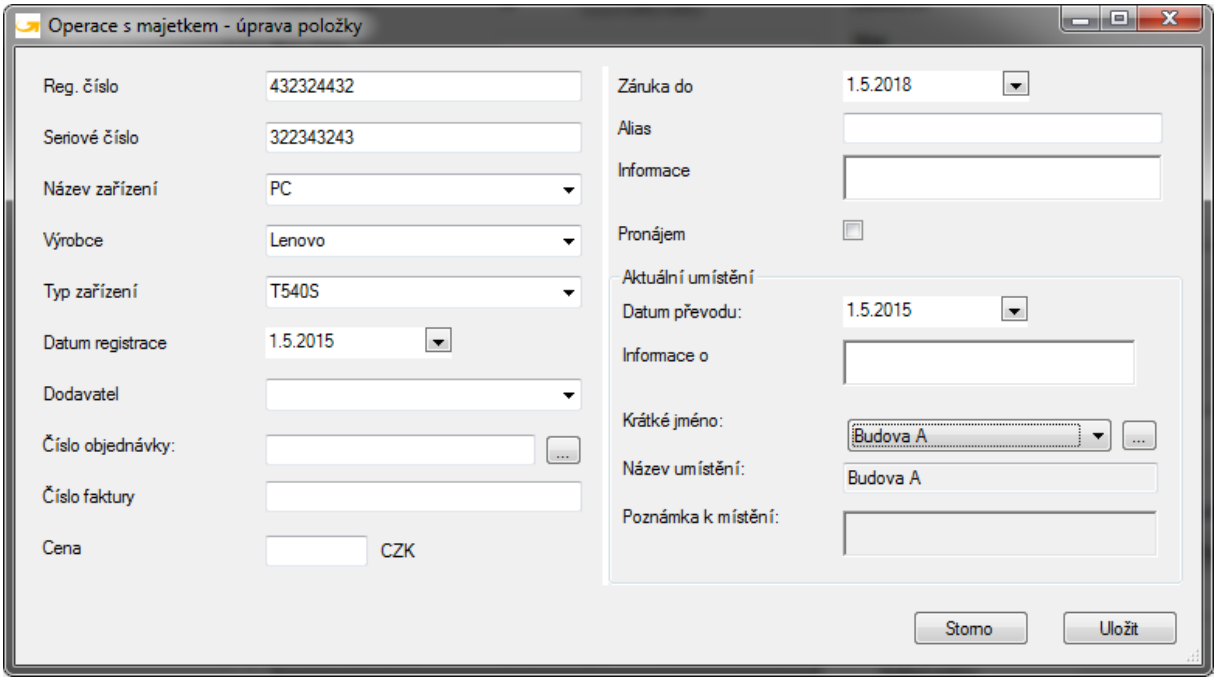
Požadované informace o subdodavateli:

- Jméno společnosti.
- Adresa.
- IČO, DIČ.

Seznam subdodavatelů bude využit při tisku objednávkového protokolu nebo při vkládání nového majetku.

Okno pro přidání a editaci majetku

Okno slouží pro přidávání a editaci majetku.



Obrázek 6. Okno pro vkládání a úpravu majetku

Požadované informace:

- Sériové číslo.
- Interní číslo – určeno pro potřebu firmy.
- Název zařízení – např. tiskárna.
- Výrobce.
- Typ zařízení – typové označení položky.
- Nákupní cena + měna.
- Číslo faktury.
- Číslo objednávky.
- Dodavatel – bude vybrán ze seznamu subdodavatelů.
- Cena.
- Pronájem – zaškrtávací políčko.
- Záruka – datum vypršení záruky.
- Dodatečné informace.
- Umístění majetku.

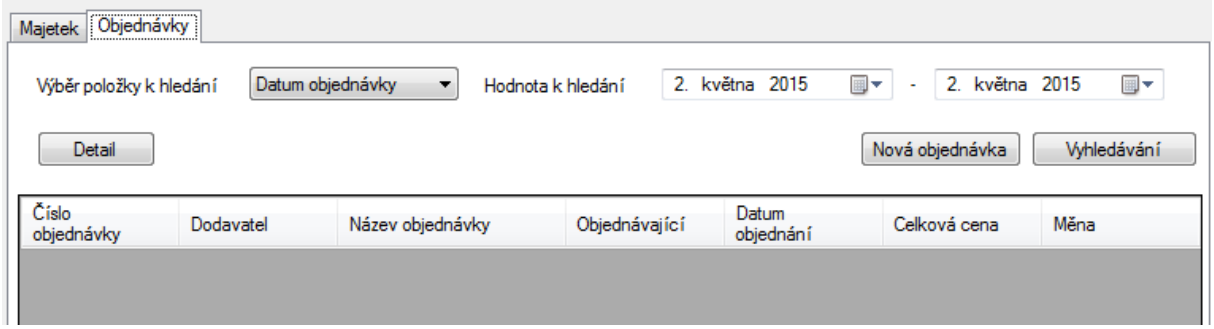
Toto okno zároveň bude obsahovat možnost vybrat inicializující umístění (typicky sklad).

V případě vytváření nové položky, po klinutí na tlačítko Uložit, bude položka uložena do databáze a program zobrazí index uložení a zprávu o úspěšném uložení.

Tlačítko Uložení nezavře aktivní okno a ani nevymaže vyplněná políčka, je to z důvodu, kdy uživatel eviduje více položek stejného druhu a mění se pouze sériové a inventární (interní) číslo. Tímto způsobem se zkracuje čas zadávání další položky.

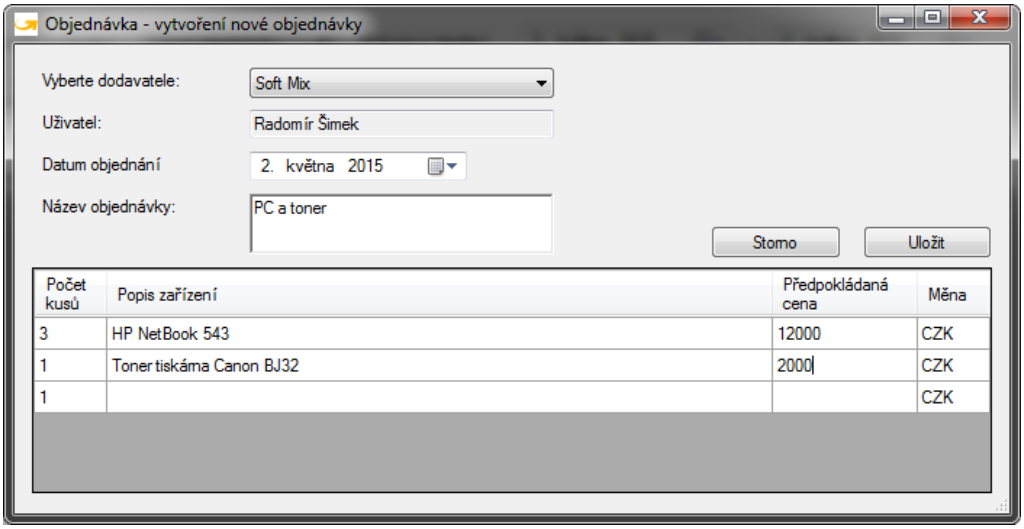
Správa a tvorba objednávky

Objednávku lze vytvořit v pravé části hlavního okna v tabulátoru Objednávky. V tomto okně je možné jak objednávky tvořit, tak i vyhledávat.



Obrázek 7. Správa objednávek

Při tvoření objednávky si uživatel může vybrat z předvolených subdodavatelů a zadávat jednotlivé položky a jejich předpokládané ceny.

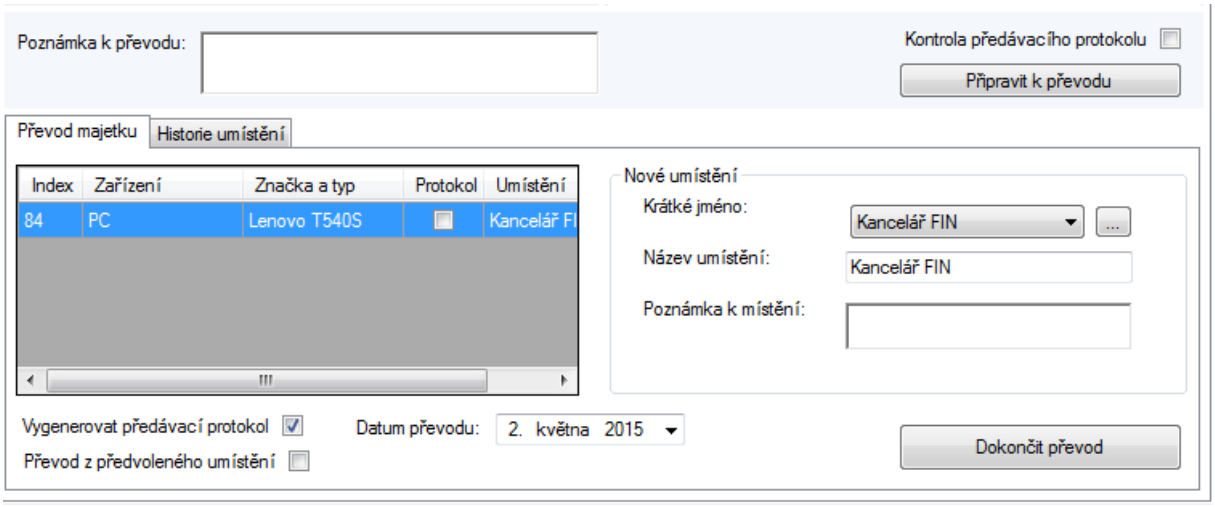


Obrázek 8. Tvorba jednotlivých objednávek

Dodavatel je povinen na každou fakturu vyplnit číslo objednávky, která byla realizována. Číslo objednávky slouží pro zpětnou identifikaci faktury a tvoří tak jedinečný pár.

Manipulace s majetkem

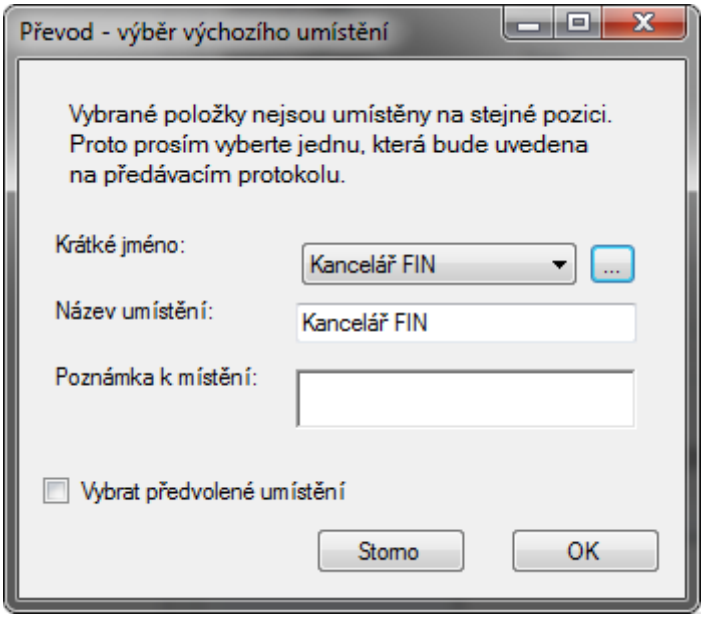
Převod majetku se uskutečňuje v pravé části hlavního okna, tabulátor Majetek. Uživatel si vybraný majetek připraví k převodu tlačítkem „Připravit k převodu", tím se vybraný majetek přidá do dočasného seznamu. Do seznamu lze přidat více položek a připravit je k převodu.



Obrázek 9. Převod majetku na nové umístnění

Vybraný majetek se přesune do nově vybrané pozice tlačítkem „Dokončit převod". Předávací protokol je vygenerován jen v případě, že je zaškrtnuto „Vygenerovat předávací protokol". Při přesunu více položek mohou nastat dvě situace:

- Všechen majetek je přesunut z jednoho umístění, v takovém případě je na předávacím protokolu původní místo, odkud je majetek převáděn.
- Jednotlivé položky jsou převáděny z různých míst. Program otevře okno s dotazem, které místo bude figurovat na předávacím protokolu jako původní.



Obrázek 10. Výběr aktuální pozice

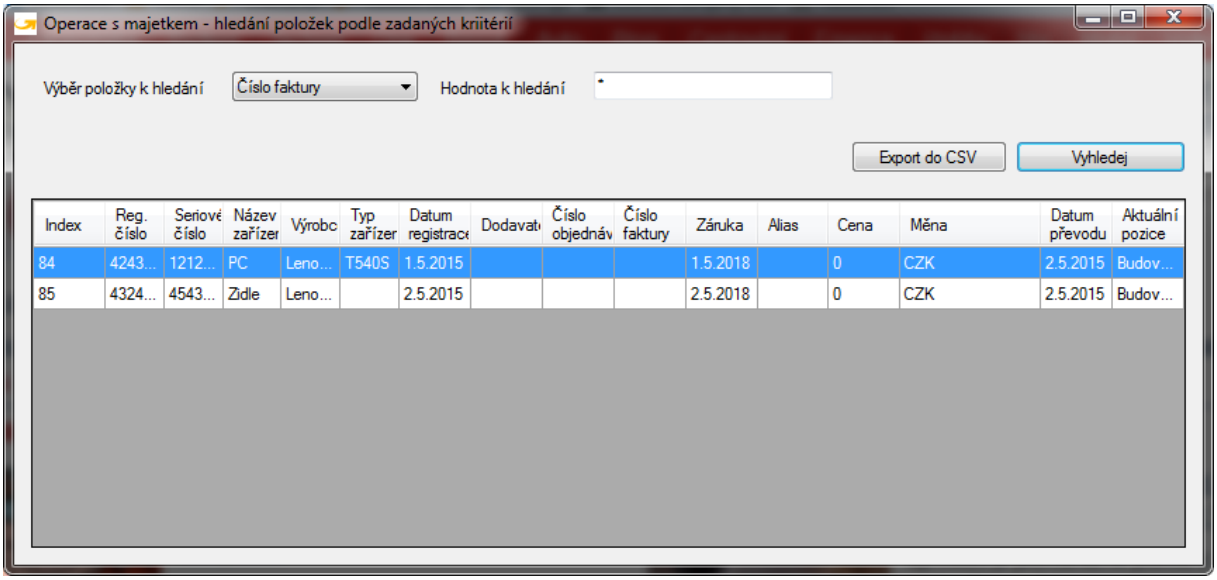
Po dokončení program přesune vybrané položky do nového umístění a aktualizuje stromovou strukturu v levé části hlavního okna.

Funkce vyhledávání majetku

Vyhledávat majetek je možné třemi způsoby:

1. Ve stromové struktuře pozic. Kliknutím na pozici, program načte všechny položky, které jsou na dané pozici umístěné.

2. Jednoduché vyhledávání v pravé části hlavního okna, tabulátor „Majetek". Uživatel může vyhledat položky pomocí sériového čísla, interního čísla a indexu. Po zadání jedné z těchto hodnot zkusí program prohledat databázi a zobrazit příslušnou položku a její aktuální zobrazení. V případě hledání podle sériového nebo interního číslo, může být použito i zástupného znaku „*".
3. Rozšířené vyhledávaní v samostatném okně. Okno je dostupné z hlavního menu nebo klávesovou zkratkou „Ctrl+F". Uživatel má možnost vyhledávat podle různých parametrů. V případě nalezení podle zadaných kritérií, program načte nalezené položky do seznamu. Seznam nalezených položek je možné vyexportovat do textového souboru *csv*.



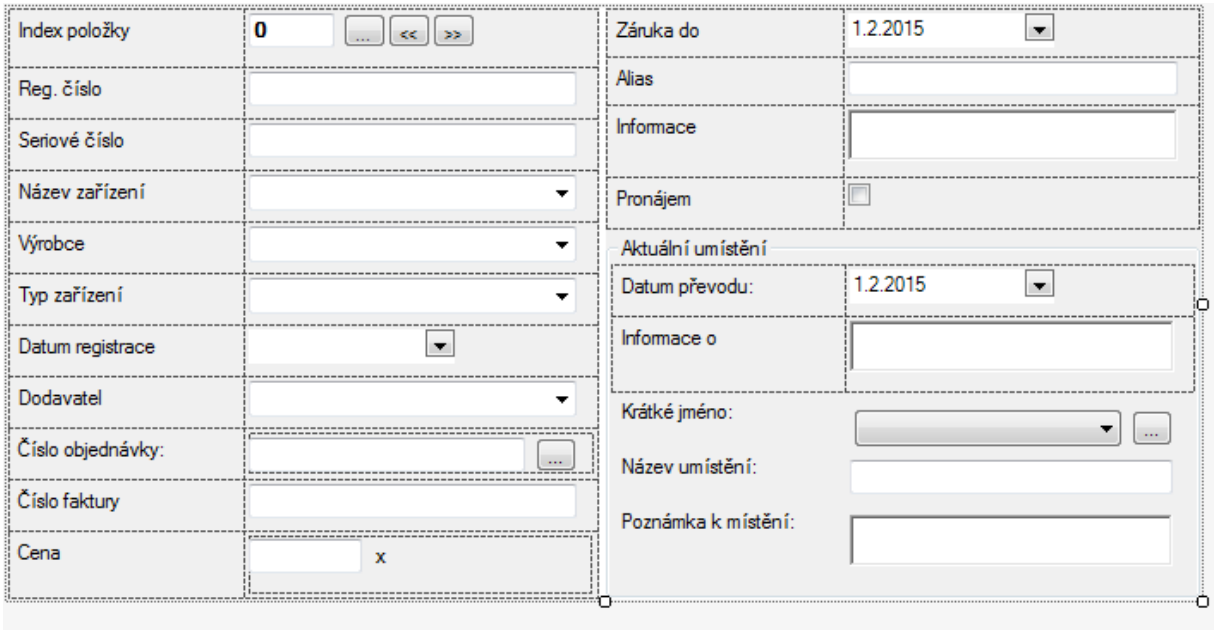
Obrázek 11. Rozšířené vyhledávání majetku

3.3.2 Návrh základních grafických prvků

Při tvorbě aplikace s rozsáhlým uživatelským prostředím, se programátor vždy potká se situací, kdy se jedny a ty samé formuláře opakují. Na jednu stranu slouží pouze pro zobrazení a uživatel nemůže položky ve formuláři měnit a na druhou stranu se používají pro úpravu a očekávají akci uživatele. V takovém případě nemá smysl tvořit dva formuláře, ale pouze jeden a nastavit mu určitou akci. V mé aplikaci je takových prvků využito několik. Windows *Forms* má pro tuto situaci třídu, která se nazývá *UserControl* a lze ji jednoduše přidat do projektu.

Ve vývojovém prostředí Visual Studio tuto třídu přidáte, tak že kliknete pravým tlačítkem na projekt v okně *Solution Explorer*, poté *Add* a *UserControl*. Visual studio vygeneruje novou třídu a uživatelské rozhraní, do kterého je možné přidávat další prvky. Příkladem v mém projetku může být *UserControl* nazvaný *FrmCtrItem* ve složce *Forms*. Tato třída se využívá jak v hlavním okně, pro zobrazení položky, tak i pro její úpravu.

Základní části ovládacího prvku FrmCtrItem



Obrázek 12. Návrh ovládacího prvku FrmCtrItem pro zobrazení majetku

Takto připravený grafický návrh lze vnořit do jiného okna. Programátor si tak může připravit jedno hlavní okno, složené z oken menších.

Jak jsem již psal třída *FrmCtrItem* je jak pro zobrazení, tak i pro úpravu položky. Musí tedy existovat vlastnost, která tento formulář přepíná do různých stavů.

Základním stavebním kamenem je vytvoření vlastního výčtového typu popisující akci *FormAction*. Tento výčtový typ se nachází v *NameSpace Enums* a je využit i v dalších formulářích. Jak je patrno, programátor může nastavit akci editace, přidání nebo pouhé zobrazení.

```
/// <summary>
/// Akce formuláře
/// </summary>
public enum FormAction
{
    edit,
    add,
    delete,
    preview,
}
```

Dalším krokem je tento výčtový typ definovat jako privátní proměnou ve třídě *FrmCtrItem.*

```
/// <summary>
/// Nastavuje akci komponenty - Add,Edit,Prewiev
/// </summary>
private Enums.FormAction action = Enums.FormAction.preview;
```

Privátním je proto, aby k němu nemohlo být přistupováno zvenčí. Zvenčí lze k této proměnné přistupovat přes veřejnou vlastnost *SetAction.*

```
[Browsable(true)]
public Enums.FormAction SetAction
{
    set
    {
        this.action = value;
        ReloadAllControl();
    }
}
```

Vlastnost *SetAction* nastavuje proměnou *this.action*. Po nastavení akce se zavolá metoda *ReloadAllControl()* a na zakladě hodnoty akce nastaví vlastnost všech prvků ve formuláři.

Možnosti akce:

- Preview, formulář slouží pouze pro prohlížení a všechny prvky jsou nastaveny na readonly.
- Add, slouží pro přidání nového majetku, hodnoty ve všech prvcích je možné upravit a zároveň je možné vybrat startovací pozici majetku.
- Edit, slouží pro úpravu vybraného majetku, hodnoty ve všech prvcích lze editovat, aktuální pozici majetku nelze změnit.

Jak bylo popsáno v požadavcích na aplikaci, tak by mělo jít vyhledávat přímo v tomto formuláři, proto jsou ve třídě *FrmCtrItem* definované tři textové pole, ze kterých je možné vyhledávat podle zadaného výrazu. Vyhledávání je možné z textového pole index, sériové číslo a registrační číslo. Uživatel například zadá sekvenci znaků k vyhledávání a poté stiskne klávesu enter jako signál pro start prohledání databáze. Problém je, ale v tom, že takto zavolaná událost vznikne a zpracuje se pouze uvnitř třídy, odkud byla stisknuta klávesa *Enter*. Událost musí být nějakým způsobem předána do jiné třídy, kde bude zpracována.

Nyní zkusím krok po kroku popsat zpracování události prvku sériového čísla, při stisknutí klávesy *Enter*.

Při stisknutí nějaké klávesy v textovém poli se vyvolá událost. Aby bylo možné tuto událost získat a zpracovat, je nutné si tuto událost takzvaně předplatit.

```
this.txSnNumber.KeyDown+=new
System.Windows.Forms.KeyEventHandler(this.txSnNumber_KeyDown);
```

Dále musí existovat metoda *this.txSnNumber_KeyDown*, která slouží jako obsluha události.

```
private void txSnNumber_KeyDown(object sender, KeyEventArgs e)
{
    if (this.action != Enums.FormAction.preview) return;
    if (e.KeyCode != Keys.Enter) { return;}
    EventDerivedClasses.ItemSearcher arg = new EventDerivedClasses.ItemSearcher();
    arg.ValueToSearch = this.txSnNumber.Text;
    arg.TypeOfRequest = Enums.SearchRequestType.serialnumber;
    arg.TypeOfSelection = Enums.SelectionType.exactThis;
    this.OnSearchEventStarted(arg);
}
```

V metodě kontroluji typ stisknuté klávesnice, pokud by stisknutá klávesnice nebyla *Enter*, tak opouštím metodu. V opačném případě inicializuji objekt *EventDerivedClasses.ItemSearcher*, který mi slouží pro vyhledávání položek.

Třída *ItemSearcher* v sobě nese informaci, co a jakým způsobem budu hledat. První vlastností je hodnota, kterou hledám, druhou vlastností je typ hodnoty k hledání (seriové číslo, index, interní číslo) a třetí, je způsob jakým hledám.

Třída *ItemSearcher* je odvozena od systémové třídy *EventArgs*. A primárně slouží jako argument při přenosu v události.

Na konci metody zavolám *this.OnSearchEventStarted(arg),* která zkontroluje, zda je událost předplacena a vyvolá ji. Ale předtím je nutné si událost definovat ve třídě *FrmCtrItem*

```
public event EventHandler<EventDerivedClasses.ItemSearcher> SearchEventStarted;
```

Ale bohužel tato definice nám nezajistí přenesení události do jiné třídy. To se udělá, tak že se opět nastaví předplatitel v hlavní třídě, odkud *FrmCtrItem* inicializujeme např:

```
private FrmCtrItem frmCtrItem1;

this.frmCtrItem1 = new AssetAdmin.FrmCtrItem();
```

A nyní událost předplatíme.

```
this.frmCtrItem1.SearchEventStarted += new
System.EventHandler<AssetAdmin.EventDerivedClasses.ItemSearcher>
(this.frmCtrItem1_SearchEventStarted);
```

Zároveň musíme definovat metodu *this.frmCtrItem1_SearchEventStarted*, ve které se bude událost obsluhovat. Tato metoda má parametry typu *object* a *ItemSearcher*. *ItemSearcher* již nese informace z formuláře a podle těchto informací již můžeme začít prohledávat v databázi.

```
private void frmCtrItem1_SearchEventStarted(object sender,
EventDerivedClasses.ItemSearcher e)
{ …                  }
```

Události jsou velmi šikovná vlastnost jak přenášet informace mezi více třídami.

Numerický TextBox

Jedno z dalších uživatelských rozhraní, které jsem vytvořil, byla třída *NumericUpAndDown*. Tato třída má opět vložené a seskupené ovládací prvky do jednoho celku. Nicméně jeden prvek je vytvořen nestandartní cestou. Jedná se prvek s názvem *NumericTextBox*. Jak již název napovídá, jedná se o *textbox*, který přijímá jen celá čísla. Numerický *textbox* je zděděná třída od klasického *textboxu* a využívá skutečnosti, že veškeré vlastnosti *textboxu* lze převzít anebo modifikovat. V mém případě stačilo pouze přepsat metodu *OnKeyPress*, která má v argumentu typ stisknuté klávesy a stačí jen kontrolovat, která klávesa byla stlačena. Pokud se jedná o číslici, tak povolit zpracovaní pomocí *e.Handled = true*. V opačném případě nic nedělat.

```
/// <summary>
/// Třída dědící vlastnosti od třídy TextBox.
/// TextBox umožňuje zadání pouze čísel a enter
/// </summary>
class NumericTextBox : TextBox
{
   /// <summary>
    /// Přetížená metoda OnKeyPress
    /// </summary>
    /// <param name="e"></param>
    protected override void OnKeyPress(KeyPressEventArgs e)
    {
        base.OnKeyPress(e);
        string keyInput = e.KeyChar.ToString();

        if (!(Char.IsDigit(e.KeyChar) == true || e.KeyChar == '\b' || e.KeyChar ==
        (char)Keys.Enter ))
        {
            e.Handled = true;
        }
    }
}
```

Přepisováním předdefinovaných grafických prvků, se programátorovi otevírají širší možnosti úpravy. Podobným způsobem jsem přepsal i prvek *DateTimePicker* nebo *ComboBox*.

ComboBox s vlastností ReadOnly

Při vytváření třídy *FrmCtrItem*, bylo nutné, aby při akci *Preview* (náhled), byly položky ve stavu *Readonly*. To znamená, že hodnoty zadané v textovém poli lze kopírovat, ale nelze je nijak měnit. Vlastnost *ReadOnly* je typická pro klasický *TextBox*, ale pro *ComboBox* a nebo pro prvek *DateTimePicker* taková vlastnost není. Použitelná vlastnost *ComboBox*, by mohla být *Disabled=true*, ale tato vlastnost prvek zamkne, text je špatně čitelný a nelze ho kopírovat. Aby všechny prvky vypadaly podobně, bylo nutné, si vlastnost *ReadOnly* vytvořit.

Abych odlišil tyto speciální a nové grafické prvky od současného projektu *AssetAdmin*, vytvořil jsem si nový projekt, který jsem nazval *ExtendedControls* a do něj přidal třídu *UserControl* s názvem *CtrComboBoxReadOnly*. Tento prvek je v podstatě *UserControl*, v kterém je vložena třída *ComboBoxWithReadOnly*.

Třída *ComboBoxWithReadOnly* je třída, která dědí vlastnosti od *ComboBox*. Dědění nám zajistí standardní chování *ComboBox*, ale můžem si přidat svoje vlastní metody a vlastnosti.

```
class ComboBoxWithReadOnly : ComboBox
{ … }
```

Tato třída v sobě ukrývá dva prvky, první je klasický *ComboBox*, který je v podstatě děděn z klasického *ComboBoxu* a druhý je klasický *TextBox*, který je v defaultním stavu neviditelný. V případě, že třída přejde do stavu *ReadOnly* je *True*, tak tento *TextBox* překryje defaultní *ComboBox* a načte si hodnotu z vlastnosti *Text*.

Proč jsem tedy tvořil *UserControl* a do něj vkládal prvek, který už má vlastnost *ReadOnly*?

Hlavním důvodem je, že pokud tento prvek vložíte do tabulkového panelu *TableLayoutPanel* nebo do panelu *FlowLayoutPanel*, tak tento panel nedokáže třídu *ComboBoxWithReadOnly* prezentovat jako jeden prvek, ale dva. Nejspíš je to tím, že třída má uvnitř dva grafické prvky. Bohužel se mi nepodařilo najít uspokojivou odpověď a musel jsem si vytvořit nový uživatelský prvek, aby oba prvky ve třídě *ComboBoxWithReadOnly* obalil.

Jak jsem se již zmínil, tak třída *ComboBoxWithReadOnly*, obsahuje vložený *textbox* a vlastnost *ReadOnly*. Vlastnost *ReadOnly* využívá metodu *ShowControl()*, která přepíná mezi *ComboBoxem* a *TextBoxem*. Tady je ukázka kódu této rozšířené třídy.

```
class ComboBoxWithReadOnly : ComboBox
{
    protected TextBox textbox;
    protected bool isReadOnly;
    protected bool visible = true;
    protected int maxLength = 255;
    public ComboBoxWithReadOnly()
    {
        this.textbox = new TextBox();
        this.ReadOnly = false;
    }
    public bool ReadOnly
    {
        get { return isReadOnly; }
        set
        {
            if (value != isReadOnly)
            {
                isReadOnly = value;

                ShowControl();
            }
     } }
```

Metoda *ShowControl* je velmi jednoduchá, pokud je *isReadOnly* nastaveno na True, tak se vlastnost Visible *TextBoxu* nastaví na True a vlastnost Visible *ComboBoxu* na False.

```
private void ShowControl()
{
    if (isReadOnly)
    {
        textbox.Visible = true && this.visible;
        base.Visible = false && this.visible;
        textbox.Text = this.Text;
    }
    else
    {
        textbox.Visible = false && this.visible;
        base.Visible = true && this.visible;
        this.Text = textbox.Text;
    }
}
```

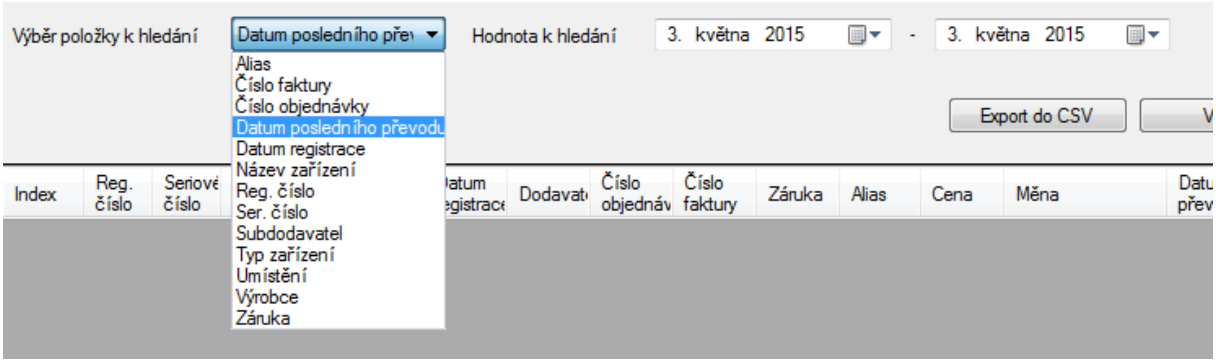
Velmi důležitá věc, kterou je třeba vyřešit je synchronizace těchto dvou prvků. Jedná se hlavně o pozicování prvku nebo nastavení *focusu*. Proto je nutné přepsat všechny zděděné metody, které mají na starost pozici z děděného prvku. Příklad jedné přepsané metody:

```
/// <summary>
/// Změna Indexu
/// </summary>
/// <param name="e"></param>
protected override void OnSelectedIndexChanged(EventArgs e)
{
   base.OnSelectedIndexChanged(e);
   if (this.SelectedItem == null)
       textbox.Clear();
   else
       textbox.Text = this.SelectedItem.ToString();    }
```

Bohužel toto řešení není příliš šťastné. Z mého pohledu to byl pouze kompromis kvůli rychlosti naprogramování. Lepším řešením by bylo přepsat vlastnost vykreslovací funkce *OnPaint* třídy *ComboBoxu.*

Ovládací prvek s vyhledávacími parametry

Posledním zajímavým grafickým prvkem, který bych rád zmínil je *FrmCtrSearchValueBox.* Tento ovládací prvek obsahuje *ComboBox* se seznamem položek k vyhledávání tzv. „Výběr položky k hledání“. Každá položka k hledání může být určitého typu. Podle zvoleného typu se otevře ovládací prvek tzv. „Hodnota k hledání“, kde uživatel může vložit vyhledávací řetězec. Např. pokud je typem datum, načte se prvek s časovým intervalem. V případě, že typ k hledání je text, tak se načte textové pole.

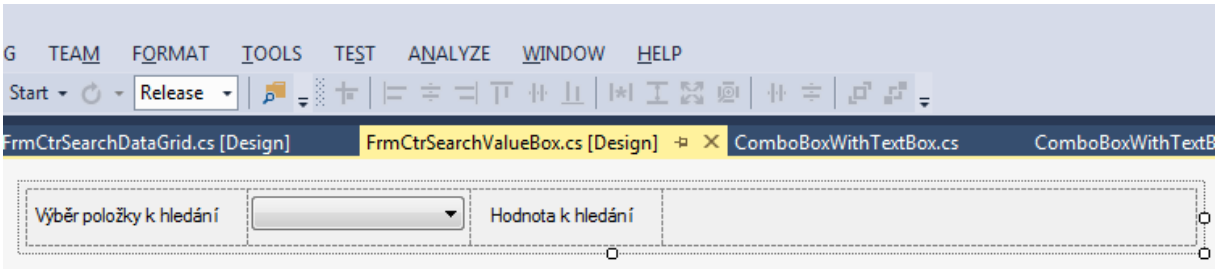


Obrázek 13. Ovládací prvek s vyhledávacími parametry

Stěžejní komponentou je *ComboBox* s objekty, které se budou vyhledávat, tyto objekty obsahují veškeré parametry. Objekty ve vyhledávacím *ComboBoxu* jsou typu *ComboSearchValuesStruct.* Tato třída *ComboSearchValueStruct* má vlastnosti, které určují název sloupce, který se bude vyhledávat v databázi a typ vyhledávaného objektu.

```
/// <summary>
/// Slouží k účelu vyhledávání položek
/// proměnná columnName je jméno sloupce ve kterém se bude vyhledávat
/// </summary>
public class ComboSearchValuesStruct
{
    private string name; /// Název objektu, bude zobrazovat uživateli
    private Type typeOfValue; /// Typ objektu - datum, text apod.
    private string columnName; /// Název sloupce v databázi

    /// <summary>
    /// Hodnota, podle které se bude plnit combobox
    /// </summary>
    private Enums.ItemValueAccordingToSearch toSerch;
    …
    …
}
```



Obrázek 14. Návrh ovládacího prvku s vyhledávacími parametry

Komponenta *FrmCtrSearchValueBox* má vlastnosti *WhatToFind* a *GetValue.* Tyto dvě hodnoty vrací uživatelem vybraný řetězec a typ.

```
/// <summary>
/// Typ hodnoty, kterou chceme hledat
/// </summary>
public ComboSearchValuesStruct WhatToFind
{
    get { return this.whatToFind; }
}

/// <summary>
/// Získá uživatelem zvolenou hodnotu
/// </summary>
public object GetValue
{
    get
    {
        if (this.whatToFind.TypeOfValue == typeof(ComboBox))
        {
            return ((ComboBox)this.valueBox).Text;
        }
            else if (this.whatToFind.TypeOfValue == typeof(TextBox))
        {
            return ((TextBox)this.valueBox).Text;
        }
            else if (this.whatToFind.TypeOfValue == typeof(FrmCtrDataTimeRange))
        {
            return ((FrmCtrDataTimeRange)this.valueBox).Value;
        }
            else return "";
        }
    }
```

3.4 Realizace a popis vnitřních funkcí programu

3.4.1 Připojení k databázi

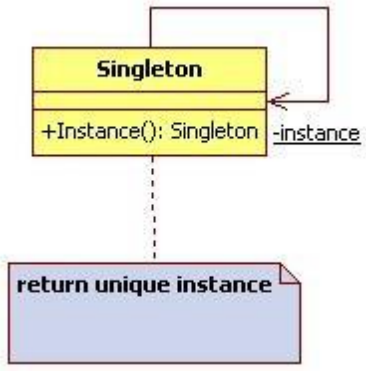
Jako převážná většina aplikaci, i tato pro svoji činnost využívá k ukládání dat databázi. Při tvorbě připojení se nabízejí dvě varianty. Program může být k databázi připojen neustále nebo využívá takzvaného odpojeného řešení. Každé z těchto řešení má své výhody i nevýhody. Odpojené řešení se používá, když aplikace neaktualizuje data často a v případě

nutnosti se připojí na malou chvíli a data aktualizuje. Odpojené řešení není vhodné, pokud do databáze zapisuje více uživatelů. Na druhou stranu, připojené řešení má neustále aktuální data, ale nevýhodou je zvýšená síťová režie při neustálém načítání dat. Pro moji aplikaci jsem vybral druhou variantu, kdy program bude připojen neustále.

Neustálé připojení k databázi nám bude zajišťovat třída *MsSqlConnection.* Třída obsahuje nejzákladnější metody pro získání a aktualizaci dat. Třída je napsaná podle návrhového vzoru *Singleton.* Což vlastně umožňuje vytvořit pouze jednu statickou instanci této třídy.

Singeton se vytvoří tak, že ve třídě definujeme privátní konstruktor. Tento privátní konstruktor je dostupný pouze z těla třídy. Ale to nám znemožní inicializace třídy zvenčí. Aby bylo možné třídu inicializovat, tak je třeba vytvořit statickou metodu nebo vlastnost třídy s veřejným přístupem, ve které inicializujeme celou třídu. Příklad klasického *Singletonu:*

```
class Singleton
{
    private static Singleton instance;
    private Singleton() { }
    public static Singleton Instance
    {
        get {
            if (instance == null) instance = new Singleton();
            return instance;
            }
     }
 }
```



Obrázek 15. Návrhový vzor Singleton [2]

Já jsem zvolil trochu jiný přístup a *Singleton* modifikoval. Pro inicializaci používám statickou metodu *LoadInstance* a pro přístup k instanci statickou metodu *GetInstance.* Důvodem tohoto přístupu je, že si chci řídit, kdy se bude instance inicializovat (při startu programu) a kdy budu přistupovat k již inicializované instanci. Zároveň při prvním spuštění programu program zkusí

ověřit, zda existuje připojení k databázi. Pokud ne, tak vyskočí okno s nastavením databáze a uživatele vyzve k nastavení přístupových údajů, kde opět zavolám *LoadInstance.* Pokud bych měl jen *GetInstance,* tak bych si musel ještě řídit první spuštění programu a inicializaci databáze.

```
class MsSqlConnection :  IDisposable
{
    protected System.Data.SqlClient.SqlConnection connection;
    protected string host;
    protected string database;
    protected string user;
    protected string pass;
    protected int port;

    private static MsSqlConnection instance;
    public static MsSqlConnection LoadInstance(string host, string database, string
   user, string pass, int port, bool force=false)
    {
        if (instance == null || force == true)
        {
            try
            {
                instance = new MsSqlConnection(host, database, user, pass, port);
                instance.Connect();
            }
            catch (Exception) { instance = null;  throw; }
        }

        return instance;
    }
    public static MsSqlConnection getInstance()
    {
        return instance;
    }
…
}
```

Třída *MsSqlConnection* má další metody pro získávání dat. Aby byl přístup do databáze bezpečný, snažím se tvořit sql dotaz přes tříd *SQLCommand* a tak ošetřit vstup uživatele.

Přiklad metody pro získání tabulky dat, kde je možné vložit SQL parametry přímo ve volání metody:

```
public System.Data.DataTable GetData(string sql, params
System.Data.SqlClient.SqlParameter[] parameters)
{
System.Data.SqlClient.SqlCommand cmd = this.connection.CreateCommand();
cmd.CommandText = sql;
if (parameters != null)
       cmd.Parameters.AddRange(parameters);
return this.GetData(cmd);
}
```

3.4.2 Získávání dat z databáze

Třída *MsSqlConnection* slouží jako komunikační prostředek s databází, to znamená, že nekontroluje, zda se dotaz týká majetku nebo objednávky. Třída se pouze snaží vkládat nebo získávat data na základě příkazů, které dostává od třídy *DBOperations.* Třída *DBOperations* je rozdělena na podtřídy a každá podtřída má na starost určitou logickou část.

Seznam podtříd třídy DBOperations:

- AddressOp – operace nad tabulkou adres Addresses.
- Budget – operace nad tabulkou Budget s finančním rozpočtem.
- HistoryOp – operace nad tabulkou historie History.
- ItemNamesOp – operace nad tabulkou ItemNames, kde jsou umístěny názvy majetku.
- ItemOp – operace s majetkem v tabulce Item.
- NotificationOp – má na starost sumarizovat data o finačním rozpočtu nebo počtu majetku, kde uživatel zapoměl zadat pořizovací cenu apod.
- Orders – operace nad tabulkou objednávek Orders a nad tabulkou OrderItems.
- Parameters – operace nad tabulkou s parametry.
- PlaceOp – operace nad tabulkou Places s pozicemi.
- ProducerNamesOp – operace na tabulkou ProducerNames s názvy výrobců.
- SubcontractorOp – operace nad tabulkou dodavatelů Subcontractors

Toto rozdělení bylo provedeno z důvodu lepší přehlednosti v kódu.

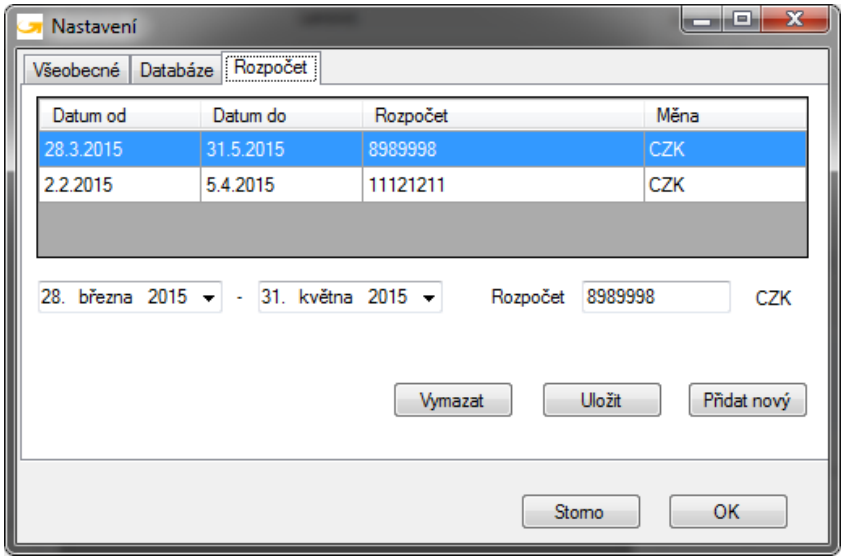
3.4.3 Nastavení programu

Pro ukládání konfiguračních informací jsem se rozhodl použít standartní třídu *System.Configuration* nabízenou v .NET. Důvodem je možnost přístupu k uloženým informacím v podstatě z jakékoliv třídy programu. Třída *Configuration* nabízí ukládání nastavení do textového souboru do složky uživatele. Pouze nastavení rozpočtu tvoří vyjímku, protože se načítá z databáze.

Okno nastavení se inicializuje přes hlavní menu, položka Možnosti. Po zadání správných parametrů, uživatel stiskne tlačítko OK a parametry se uloží do textového souboru ve složce uživatele. V mém případě je toto nastavení uloženo v cestě „c:\Users\uzivatel\AppData\Local\AssetAdmin\AssetAdmin.vshost.exe_3dus\1.0.0.0\“.

Pro nastavení jsem vytvořil dvě třídy, *CommonSettings a DBSetting*. K jednotlivým nastavením přistupujeme například přes tento řetězec:

```
AssetAdmin.DBSetting.Default.DBHost;
```

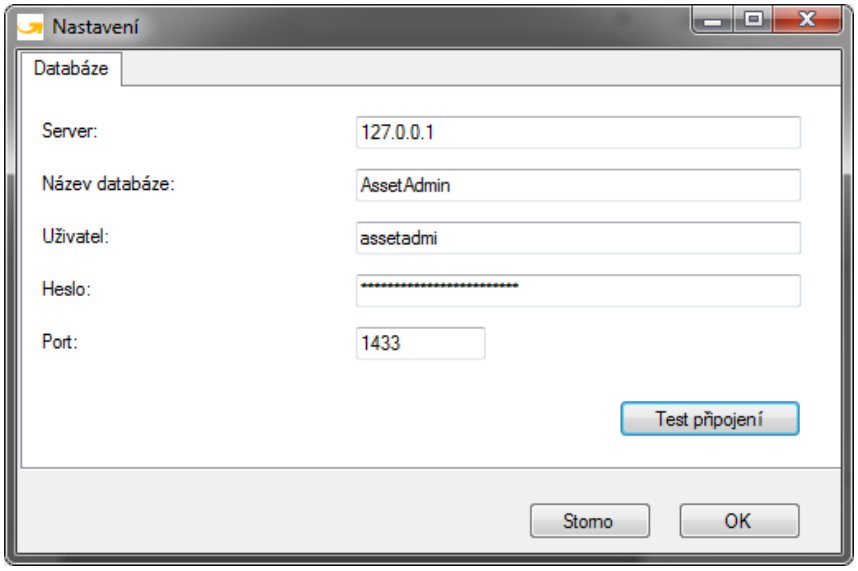


Obrázek 16. Možnosti programu

Rozhodnutí, že nastavení rozpočtu bude umístěno společně s ostatním nastavením ve stejném okně, mi malinko zkomplikovalo život. A to hlavně při prvním spuštění, kdy neexistuje připojení k databázi. Proto bylo nutné tento nedostatek ošetřit v kódu.

3.4.4 První spuštění

S přihlédnutím na fakt, že aplikace bude využívat SQL databázi, je třeba při prvním spuštění vyzvat uživatele k zadání dat pro přístup do databáze. Na základě těchto přihlašovacích údajů se program pokusí připojit k databázi. Tyto přihlašovací údaje se poté zapíší (zašifrované) do konfigurace programu pro další použití. Při prvním spuštění se program ptá ihned, protože nemá zadané žádné parametry k připojení databáze. V případě, že dojde ke změně databázového stroje nebo přihlašovacích údajů a programu se nedaří připojit k databázi, tak vyvolá výjimku a zobrazí výzvu k zadání přihlašovacích údajů. Další funkcí, je možnost si otestovat připojení přímo z aplikace. Aplikace má tu nevýhodu, že se dokáže připojit pouze přes IP adresu a ne přes tzv. *NamedInstance* MSSQL serveru. Zároveň aplikace nepodporuje vytvoření databázové sktruktury. Databázová struktura již musí na serveru existovat.



Obrázek 17. Okno s nastavením databáze při prvotním spuštění.

Pokud prvotní konfigurace proběhne v pořádku, program přejde do standartního spuštění. A uživatel může pracovat.

3.4.5 Standartní spuštění programu

V případě, že proběhlo první spuštění programu, tak program je již připojen do databáze a má nahrané veškeré parametry konfigurace. V opačném případě program načte konfiguraci z konfiguračního souboru, rozšifruje přihlašovací údaje k databázi a zkusí se připojit k databázovému serveru.

Inicializaci všech prvků a formulářů po připojení k databázi, má na starost metoda *LoadBaseComponent*. Postup načtení jednotlivých komponent:

1. Vytvoří adresář pro ukládání předávacích protokolů.
2. Načte naposledy přidaný majetek z databáze a zobrazí ve formuláři.
3. Načte stromovou strukturu.
4. Předplatí si události ze stromové struktury pro případ, že by došlo k vybrání nějakého prvku.
5. Nahraje pozice do ComboBoxu pro nové umístění.
6. Nahraje informace do notifikačního okna.

Tím program ukončí standartní spouštěcí proces a čeká na vstup od uživatele.

Načtení stromové struktury jednotlivých pozic

Načtení stromové struktury jednotlivých pozic není triviální proces. V tabulce *Places* je definován sloupec *Pl_BelongTo*, který nám ukazuje na nadřazené umístění. V případě, že je u tohoto sloupce vyplněna 0, tak tato pozice je v hierarchii pozic na nejvyšším stupni. Celý algoritmus zde:

```
public void ReloadWholeTree()
{
   /// Vyčistíme strom
   this.treePlaceTree.Nodes.Clear();
   /// Nahrajeme si seznam pozic do kolekce Dictonary
   Dictionary<int, Place> dlist = null;
   try { dlist = DBOperations.PlaceOp.GetDictonaryOfAllPlaces(); }
   catch { return; }
   /// Vytvoříme enumerator pro lepší procházení
   var numerator = dlist.GetEnumerator();
   TreeNode hTreeNode = null;
   Place hPlace = null;
   TreeNode parrent = null;
   /// Pokud vložíme pozici do stromu, tak jí vložíme do HashSetu,
   /// abychom mohli kontrolovat, jestli byla vložena
   HashSet<int> alreadyAdded = new HashSet<int>();
   ////
   /// Projdeme si všechny položky z databáze a vložíme do stromu
   while(numerator.MoveNext())
   {
          hPlace = numerator.Current.Value;
          /// pokud byla pozice vložena do menu tak ji přeskočíme
          ///
          if (alreadyAdded.Contains(hPlace.Id) == true) continue;
          /// Nejdříve vložíme všechny, které budou v hlavním nodu
          ///
          if (HelpClass.Common.RemoveNull<int>( hPlace.BelongTo,0) == 0)
          {
                hTreeNode = PlaceToTreeNode(hPlace);
                this.treePlaceTree.Nodes.Add(hTreeNode);
                alreadyAdded.Add(hPlace.Id);
                continue;
          }
         if (alreadyAdded.Contains(hPlace.BelongTo))
         {
                parrent = FindPlaceInTree(hPlace.BelongTo);
                parrent.Nodes.Add(PlaceToTreeNode(hPlace));
                alreadyAdded.Add(hPlace.Id);
                continue;
         }

    /// zkusime postavit nod rekurzivně a pozpátku a poté ho přidat do stromu
    hTreeNode = PlaceToTreeNode(hPlace);
    this.ReloadTreeFindParrentNodeRecur(ref dlist, ref alreadyAdded, hTreeNode);
    }

/// Nakonec nahrajeme zbloudilý majetek do stromu
this.AddRangeItemsToNode(null, null);
}
```

Algoritmus pro zobrazení stromové struktury, si tedy nahraje všechny pozice a seřadí je podle sloupce *Pl_BelongTo*, tam kde je uvedeno 0, tak se položky přidají do hlavního uzlu a pokračuje se v přidávání ostatních „poduzlů“ rekurzivní metodou.

Při řazení pozic jedné do druhé nám poslouží kolekce *Dictonary,* která z urychlí vyhledávání správných pozic.

ReloadTreeFindParrentNodeRecur vezme daný node a zkusí zjistit, zda jeho rodič je již ve stromě, pokud ano, tak ho najde a node do něj vloží.

V případě, že rodič není ve stromě, tak si ho zkusí najít ve slovníku *dList,* vytvoří z něj *TreeNode* a vloží do něj hledaný node. Metoda pak volá sama sebe rekurzivně, ale v parametru už je rodič.

```
private void ReloadTreeFindParrentNodeRecur(ref Dictionary<int,Place> dList, ref
HashSet<int> alreadyAdded, TreeNode node)
{
   Place plHelp = (Place)node.Tag;
   if (alreadyAdded.Contains(plHelp.Id) == true) return;
   TreeNode parrentNode = null;
   alreadyAdded.Add(plHelp.Id);
   /// Zkusíme si najít rodiče
   ///
   /// Pokud už jsou ve stromě, tak je to jednoduché
   if (alreadyAdded.Contains(plHelp.BelongTo))
   {
          parrentNode = FindPlaceInTree(plHelp.BelongTo);
          parrentNode.Nodes.Add(node);
          return;
    }
   Place parrentPlace = dList[plHelp.BelongTo];
   parrentNode = PlaceToTreeNode(parrentPlace);
   parrentNode.Nodes.Add(node);
   ReloadTreeFindParrentNodeRecur(ref dList, ref alreadyAdded, parrentNode);
}
```

3.4.6 Vytvoření a editace pozice

Pro vytvoření či editaci pozice slouží formulář, který je dostupný z hlavního menu programu. Tento formulář bude stejný jak pro úpravu pozice, tak i pro přidání nové pozice. Rozlišení mezi přidáním a úpravou bude určovat veřejná vlastnost formuláře, která se nastaví před zavoláním tohoto formuláře.

Při úpravě místa, stačí zavolat veřejnou metodu formuláře.

```
FrmPlace place = new FrmPlace();
place.SetEdit(id);
```

Po vyplnění všech nezbytných údajů, program zkusí data uložit data nebo aktualizovat tabulku *Places.* Nově přidaná či upravená pozice se samozřejmě musí objevit i v stromové struktuře pozic. Je tedy nutné podle položky *Pl_BelongTo* vyhledat nadřazený uzel, který volil uživatel, a tento nový záznam do něj vložit a zobrazit.

O aktualizaci stromu se stará metoda komponenty *FrmPlaceTree:*

```
UpdatePlaceInTreeNode(pozice)
```

Nebo.

```
InsertPlaceToTreeNode(pozice);
```

Objekt pro pozici obsahuje zároveň adresu umístění. Tato adresa se neukládá přímo do tabulky *Places*, ale do tabulky *Addresses.* V případě nutnosti je třeba, zapsat dané změny i do tabulky *Addresses.*

3.4.7 Vytvoření a úprava položky

Pro vytvoření či editaci majetku slouží formulář, který je dostupný z hlavního menu programu. O vzhledu formuláře bude rozhodovat veřejná vlastnost formuláře, která přepíná mezi funkcemi pro vložení nebo pro úpravu. Pokud budeme vkládat novou položku, tak bude existovat i možnost vybrat inicializační pozici pro tuto položku. Ale v případě, že budeme položku aktualizovat, tak již možnost změny pozice není povolena. Je to z toho důvodu, že přesun mezi pozicemi by se měl vždy zaznamenávat do historie pomocí standartní operace volané z hlavního okna. Vložení nebo úprava položky je realizováno v tabulce *Items.*

Okno pro vložení nebo úpravu má dvě akce, přidání anebo editace položky.

Přidání nové položky

Když přidáváme novou položku, tak je formulář nastaven tak, že po přidání položky se nezavře, ale v tom případě musíme nějakým způsobem dát vědět hlavnímu oknu, že byla položka přidána, aby ji hlavní okno mohlo přidat do stromu pozic a nastavit jako poslední. Proto jsem vytvořil událost, které se spustí při přidání nového majetku.

```
FrmItem i = new FrmItem();
// Nastavení akce přidej položku
i.SetActionAdd();
// Předplacení události a nastavení metody, kde bude událost zpracována
i.ItemWasAddEvent += ItemWasAddEvent;
i.ShowDialog();
```

Metoda reagující na událost:

```
/// <summary>
/// Reakce na přidanou položku
/// </summary>
/// <param name="item"></param>
void ItemWasAddEvent(Item item, int? placeId)
{
   // Aktualizace dat ve formuláři
   ReloadItemInForm(this, new
   EventDerivedClasses.ItemSearcher(Enums.SearchRequestType.index,
   Enums.SelectionType.last, item.Id));
   // Aktualizace položky v hlavním okně
   this.frmPlaceTree1.InsertItemToTreeNode(item, placeId);
}
```

Po přidání nové položky musíme také tuto informaci zadat do tabulky historie. Bude popsáno dále.

Editace položky

Při úpravě majetku, po zadání všech nutných vlastností a stisknutí tlačítka OK, se formulář zavře. Ale abych dodržel podobnou logiku jako u přidávání položky, tak je nutné také zaregistrovat událost.

```
FrmItem frm = new FrmItem();
// Předplacení události a metody, která událost obslouží
frm.ItemWasChangeEvent += frm_ItemWasChangeEvent;
// Nastavení akce editace
   frm.SetActioinEdit(this.frmCtrItem1.ItemObject, this.frmCtrItem1.PlaceObject,
   this.frmCtrItem1.TransferInfo);
frm.ShowDialog();
```

Po dokončení úpravy je nutné také položku aktualizovat ve stromu. Metoda zpracovávající událost z okna:

```
void frm_ItemWasChangeEvent(Item item)
{
   //Aktualizace dat ve formuláři
   ReloadItemInForm(this, new
   EventDerivedClasses.ItemSearcher(Enums.SearchRequestType.index,
   Enums.SelectionType.exactThis, item.Id));

    //Aktualizace dat ve stromě
   frmPlaceTree1.UpdateItemInTreeNode(item);
}
```

3.4.8 Přidání nebo úprava dodavatelů

Pro vytvoření či editaci subdodavatele slouží formulář, který bude dostupný z hlavního menu programu. O vzhledu formuláře bude rozhodovat veřejná vlastnost formuláře, která přepíná mezi funkcemi pro vložení nebo pro úpravu.

Informace o dodavatelích jsou ukládány do tabulky *Subcontractors.*

Objekt pro dodavatele obsahuje zároveň adresu dodavatele. Tato adresa se neukládá přímo do tabulky *Subcontractors,* ale do tabulky *Addresses.* V případě nutnosti zapíšeme změnu i do tabulky *Addresses.*

3.4.9 Manipulace s majetkem, ukládání do historie

Jak již bylo zmíněno výše, veškerá manipulace s majetkem probíhá v hlavním okně. Uživatel vybere danou položku, zvolí nové umístění a pomocí tlačítka převede majetek.

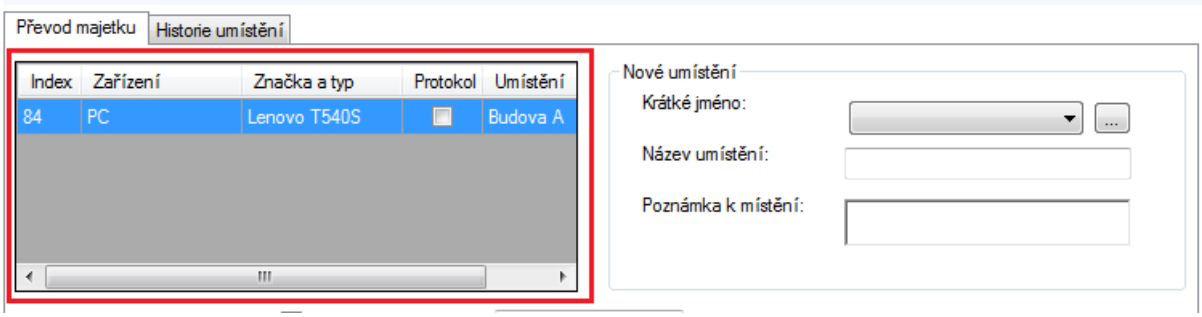
Program umí přesouvat jak jednu, tak i více položek na jedno místo. Během přesunu si bude uživatel moci zvolit:

- datum převodu
- funkci pro tisk předávacího protokolu
- přesun více položek
- zadat dodatečnou informaci k převodu

Při přesunu více položek můžeme narazit na problém. Vzhledem k tomu, že všechny přesouvané položky se nemusí nacházet na stejném místě, tak uživatel zvolí předávajícího podle rozmyslu a nabízejících se možností.

Každá položka, která je připravena k převodu tlačítkem „Připravit k převodu“, se uloží do dočasného seznamu.

Tento dočasný seznam je realizován ovládacím prvkem *DataGridView*. Objekt, který nese informaci o přesouvaném majetku, je vložen do objektu *Tag* v *DataGridViewRow.*



Obrázek 18. Dočasný seznam položek připravených k převodu.

Převod položky se ukládá to tabulky *History*. Důležité vlastnosti přesunu v této tabulce jsou ID majetku, ID pozice, datum převodu, datum ukončení a sloupec *Hi_IsClosed*.

Samotný převod položky probíhá v několika krocích:

1. Vyhledání aktuální pozice majetku v tabulce *History* podle *ID* majetku a sloupce *IsClosed = false*

2. Po vyhledání záznamu s aktuálním stavem, zapíšeme datum ukončení podle zadaného data převodu. A zároveň si nastavíme bit *IsClosed=true*.
   Tímto krokem položku vložíme do historie.

3. Přidáme nový řádek do tabulky *History* s následujícími vlastnostmi:
   - Datum převodu nastavíme na datum převodu zadané uživatelem.
   - Zapíšeme ID přesouvaného majetku.
   - Zapíšeme ID nové pozice.
   - Nastavíme bit *IsClosed=false*.

Pokud přesouváme více položek, tak body 1 až 3 se opakují v cyklu, jen měníme ID majetku.

Přesun majetku provádí metoda *MoveItemToNewPlace,* která k přesunu používá transakce. Právě transakce by nám měla zajistit, aby se majetek nedostal do nekonzistentního stavu. Transakce je v podstatě nedělitelná operace, která se provádí jako celek. Transakce se často používá v případech, vždy když k jedné položce v databázi přistupuje více zdrojů.

3.4.10 Zobrazení historie jedné položky

Tato funkce nám dá ucelený přehled o všech přesunech jedné položky. Pro zobrazení jsem použil ovládací prvek *DataGridView*. Tabulka s historii je umístěna v pravé časti hlavního okna v tabulátoru Majetek. Historie majetku se aktualizuje automaticky při změně položky.

Tabulka historie obsahuje sloupce:

- Časový interval.
- Umístění.
- Poznámku.

Funkce najde všechny položky v tabulce *History* podle *ID* majetku, seřadí je podle data převodu a zobrazí v *DataGridView*.

Dotaz pro získání historie jedné položky z databáze

```
"SELECT *  FROM History LEFT JOIN Places ON Hi_PlaceId = Pl_Id WHERE Hi_ItemId=" + iid
+ " ORDER BY Hi_StartDate"
```

3.4.11 Generování předávacího protokolu

Tato funkce je zavolána při převodu majetku. Parametry funkce jsou předávající, příjemce, seznam s položkami a datum převodu. Příjemce a předávající jsou v hlavičce protokolu. Na konci stránky jsou dvě políčka pro podpis předávajícího i přebírajícího.

Jednotlivý předávaný majetek je tisknut v seznamu pod sebou a je reprezentován:

- Registračním číslem
- Sériovým číslem
- Indexem položky
- Názvem a typem zařízení
- Dodatečnou informací

Pro generování předávacího protokolu ve formátu PDF jsem použil volně dostupnou knihovnu *PdfSharp.* Aby bylo možné tuto knihovnu použít, je nutné ji po stažení přidat do projektu jako referenci.

PdfSharp je velmi užitečná knihovna, která uspokojila téměř všechny mé požadavky. Jedinou funkci, kterou jsem v ní nenašel, bylo automatické odřádkování na pravém okraji stránky nebo na pravé straně definovaného rámce. Rozhodl jsem tento problém obejít a napsat svoji vlastní metodu *WriteText,* která je umístěná ve třídě *TransferProtocol.* Metoda *WriteText* odřádkuje pouze, pokud nalezne prázdný znak. Protokol se po vygenerování uloží na disk, uživatel může určité protokoly jednoduše dohledat a vytisknout ještě jednou.

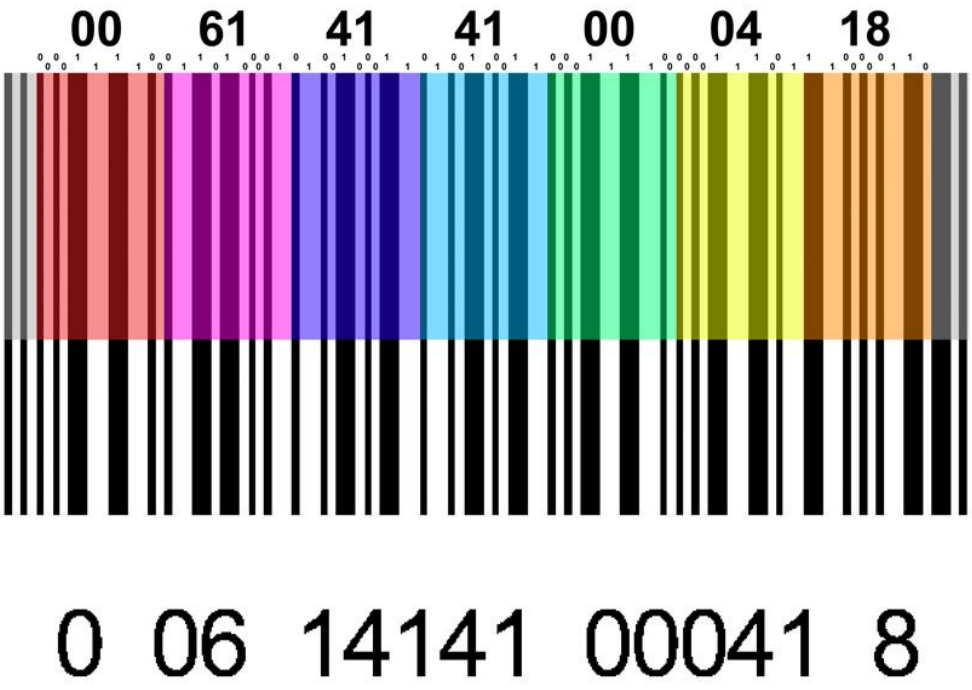
Aby bylo vyhledávání majetku efektivnější, rozhodl jsem se do protokolu přidat sériové a registrační číslo ve formě čárového kódu. Tento kód může být sejmut ruční čtečkou čárových kódů přímo do aplikace a majetek je okamžitě vyhledán.

O generování čárového kód se stará metoda *public override Image DrawIt* umístěná ve třídě *Barcode2of5*. Jak název třídy napovídá, třída generuje typ čárového kódu 2of5 interleaved [6]. Metoda byla vytvořena podle dokumentace uvedené na webových stránkách. Nevýhodou standardu *2of5interleaved* je, že lze do čárového kódu převést pouze číslice a počet číslic může být pouze sudé číslo.

Digit	Bar or space width					Mnemonic (using weights)
0	n	n	W	W	n	4+7=11, replaced by 0
1	W	n	n	n	W	1+0=1
2	n	W	n	n	W	2+0=2
3	W	W	n	n	n	1+2=3
4	n	n	W	n	W	4+0=4
5	W	n	W	n	n	1+4=5
6	n	W	W	n	n	2+4=6
7	n	n	n	W	W	7+0=7
8	W	n	n	W	n	1+7=8
9	n	W	n	W	n	2+7=9
Weight	**1**	**2**	**4**	**7**	**0**	

Obrázek 19. Tabulka převodu čísel na čárový kód

Při převodu čísel se kódují čísla vždy po dvou. Podle obrázku 19. a 20. je například číslo 18 kódováno na číslo 1 pomocí černých pruhů na *WnnnW* a číslo 8 na *WnnWn* pomocí prázdných pruhů. Tento čárový kód musí být uvozen speciální sekvencí znaků na začátku *nnnn* a *Wnn* na konci. V případě, že uživatel zadá číslo liché, algoritmus přidá na začátek číslo 0.



Obrázek 20. Příklad čárového kódu, kde různé barvy označují čísla ke kódování.

Ukázka kódu pro tvoření čárového kódu:

```
public override Image DrawIt(int width, int height)
{
        Image picture = new Bitmap(width, height);
        Graphics gfx = Graphics.FromImage(picture);
        Single BWidth = width;
        Single BHeight = height;
        /// Sekvence kódování
          string[] NValues = new String[] { "11331", "31113", "13113", "33111",
          "11313", "31311", "13311", "11133", "31131", "13131" };
        int GWRatio;
        Single BTn;
        String OddRep, EvenRep;
        Single ActPos = 0;
        Single ActLine;
        if (base.codeText.Length % 2 > 0)
        {
            base.codeText = "0" + base.codeText;
        }
        GWRatio = base.codeText.Length * 9 + 4 + 5;
        BTn = BWidth / GWRatio;
        gfx.FillRectangle(Brushes.White, 0, 0, BWidth, BHeight);
        gfx.FillRectangle(Brushes.Black, ActPos, 0, BTn, BHeight);
        ActPos = ActPos + BTn;
        ActPos = ActPos + BTn;
        gfx.FillRectangle(Brushes.Black, ActPos, 0, BTn, BHeight);
        ActPos = ActPos + BTn;
        ActPos = ActPos + BTn;
        for (int i = 1; i <= base.codeText.Length; i++)
        {
            if (i % 2 == 0) continue;
            OddRep = NValues[Convert.ToInt32(base.codeText.Substring(i - 1, 1))];
            EvenRep = NValues[Convert.ToInt32(base.codeText.Substring(i, 1))];
```

```
            for (int b = 1; b <= 5; b++)
            {
                ActLine = BTn * Convert.ToSingle(OddRep.Substring(b - 1, 1));
                gfx.FillRectangle(Brushes.Black, ActPos, 0, ActLine, BHeight);
                ActPos = Convert.ToSingle(ActPos + ActLine);
                ActLine = BTn * Convert.ToInt32(EvenRep.Substring(b - 1, 1));
                ActPos = Convert.ToSingle(ActPos + ActLine);
            }
            }
        gfx.FillRectangle(Brushes.Black, ActPos, 0, BTn * 3, BHeight);
        ActPos = Convert.ToSingle(ActPos + 3 * BTn);
        ActPos = ActPos + BTn;
        gfx.FillRectangle(Brushes.Black, ActPos, 0, BTn * 1, BHeight);
        return picture;
}
```

4 Závěr

Aplikace byla vytvořena v důrazu na jednoduchost a intuitivnost uživatelského ovládání. Byla důkladně otestována, ale zatím ještě nebyla nasazena do produktivního prostředí. Během testování jsem nepřišel na žádné chyby, program realizuje požadované operace správně. Nyní je naplánováno ostré nasazení ve firmě. Po nasazení do produkčního prostředí plánuji vytvořit v programu koš, pro staré a vyřazené položky a možnost smazání pozice. Zároveň bych rád přidal modul pro sledování softwarových licencí a ukládání předávacích protokolů do databáze.

Přestože to byl můj první větší projekt, tak jsem se snažil co nejvíce používat pokročilé techniky a oddělit kód uživatelského prostředí od databázové logiky. Během této práce jsem si osvojil programovací techniky jako je dědičnost mezi třídami, genericitu nebo události.

Dědičnost programátor využije v případě, když chce rozšířit základní třídy .NET o svoje vlastní metody a vlastnosti. Důvod je ten, že základní formulářové komponenty nemusí splňovat speciální požadavky programátora. Další zajímavou technikou jsou události a delegáti. Události jsou nejčastěji spojovány s grafickými prvky, ale lze je použít i pro přenos jakékoliv zprávy mezi třídami. Já jsem je například využil při přenosu informací z vlastních komponent při vyhledávání a přidávání majetku. Genericita je technika, která dovoluje vytvořit jednu metodu či třídu, která umí zpracovat různé typy proměnných tzv. generický typ. Můžeme si to představit tak, že se generický typ ve třídě změní např. na string ve chvíli, když vytvoříme její instanci. Jedná se tedy o možnost třídy nějakým způsobem parametrizovat. Dalším zajímavou částí kódu bylo vytváření vlastních výstupů do souboru *pdf* pomocí třídy *PDFSharp*. Třída je dostupná zdarma ke stažení a umožňuje tvořit různorodé typy pdf dokumentů. *PDFSharp* zvládá pokročilou práci s textem, obrázky a celkovou manipulaci s dokumenty.

Během psaní této aplikace se u nás ve firmě změnil požadavek na změnu tvorby procesu objednávek. Ty se nyní budou tvořit pomocí software pro oběh dokumentů, proto jsem v této aplikaci již nepřikládal takový důraz na tvorbu objednávek. Z tohoto důvodu bude dalším nutným krokem napojení aplikace na databázi našeho nového software pro oběh dokumentů, který je postavený na MS SharePoint.

Pokud bych se mohl vrátit na začátek a začal psát program znovu, určitě některé věci změnil. V prvé řadě bych si vytvořil lepší adresářovou strukturu a rozdělil kód do více *Namespaces*. Při dědění od tříd .NET, bych si v nově vytvořené třídě pojmenovával vlastnosti a metody

speciálním způsobem, například vložením slova *My* před každou vlastní metodu. Tyto postupy by kód zpřehlednily a tím i zlepšily orientaci při hledání.

Jak jsem již napsal na začátku, WinForms jsou již zastaralé a tak moje volba pro grafické rozhraní, by byla použít WPF.

Velmi zajímavou vlastností, kterou .NET disponuje jsou tzv. rozhraní - *Interface*. Rozhraní bych využil hlavně při připojení k databázi nebo pro operace tisku předávacího protokolu. Třída pro operace s databází by implementovala rozhraní. A toto rozhraní by bylo inicializováno místo databázové třídy. V případě, že bych se později rozhodl změnit připojení k databázi, stačilo by pouze napsat novou databázovou třídu, která by implementovalo rozhraní.

Ke konci práce jsem začal využívat i dotazovací jazyk *LinQ* [7]. *LinQ* je jazyk velmi podobný dotazům *SQL* a slouží k prohledávaní dat v datových kolekcích. Co dříve bylo nutné psát na několik řádků, *LinQ* dokáže pouze s jedním dotazem. V dalších projektech již budu s touto technologií plně počítat.

I když tato práce není dokonalá, tak v osobním rozvoji mi velmi pomohla a posunula zase o něco výše.

Seznam použité literatury

[1] MSDN: Visual C#. [online]. 2015. Dostupné z: https://msdn.microsoft.com/cs-cz/library/kx37x362.aspx

[2] Programujte.com: Seriál návrhových vzoru. [online]. 2003–2015. Dostupné z: http://programujte.com/clanek/2012032900-serial-navrhovych-vzoru-1-dil/

[3] Virius, Miroslav. C# Hotová řešení, Computer Press, Brno, 2006, ISBN 80-251-1084-2

[4] Mareš, Amadeo, 1001 tipů a triků pro C#, 1.vydání, Brno: Computer Press, 2008, 360, 978-80-251-2125-2.

[5] Trey Nash, C# 2010, Computer Press, Brno, 2010, ISBN: 978-80-251-3034-6

[6] Wikipedia, Interleaved 2 of 5. [online]. 2015 Dostupné z: http://en.wikipedia.org/wiki/Interleaved_2_of_5

[7] MSDN: LinQ. [online]. 2015. Dostupné z: https://msdn.microsoft.com/cs-cz/library/bb397933.aspx

Seznam obrázků

Přílohy

1 Přiložené CD

Na přiloženém CD v adresáři AssetAdmin jsou zdrojové kódy programu. V adresáři DB jsou skripty pro vytvoření základní databázové struktury.