Ondřej Klein
Využití DLL knihoven pro rozšíření funkcí aplikace

Využití DLL knihoven pro rozšíření funkcí aplikace
Using of DLL libraries for functionality extension

Bakalářská práce

Ondřej Klein

Vedoucí bakalářské práce: Ing. Václav Novák, Csc.

Konzultant bakalářské práce: Bc. Pavel Vacikar

Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra: Katedra informatiky

2008

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě pedagogickou fakultou elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách.

V Českých Budějovicích dne

Anotace

Bakalářská práce se zabývá problematikou pluginů ve formě DLL knihoven. Řeší problémy vznikající při tvorbě dynamicky linkovaných knihoven i problematiku navržení programu, který tyto knihovny využívá. Práce vznikla na podnět mnoha začínajících programátorů a právě pro ně je tato publikace koncipována.

Abstract

The subject of this work is troubleshooting with DLL libraries. It treat of a problems about creating DLL libraries and troubleshooting with making application for usage of these libraries. This work has been created becouse of beginning programmers.

Poděkování

Rád bych poděkoval Vedoucím své práce za pomoc při vytváření této publikace. Chtěl bych také poděkovat za názory a pomoc od ostatních programátorů z diskusních fór i z mého okolí.

Obsah

1 Úvod

Každý, kdo častěji používá počítač se jistě setkal s pojmem zásuvný modul nebo plugin. Ať už jej uživatel zaregistroval ve varování, které se objevilo při návštěvě webové stránky a upozorňovalo o chybějící části programu internetového prohlížeče, nebo při stahování updatů, kterými může přidávat nové funkce do svých oblíbených programů, jako jsou například instant messengery Icq či Miranda.

S tímto pojmem se tedy můžeme setkat poměrně často. To je způsobeno tím, že dnes je prakticky každá významnější aplikace navržena modulárně - tedy využívající moduly (pluginy), které dokáží tuto aplikaci řídit, ovlivňovat její chod nebo přidávat nové funkce a proto se s tímto pojmem musíme dříve či později setkat.

2 Základní pojmy

2.1 Seznámení

Pluginy – z úvodu je zřejmé, že ve světě programování se jedná o zcela zásadní záležitost. Jak jsem se však mohl osobně přesvědčit, sehnat publikaci se souhrnnými informacemi a ukázkami je velmi obtížné a když už nějakou naleznete, je psána velmi odborně, takže pokud takovouto publikaci otevře začínající programátor nebo samouk velmi záhy se začne ztrácet v záplavě odborných termínů a zkratek.

Cílem této práce je tedy pomoci začínajícím programátorům, kteří nemají žádné nebo ne příliš velké zkušenosti s vytvářením modulárních aplikací nebo samotných zásuvných modulů-pluginů.

Abychom mohli úspěšně pochopit problematiku tohoto tématu, je potřeba se seznámit s několika pojmy, které nás budou celou touto publikací provázet, a je tedy důležité znát jejich význam.

2.2 Samotné pojmy

- **DLL**

  Z anglického Dynamic Link Library – dynamicky linkovaná knihovna

- **Plugin**

  Z anglického To plug-in zasunout do (zásuvný modul)

- **Funkce**

  Je část programu, kterou je možné opakovaně volat z různých míst zdrojového kódu. Může obsahovat parametry(údaje) které jí jsou při zavolání předány, nebo návratové hodnoty které naopak vrací.

- **Implementace**

  je uvedení teoretické myšlenky do praktického příkladu mající za účel další použití.

- **Inicializace**

  dané proměnné přiřadíme počáteční hodnotu

- **Instance**

  Konkrétní (jednotlivý) výskyt daného obecného objektu či třídy

- **Kompilace**

  Proces překládání algoritmu zapsaného ve vyšším programovacím jazyce do strojového kódu

- **Kontrolka**

  Aktivní prvek formuláře ve vývojovém prostředí - například tlačítko, seznam nebo panel

- **Komponenta**

  Soubor vlastností ve vývojovém prostředí – například vlastnosti kontrolky

3 „Plugin“ nebo také „Zásuvný modul“

3.1 Co je to plugin (modul)

Název plugin je odvozen z anglického to plug-in (zasuň do, zasunout). Odtud tedy český pojem „zásuvný“ modul. Zjednodušeně řečeno je zásuvným modulem nebo pluginem označován soubor, který dokáže ovlivnit chod nebo funkcionalitu aplikace či programu. Plugin tedy nepracuje samostatně, ale je součástí nějaké aplikace, které jej využívá.

Pokud chce programátor umožnit pozdější funkční rozšiřování své aplikace bez nutnosti zásahu do zdrojového kódu, jistě využije právě možnosti pluginů.

3.2 Příklady pluginů

Jako jednoduchý modul může sloužit téměř jakýkoli soubor, který dokáže ovlivnit funkcionalitu aplikace. Pro bližší představu uvedu nejčastěji používané formáty souborů.

3.2.1 Prostý textový soubor

Jedná se o soubor, který je velmi jednoduše strukturovaný, poměrně přehledný a může obsahovat například funkce (soubor definic funkcí- využívaný například v php), konfigurační údaje, které ovlivňují samotnou aplikaci, nebo jazykové modifikace. Ačkoli se úplně nejedná o plugin jako takový, je využívání tohoto souboru poměrně časté a je tedy dobré, se o něm zmínit.

3.2.2 Strukturovaný soubor-XML

Další možnou variantou modulu může být strukturovaný soubor XML. Výhodou tohoto souboru oproti prostému textovému je rozšířená podpora ve vývojových aplikacích, a to především díky možnosti zadávat a třídit jednotlivá data do struktur a podstruktur.

Je tedy možné jej využít například jako efektivní konfigurační soubor, databázi, nebo jako modul řídící celou aplikaci na základě volaných metod, Aplikace se může například odkázat na „hlavičku“ v XML dokumentu a zavolá jednotlivé funkce podle zadaných parametrů.

Tento způsob byl využit například v rámci výuky, kdy měli studenti navrhnout textovou hru, jejíž děj a hratelnost by bylo možné měnit jednotlivými moduly. Jako tyto moduly byly využity právě soubory XML, které obsahovaly údaje o světě. Například velikost světa určená polem o velikosti 3x3 buněk, popis krajiny, informace o příšerách v jednotlivých částech tohoto světa, o vlastnostech těchto příšer a jsou - li agresivní. Pokud byli agresivní mohl modul aktivovat soubojový systém, který byl součástí aplikace a nastavit jeho parametry jako například počet životů a sílu útoku.

3.2.3 Další možné varianty modulů

Existuje mnoho dalších jednoduchých i složitějších formátů souborů, které jsou využívány jako pluginy. Může se jednat o standardně rozšířené formáty souborů jako jsou například :

- *.lng - soubor obsahující různé jazykové modifikace
- *.m3u - soubor obsahující seznam a popis skladeb pro multimediální přehrávače, takzvaný playlist
- *.ini - soubor, který má využívá specifickou strukturu – sekce a obsahy sekcí
- *.dat – soubor který může obsahovat různé zdroje (hudbu, obrázky)

Existují však také speciální formáty souborů navržené pouze pro specifické aplikace:

- *.wsz - soubor programu winamp obsahující data o vzhledu-„skinu“).
- *.plc – soubor který využívá multimediální program Aimp namísto výše zmíněných *.m3u playlistů.

Všechny tyto soubory je možné využívat k rozšiřování funkcí aplikace. Žádný z nich však neposkytuje takové možnosti, jako nejrozšířenější formát pluginů jakým je *.dll - Dynamic Link Library.

4 DLL- Dynamic Link Library

4.1 Co je to DLL

Jak již bylo v předchozí kapitole naznačeno, název DLL je odvozen z anglického „Dynamic Link Library“ neboli „Dynamicky linkovaná knihovna“.

V názvu: „Dynamicky linkovaná knihovna“, je prakticky obsažena celá podstata problému (podrobněji dále).

DLL knihovna je soubor, který se ke spuštěné aplikaci připojuje (linkuje) za chodu v případě potřeby. Jelikož knihovna DLL obsahuje kód funkcí a procedur, je velmi podobná běžnému programovému kódu. Může také obsahovat zdroje (například obrázky,texty nebo zvuky), tzv. resources . Je tedy možné říci, že mají svou vnitřní strukturu podobnou spustitelným *.EXE souborům.

Existuje mnoho odlišných názorů, je-li využívání DLL knihoven správnou volbou. Aby bylo možné posoudit všechny tyto názory, bude zapotřebí se o této problematice dozvědět více.

Prvotní myšlenka DLL knihoven byla dobrá. Do knihovny uložíme funkce nebo nějaké zdroje a různé aplikace je pak budou z těchto knihoven volata využívat. Z toho plyne jasná výhoda - a to, že více aplikací využije jednu knihovnu, která je však v paměti pouze jedenkrát, a navíc se z ní může dynamicky podle potřeby odstraňovat a znovu načítat. Tím dochází k šetření zdrojů a jejich sdílení mezi jednotlivými aplikacemi.

Někteří zkušenější programátoři by možná na tuto výhodu reagovali záporně cílenými argumenty-berme ale na vědomí fakt, že princip DLL knihoven je myšlenka kvalitní a že většina problémů se stabilitou vzniká jinde.

Zajímavostí DLL knihoven je, že není podstatné v jakém programovacím jazyku byly napsány. Jedná se totiž o již přeložené kódy a existuje jediný způsob, jak volat funkce uvnitř modulu. Dobrá je také skutečnost, že samotné DLL knihovny mohou využívat funkce z jiných DLL knihoven.

Další zajímavostí může být případ, kdy některé aplikace využívají DLL knihovny jako jazykový modul. Používáte - li například vývojové prostředí Sharp develop nebo Microsoft visulal studio, můžete velmi snadno vytvořit takový jazykový modul pouhým kliknutím na jednotlivé kontrolky a jejich komponenty. Lepším řešením však bývá prostý - daleko lépe editovatelný textový soubor.

Shrneme-li všechna výše uvedená fakta, můžeme říci že základními charakteristickými znaky DLL knihoven jsou především tyto:

- DLL knihovny dynamicky připojují (linkují) svůj kód a data až za chodu aplikace pouze v případě potřeby.
- DLL knihovny mohou sdílet procedury jiných DLL knihoven.
- Různé aplikace mohou využívat jednu DLL knihovnu a tím šetřit zdroje.

4.1.1 Výhody

Pro snazší pochopení rozdílných názorů na DLL knihovny je dobré znát výhody a nevýhody jejich použití. Zásadními výhodami jsou následující:

- Chceme-li rozšířit funkce aplikace, není potřeba zasahovat do kódu řídící aplikace a znovu jej kompilovat. Stačí nahradit nebo nahrát novou nebo aktualizovanou knihovnu.
- Pokud sdílí více aplikací jednu DLL, stačí k jejich aktualizaci nahradit starou knihovnu za novou.
- Nezanedbatelnou výhodou využívání DLL knihoven je podstatné zmenšení velikosti *.EXE souboru aplikace.
- Funkce nebo jednotlivé procedury z knihoven je možné do paměti načítat když je potřeba.
- Pokud již není zavolaná funkce nebo procedura zapotřebí, je možné jí z paměti uvolnit. Tím se šetří systémové zdroje.
- DLL knihovny nejsou závislé na jednom programovacím jazyku. Je tedy možné napsat jednu knihovnu v různých jazycích (C++,C#, Delphi a jiné).
- Stejně tak aplikace napsaná například v Delphi, dokáže pracovat s DLL napsanou v jazyce C#.

4.1.2 Nevýhody

Výše uvedené výhody by mohli svádět k názoru, že DLL knihovny jsou dokonalou myšlenkou. Není to však úplně pravda. Používání DLL knihoven může způsobit neskutečný chaos nejen ve funkcionalitě aplikací, které DLL využívají, ale i v operačním systému.

- Pokud často instalujeme a odebíráme aplikace, zůstávají po mnohých z nich v podadresáři %SYSTEMROOT%\system32 jejich knihovny. Deinstalační program je při odinstalaci neodstranil, protože zjistil, že mohou být sdílené a není jisté, zdali tuto knihovnu bude ještě nějaká aplikace využívat či ne. Tyto knihovny pak v tomto adresáři zůstávají a to až do doby, kdy se rozhodneme zformátovat disk. To je také jedna z příčin, proč se neustále zvětšuje obsah adresáře s nainstalovanými Windows.
- Také může nastat opačný případ a to, že knihovna na kterou se odkazuje spouštěná aplikace chybí. V tomto případě se obvykle zobrazí hlášení o chybě a aplikace se nespustí. Odstranit tento způsob lze dvěma způsoby:
    - Sehnat a nahrát chybějící DLL do adresáře
    - Reinstalovat aplikaci.

- Nejčastějším problémem sdílených DLL knihoven je situace nazývaná DLL HELL(dll peklo), kdy při instalaci nových aplikací může dojít k přepsání knihovny její starší verzí, nebo naopak verzí novější, která však není zpětně kompatibilní.Tím pádem aplikace, odkazující se na procedury, které se v nové knihovně nenacházejí, nebude fungovat správně nebo se vůbec nespustí. Nicméně s nástupem platformy .NET od společnosti Microsoft přišlo i řešení těchto problémů. Tato platforma obsahuje inteligentní správu verzí s konfigurovatelnou podporou obousměrné kompatibility, namísto odkazů na jména souborů.
- Přístup k volaným funkcím nebo procedurám je o trochu pomalejší. Toto zpomalení by se mohlo projevit například ve chvíli, kdy nalinkujem několik knihoven naráz a zavoláme z nich náročnější funkce.

5 Více o DLL

Aby bylo možné proniknou ještě o něco hlouběji do problematiky DLL knihoven, bude dobré se seznámit s vývojem sdílení DLL knihoven.

5.1 Rozdíl ve sdílení DLL knihoven v šestnáctibitových Windows a ve Windows32

Způsob realizace sdílení DLL knihoven v šestnáctibitových Windows a ve dvaatřicetibitových Windows je radikálně odlišný. Nejprve popíši, jakým způsobem fungovalo sdílení knihoven dříve, tedy ve Win16.[1]

5.1.1 Sdílení ve Win16

Ve Win16 bylo sdílení realizováno tak, že v operační paměti počítače byla vždy jen jedna instance knihovny (jeden jediný výskyt-knihovna byla v paměti jen jednou).

Tato instance byla globálně přístupná všem aplikacím a byla tedy připravena ke sdílení. 16bitová knihovna sice využívala zásobník své „mateřské“ aplikace (tedy aplikace, která ji zavedla do paměti), nicméně měla svůj vlastní paměťový prostor.

Knihovna byla zavedena do paměti pouze jednou. Při dalších pokusech o její zavedení (např. dalšími aplikacemi, které chtěly používat její funkce) se pouze zvýšila hodnota čítače, který byl součástí Windows. Windows tedy při každé žádosti o zavedení knihovny pouze inkrementovaly svůj čítač a naopak, žádala-li nějaká aplikace o uvolnění knihovny z paměti, čítač se dekrementoval.

Byla-li hodnota čítače rovna 0, knihovna se z paměti skutečně uvolnila.

Tato konstrukce samozřejmě programátorům umožňovala mnoho dalších „triků“, např. mohli prostřednictvím sdílené DLL knihovny realizovat komunikaci mezi aplikacemi.

Protože bylo řečeno, že každá DLL knihovna měla v paměti svoji část (tzv. segment), bylo možné využívat např. globální proměnné deklarované v knihovně k mezi-aplikační komunikaci.

Shrňme tedy způsob sdílení knihovny ve Win16

(aplikace A1 a A2 sdílejí knihovnu K1):

Požadavek	Provede se
Zaveď aplikaci A1 do paměti	Provede se zavedení
Zaveď aplikaci A2 do paměti	Provede se zavedení
Aplikace A1 žádá funkci z knihovny K1	Zavede se K1 do paměti a inkrementuje se čítač pro K1, hodnota čítače = 1
Aplikace A2 žádá funkci z knihovny K1	Inkrementuje se čítač pro K1, hodnota čítače = 2
Aplikace A2 chce sdělit aplikaci A1, že výsledek výpočtu je 150	Uloží se hodnota 150 do globální proměnné Mezivýsledek, která je deklarována v knihovně K1
Aplikace A1 chce zjistit výsledek operace prováděné aplikací A2	Přečte se hodnota proměnné Mezivýsledek, která je deklarována v knihovně K1
Uvolni aplikaci A2 z paměti	Aplikace se uvolní, dekrementuje se čítač pro K1, hodnota čítače = 1
Uvolni aplikaci A1 z paměti	Aplikace se uvolní, dekrementuje se čítač pro K1, hodnota čítače je rovna nule, takže se uvolní také knihovna K1 z paměti

Tato jednoduchá tabulka demonstruje, jakým způsobem probíhalo sdílení knihoven a komunikace mezi aplikacemi ve Win16.

5.1.2 Sdílení ve Win32

Ve Win32 již neplatí jednoduchá pravda, že knihovna je v paměti zavedena nejvýše jedenkrát.

Složitější je totiž i vlastní zavádění aplikací do operační paměti. Toto zavádění je však již poněkud vyšší matematikou, proto je zmíněno v **poznámce pro pokročilé programátory**.

Pro pochopení momentálně postačí, řekneme-li, že každý proces (každá běžící aplikace) ve Windows má k dispozici část (virtuálního) adresového prostoru – to si můžeme představit například tak, že každá spuštěná aplikace má oplocenou určitou část operační paměti počítače a v ní může libovolně operovat. Nesmíme však zapomenout na fakt že, například Windows 95 spravují paměť úplně jinak než Windows NT.

Jakmile se aplikace namapuje do tohoto adresového prostoru, zjistí se, jaké dynamicky linkované knihovny hodlá tato aplikace využívat a ty se následně namapují do stejného adresného prostoru jako je tato aplikace.

Přestože také ve dvaatřicetibitových Windows je možné knihovny sdílet, každá aplikace mapuje knihovnu do svého vlastního adresového prostoru.

Čítač, o kterém jsme se zmiňovali při popisování situace ve Win16, je také přítomen, nyní ovšem čítá pokusy o zavedení knihovny pouze v rámci jedné, příslušné aplikace.[1]

5.1.3 Poznámka pro pokročilejší programátory

V okamžiku, kdy Windows dostanou žádost o zavedení aplikace do operační paměti, vytvoří se tzv. objekt procesu jádra (zpravidla nazývaný jen proces).

Tento proces dostane k dispozici určitý (virtuální) paměťový prostor. Poté se dále vytvoří tzv. objekt mapování souboru jádra. Tento druhý zmiňovaný objekt zabezpečí mapování spustitelného souboru do adresového prostoru procesu. Aplikace se tedy mapuje ze svého umístění na disku.[1]

6 Základní informace k ukázkové aplikaci

Součástí této publikace je i ukázkový program, který by měl obzvláště začínajícím programátorům, pomoci pochopit funkcionalitu a problematiku modulární aplikace.

- Kompletní zdrojové kódy jsou k dispozici v příloze na CD. V následujících kapitolách bude uvedena pouze ta část kódu, která se vztahuje k tématu této publikace.

- Předpokladem k pochopení problematiky kódu je alespoň základní znalost programovacího jazyka C# na platformě .NET Framework.

- Všechny zdrojové kódy jsou pro snadnější pochopení podrobně komentovány.

6.1 Vývojové prostředí

Aby bylo možné vytvořit vlastní aplikaci s využitím DLL knihoven, bude zapotřebí nějakého vývojového prostředí, které podporuje námi ovládaný programovací jazyk.

- Ukázková aplikace je psaná v jazyce C# za využití rozhraní .NET Framework 2.0 od společnosti Microsoft.

- Využito bylo vývojové prostředí Sharp Develop, které je na internetu (http://www.sharpdevelop.com) volně k dispozici.

- Další možnou variantou vývojového prostředí je dnes již velmi rozšířené Visual Studio 2005 (případně novější verze 2008) od společnosti Microsoft. Tento produkt však není poskytován bezplatně
    - Lze však využít takzvané Express Editions těchto prostředí, které jsou k dispozici na domovských stránkách společnosti Microsoft (http://www.microsoft.com) zdarma. Toto vývojové prostředí je omezené obsahem nástrojů základní úrovně.

6.1.1 Seznámení s rozhraním .NET Framework

.NET Framework je nadstavba operačního systému. Obsahuje balík knihoven, jehož jednotné prostředí nabízí své objektové třídy vyšším programovacím jazykům, aby jej využily k práci při řešení konkrétních úloh nezávisle na použitém programovacím jazyce. (6)

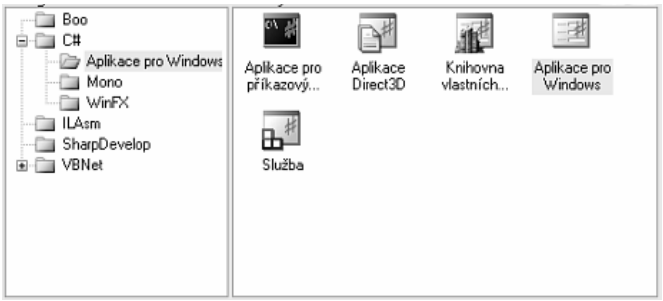
Operační systém Windows Vista obsahuje rozhraní .NET Framework 3.0 již standardně. U starších verzí operačních systémů Windows je potřeba platformu .NET Framework doinstalovat. Toto rozhraní bylo pro ukázkovou aplikaci zvoleno především kvůli jeho jednoduchosti na pochopení a architektuře, díky níž je ideální právě pro tvorbu modulárních aplikací.

6.2 Stručné seznámení s prostředím Sharp develop

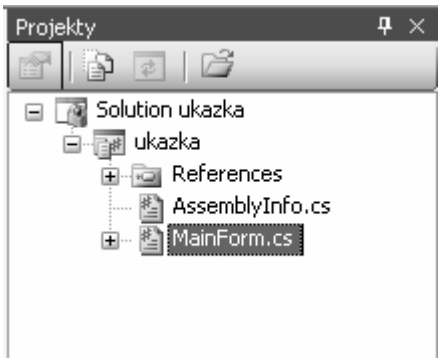
Jelikož ne každý zná práci s prostředím Sharp develop, uvedu několik základních kroků potřebných k vytvoření nového projektu.

6.2.1 Spuštění a vytvoření nového projektu

1. Aplikace spustíme souborem SharpDevelop.exe
2. Z hlavního menu zvolíme položku Soubor dále pak Nový a nakonec řešení.
3. Z otevřeného dialogu kliknutím zvolíme požadovaný programovací jazyk (v případě ukázkové aplikace C#) a zobrazíme tak nabídku možností. Jako první vytvoříme řídící aplikaci.
    - Pro vytvoření řídící aplikace zvolíme následující postup:
        - Rozbalíme položky složky označené C#
        - Kliknutím zvolíme položku „Aplikace pro Windows“
        - Ze zobrazené nabídky zvolíme ikonu „Aplikace pro Windows“



- Do kolonky „Název“ zadáme název aplikace.
- Zvolíme umístění, kam se má celý projekt uložit
- Ostatní kontrolky ponecháme beze změn a potvrdíme tlačítkem vytvořit
- Nyní je vytvořen projekt se kterým můžeme dále pracovat. Vlastní kód bude obsahem hlavního formuláře pojmenovaného MainForm.cs. Tento formulář se zobrazí po dvojím kliknutí na položku v seznamu Projekty.



6.2.2 Vytvoření DLL knihovny

DLL knihovnu je možné vytvořit několika metodami. Jeden z možných způsobů jak DLL knihovnu vytvořit, je využití podpory ve vašem vývojovém prostředí. Tento způsob je rychlý , pohodlný a nevyžaduje žádné zvláštní znalosti.

- Vytvoření DLL knihovny pomocí prostředí Sharp develop
  - Zopakujeme postup uvedený při vytváření nové aplikace a to až k bodu 2.
  - Označíme kliknutím položku s názvem C#
  - Ze zobrazené nabídky zvolíme ikonu „Knihovna tříd“
  - Opět zadáme název projektu (například plugin1) a zvolíme cíl, kam se má uložit
  - Potvrdíme tlačítkem vytvořit
  - Další postup je stejný jako při tvorbě aplikace.
  - Zkompilováním kódu získáme výslednou DLL knihovnu

- Vytvoření DLL knihovny pomocí příkazového řádku
    - Nejprve je potřeba vytvořit třídu funkcí(*.CS)
    - Tuto třídu můžeme napsat v libovolném textovém editoru, nesmíme však zapomenout uložit ho jako *.CS
    - Pro usnadnění práce můžeme vytvořit spustitelný soubor *.CMD nebo *.BAT
    - Do tohoto souboru uložíme následující příkaz: %WINDIR%\Microsoft.NET\Framework\v2.0.50727\csc.exe /target:library UkazkaDll.cs
    - Všechny soubory *.CS a *.CMD nebo *.BAT nahrajeme do jedné složky
    - Spustíme „tvořící“ soubor. Po provedení příkazu získáme výslednou DLL knihovnu

- Jednotlivé prvky tvořícího příkazu:
    - Volání programu který nám zkompiluje vytvořenou třídu funkcí.
      %WINDIR%\Microsoft.NET\Framework\v2.0.50727\csc.exe
    - Místo(cílový adresář), kde se DLL vytvoří
      /target:library
    - Název zdroje
      UkazkaDll.cs

7 Ukázková aplikace

7.1 Krátké připomenutí

Aby bylo patrné, proč je využívání modulů tolik rozšířenou metodou, krátce připomenu, jak vypadá aplikace bez využití pluginů.

1. Jakákoli změna v kódu aplikace vyžaduje novou kompilaci a tím pádem i novou distribuci aktualizovaného *.EXE souboru.
2. Obsah komponent programu (třídy, resources, atd.) jsou nedělitelnou součástí aplikace a tím narůstá její velikost.
3. Součástí *.EXE souboru mohou bít i podprogramy a deklarace které aplikace nemusí vůbec používat.
4. Procedury se načtou automaticky po spuštění do paměti (nelze totiž stanovit, že se procedura má vykonat až v určitém okamžiku) a tím dochází k zahlcování zdrojů.

7.2 Úvod

- Uváděný kód by neměl žádné větší využití, pokud by nebyl součástí aplikace. Využil jsem tedy mnou vytvořený internetový prohlížeč do kterého byl ukázkový kód implementován.
- Celý projekt byl sestaven ve vývojovém prostředí Sharp Develop za využití rozhraní .NET Framework 2.0
- Kompletní aplikace i veškeré zdrojové kódy a jsou přiloženy na CD, které je součástí této publikace.

7.3 Základní rozhraní

Nejprve je nutné nadefinovat rozhraní, přes které bude probíhat komunikace. Vytvoříme hlavní řídící knihovnu pomocí které budou ostatní pluginy komunikovat s aplikací.

Tuto řídící DLL knihovnu je pojmenovaná Main_Plugin („hlavní plugin“) a každá další DLL knihovna určená pro tuto aplikaci bude muset implementovat její následující rozhraní. [2]

7.3.1 Rozhraní KomPluginMain

```
Definujeme rozhraní, pomocí kterého bude jádro(více dále) komunikovat
s Pluginy
public interface KomPluginMain
{
    Definujeme metodu kterou jádro volá a předává i instanci jádra.
    void Load(KomAplikaceMain aplikace);

    Definujeme metodu kterou jádro pluginu předá pole všech pluginů.
    Díky tomu může probíhat komunikace mezi jednotlivými pluginy.
    void PluginsLoaded(KomPluginMain[] pluginy);

    Definujeme demonstrační metodu, kterou jádro volá po zavolání.
    (například po kliknutí na položku v seznamu aplikace)
    void Proved();
```

Definujem metody, které budou slouží k vzájemné komunikaci mezi pluginy.

```
object Metoda1(object param);
object Metoda2(object param);
```

Definujeme metodu, díky které určíme, má-li být modul načten automaticky po spuštění aplikace

```
bool AutoStart
{
   get;
}
```

Definujeme metodu, která zjistí název aktuálního pluginu. Tato metoda bude později využita pro identifikaci jednotlivých pluginů

```
string Nazev
{
   get;
}
```

Definujeme metodu, která zjistí název aktuálního pluginu. Tato metoda bude později využita pro identifikaci jednotlivých pluginů

```
   string Tiulek
   {
      get;
   }
}
```

- Jako první je volána metoda **Load()**, kterou pluginu předáme instanci jádra(tedy rozhraní, pomocí něhož bude plugin komunikovat s jádrem). Tuto metodu uložíme jako statický člen. Díky tomu je pak možné z kteréhokoli místa v pluginu komunikovat s jádrem.
- Jakmile u všech pluginů proběhne metoda **Load()**, zavolá se metoda **PluginsLoaded()**. Získané pole rozhraní si opět uložíme do statického členu a komunikujeme s ostatními pluginy.
- Pomocné metody **Metoda1()** a **Metoda2()** nemusí budoucí pluginy implementovat. Postačí když vrátí hodnotu null a jádro je nebude nikdy volat. Slouží jako metody pro speciální funkce, které mohou využívat ostatní pluginy.

7.3.2 Rozhraní KomAplikaceMain

Nyní v tomto modulu nadefinujeme rozhraní, pomocí kterého bude možné komunikovat s jádrem

```
public interface KomAplikaceMain
{
    Ukázková funkce-maximalizuje okno
    void Maximalizuj();

    Ukázková funkce-obnovný okno z maxima
    void Normal();
}
```

- Metoda Maximalizuj slouží jako ukázka toho, jak lze funkcionálně komunikovat pomocí pluginu s aplikací. Tato metoda maximalizuje hlavní okno aplikace
- Metoda Normal obnoví okno aplikace z maxima .
- Po kompilaci projektu získáme hlavní DLL knihovnu kterou budou využívat ostatní pluginy i aplikace.

7.4 Jádro aplikace

Aby bylo umožněna komunikace mezi pluginy a aplikací-přesněji s hlavním formulářem aplikace, je potřeba navrhnout takzvané jádro. Právě díky němu je tato komunikace možná.

7.4.1 Navržení jádra

Vytvoříme třídu pomocí které budou moci pluginy komunikovat s aplikací Tato třída implementuje rozhraní KomAplikaceMain

```
public class JadroAplikace : KomAplikaceMain
{
```

Vytvoříme delegát události maximalizuj

```
  public delegate void MaximalizujEventHandler(object
  sender, EventArgs e);
```

Vytvoříme delegát události normal

```
  public delegate void NormalEventHandler(object
  sender, EventArgs e);
```

Definujeme událost pro delegáty a pojmenujeme je

```
  public event MaximalizujEventHandler
  MaximalizujEvent;
  public event NormalEventHandler NormalEvent;
```

Definujem metodu Maximalizuj kterou může volat jakýkoli plugin

Metoda Maximalizuje hlavní okno aplikace

```
public void Maximalizuj()
{
    ProvedMax(this);
}
```

Definujeme metodu, která vyvolá událost MaximalizujEvent.

```
protected void ProvedMax(object sender)
{
    if(MaximalizujEvent != null)
    MaximalizujEvent(sender, new EventArgs());
}
```

Definujem metodu Normal kterou může volat jakýkoli plugin

Metoda obnoví okno aplikace z maxima

```
public void Normal()
{
    ProvedNorm(this);
}
```

Metoda které vyvolá údálost NormalEvent.

```
protected void ProvedNorm(object sender)
{
    if(NormalEvent != null) NormalEvent(sender, new
    EventArgs());
}
}
```

- Jelikož se metody **Maximalizuj()** a **Normal()** nemůžou dostat přímo k hlavnímu formuláři, bylo nutné nadefinovat události, na které může formulář reagovat.

7.5 Hlavní formulář aplikace

Hlavní formulář aplikace obsahuje zdroje - resources , kód designeru a základní metody s funkcemi, které by měla aplikace umět i bez přídavných modulů .

7.5.1 Komunikace s jádrem

Aby mohl formulář aktivně reagovat na pluginy, musíme vytvořit instance odkazující se na metody jádra.

```
public static JadroAplikace aplJadro;
```

Dále vytvoříme seznam, do kterého budeme později načítat jednotlivé pluginy umístěné v podsložce aplikace.

- Jelikož byl tento projekt navržen s ohledem na kompatibilitu mezi .NET Frameworkem 1.1 a .NET Frameworkem 2.0 (Windows XP, stále ještě převážně rozšířený operační systém u nás, standardně neobsahuje .NET Framework 2.0 a vyšší, proto byl brán ohled na verzi starší, která je součástí základní instalace), použijeme třídu ArrayList.

Ve verzích .NET Framework 2.0 a vyšší bude lepší využít generickou třídu List.

Vytvoříme statický seznam načtených pluginů.

```
public static ArrayList pluginy;
```

7.5.2 Metody při načítání hlavního formu

Aby vše fungovalo tak jak má, je zapotřebí provést několik operací hned při spouštění aplikace. Do události hlavního formuláře MainFormLoad() přidáme následující metody a funkce.

```
Inicializujeme seznam do kterého budeme načítat aktivní pluginy
pluginy = new ArrayList();

Nyní inicializujeme instance jádra aplikace, abychom mohly volat jednotlivé funkce
aplJadro = new JadroAplikace();

Nastavíme přihlášení k události MaximalizujEvent.
Tuto událost pak může volat libovolný plugin
aplJadro.MaximalizujEvent += new
PluginAplikace.JadroAplikace.MaximalizujEventHandler(
aplJadro_MaximalizujEvent);

Nastavíme přihlášení k události NormalEvent.
Tuto událost pak může volat libovolný plugin
aplJadro.NormalEvent += new
PluginAplikace.JadroAplikace.NormalEventHandler(aplJa
dro_NormalEvent);
```

Volání metody, která načte aktivní pluginy uložené ve složce Plugins.

```
LoadPlugins();
```

Volání metody, která zaplní seznam v aplikaci aktivními pluginy

```
ZaplnList();
```

Volání metody, která spustí metodu Proved() všech pluginů, které mají nastavené automatické načítání

```
PlugAutoStart();
```

- Pořadí metod je důležité. Nelze například prohodit metodu ZaplnList() s metodou LoadPlugins(). Tato změna by měla za následek nesprávnou funkčnost programu. (do seznamu by se načetlo prázdné pole). Toto upozornění je určeno méně zkušeným programátorům. Ač by se mohlo zdát, že jde o zcela logickou záležitost, jedná se o poměrně častou, převážně začátečnickou chybu. Z tohoto důvodu považuji za dobré tuto poznámku uvést.

Nadefinujeme metody pro události MaximalizujEvent a NormalEvent. Tyto metody budou samostatné - nebudou načítány ihned při spuštění aplikace.

Metoda obsluhuje událost maximalizace (MaximalizujEvent) okna hlavního Formu

```
private void aplJadro_MaximalizujEvent(object sender,
EventArgs e)
{
  WindowState = FormWindowState.Maximized;
}
```

Metoda obsluhuje událost obnovení (NormalEvent) okna hlavního Formu

```
private void aplJadro_NormalEvent(object sender,
EventArgs e)
{
     WindowState = FormWindowState.Normal;
}
```

- Ve zdrojovém kódu ukázkové aplikace jsou všechny zmíněné události a metody pro přehlednost zahrnuty v Regionu označeném „Inicializace & Eventy(události) „

7.5.3 Jednotlivé metody

- Díky metodě LoadPlugin můžeme zaplnit list „pluginy“ a získat informace o jednotlivých modulech.

```
private void LoadPlugins()
{
```

Nastavíme umístění, odkud se mají všechny pluginy načítat a uložíme jej do proměnné typu string.

```
string pluginDir =
Application.StartupPath+@"\\Plugins";
```

Opět zajistíme, aby nedocházelo k opakovanému výpisu pluginů a informací tím, že vyprázdníme list „pluginy“ a kontrolce label2, která zobrazuje počet všech načtených pluginů nastavíme prázdnou hodnotu(lze použít i hodnotu null namísto prázdného řetězce)

```
Label1.Text = "";
pluginy.Clear();
```

Nejprve zkontrolujeme, zdali existuje adresář s Pluginy a zdali má cenu volat další funkce.

```
if(Directory.Exists(pluginDir))
{
```

Pokud adresář existuje, nastavíme filtr pro prohledávané soubory a tím zajistíme, že budou načteny pouze DLL knihovny. Filtr najde všechny soubory s příponou *.DLL a uloží je do pole „data“

```
string[] data = Directory.GetFiles(pluginDir,
"*.dll");
```

Dále začneme procházet již odfiltrované soubory *.DLL

```
for(int i = 0; i < data.Length; i++)
{
```

Jelikož složka Plugins obsahuje i řídící knihovnu Main_plugin.dll je zapotřebí zajistit, aby se tato knihovna nenačetla společně s ostatními. Nastavíme tedy podmínku, která nám tento modul při načítání vyloučí.

```
if(data[i] != pluginDir+@"\\Main_plugin.dll")
{
```

Ošetříme odchytávání případných chyb a vytvoříme instanci třídy Assembly obsahující prvky seznamu „data“.

```
try
{
Assembly asbl = Assembly.LoadFile(data[i]);
```

Vytvoříme instanci třídy "plugin", která implementuje rozhraní "KomPluginMain".

```
KomPluginMain plugin =
(KomPluginMain)asbl.CreateInstance("PluginAplika
ce.Plugin");
```

Pokud se podaří získat rozhraní ke komunikaci s pluginem, přidáme jej do seznamu „pluginy“.

```
if(plugin != null) pluginy.Add(plugin);
}
catch(Exception e)
{
MessageBox.Show("Chyba při načítání pluginu
"+data[i]+": "+e.Message, "PluginAplikace",
MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

- Vyskytne-li se chyba při načítání pluginů (obsahuje-li například složka cizí plugin, nebo plugin neplatný), zobrazí se varování o chybě a plugin se nenačte.

```
}
}
}
```

Zavoláme metodu Load všech pluginů načtených v seznamu „pluginy“.

```
foreach(KomPluginMain plugin in pluginy)
{
   plugin.Load(aplJadro);
}
```

Vytvoříme pole rozhraní všech načtených pluginů „plugins“

```
KomPluginMain[] plugins = new
KomPluginMain[pluginy.Count];

for(int a = 0; a < pluginy.Count; a++)
{
   plugins[a] = (KomPluginMain)pluginy[a];
}
```

Po inicializaci všech pluginů v seznamu „pluginy“, předáme každému z nich pole rozhraní ke vzájemné komunikaci.

```
foreach(KomPluginMain plugin in pluginy)
{
   plugin.PluginsLoaded(plugins);
}
```

Aby mohl uživatel snadno sledovat, načetl-li se v aplikaci nějaký další plugin, zobrazíme do kontrolky label1 počet aktivních pluginů, který se při každém znovunačítání modulů aktualizuje.

```
label1.Text = "Počet aktivních pluginù:
"+pluginy.Count.ToString();
}
```

- Metoda ZaplnList() demonstruje, jak je možné načíst a zobrazit jednotlivé moduly v kontrolce aplikace.

```
private void ZaplnList()
{
```

Aby nedocházelo k opakovanému výpisu při znovuzavolání této metody, je potřeba vyprázdnit všechny komponenty, které obsahují načtené pluginy. Prvním krokem bude tedy vyprázdnění formulářových kontrolek.

```
listPluginy.Items.Clear();
listPluginyAuto.Items.Clear();
```

Projdeme seznam aktivních pluginů „pluginy“
Pokud se jedná o samospustitelný plugin, zajistíme, aby byl viditelně oddělen od ostatních tak, že jej načteme do jiné připravené kontrolky formuláře.

```
foreach(KomPluginMain plugin in MainForm.pluginy)
{
    if(plugin.AutoStart==true)
    {
        listPluginyAuto.Items.Add(new
        ListViewItem(new string[]{plugin.Nazev,
        plugin.Titulek}));
    }
```

Pokud se jedná o běžný plugin, přiřadíme jej do kontrolky formuláře, která umožňuje uživateli s modulem dále pracovat.

```
    else
        {
            listPluginy.Items.Add(new ListViewItem(new
            string[]{plugin.Nazev, plugin.Titulek}));
        }
    }
}
```

- Někdy je zapotřebí při načítání aplikace spustit plugin bez zásahu uživatele. Například chceme-li zavolat metody, které mají být vykonávány automaticky – načtení grafického rozhraní, klíčové funkce aplikace, nebo zjišťování aktualizací pro danou aplikaci
- Pro tento případ je definována metoda PluginAutoStart() díky které aplikace pozná, že má tento modul zavádět sama při spuštění.

```
private void PlugAutoStart()
{
   Projdeme seznam všech načtených pluginů „pluginy“ a získáme hodnotu
   z funkce Autostart. Pokud dotaz vrátí hodnotu true, spustí se metoda pluginu
   Proved().
   foreach(KomPluginMain plugin in pluginy)
   {
      if(plugin.AutoStart == true)
      {
            plugin.Proved();
      }
   }
}
```

- Posledním krokem této kapitoly je nastavení události pro spuštění jednotlivých modulů.
- Vytvoříme posluchač, který reaguje při dvojitém kliknutí na položku v kontrolce aplikace a poté spustí metodu Proved() zvoleného pluginu.

```
void ListPluginyDoubleClick(object sender,
System.EventArgs e)
{
```

Abychom mohli určit, který plugin voláme, využijem vlastnosti text u všech pluginů(již načtený v seznamu kontrolky formuláře) a tím určíme aktuálně zvolený modul. Získanou hodnotu z označené položky uložíme do proměnné typu string.

```
string pluginName =
listPluginy.SelectedItems[0].Text.ToString();
```

```
    Projdeme seznam všech načtených pluginů v seznamu „pluginy“
    foreach(KomPluginMain plugin in pluginy)
    {
        Podle zjištěné hodnoty Text zjistíme, jedná li se o plugin se kterým
        chceme pracovat. Pokud ano, zavoláme jeho metodu Proved()
        if(plugin.Nazev == pluginName)
        {
            plugin.Proved();
        }
    }
}
```

- Ve zdrojovém kódu ukázkové aplikace jsou všechny zmíněné události a metody pro přehlednost zahrnuty v Regionu označeném „Metody pluginu„

7.6 Vytvoření Pluginu

K vytvoření jednotlivých modulů využijeme prostředí Sharp develop a to především kvůli efektivnějšímu způsobu psaní tříd pluginu. Při tvorbě pluginu, který obsahuje formulář, nám Sharp develop zásadním způsobem usnadní práci při psaní kódu pro design tohoto formu.

7.6.1 Kostra pluginu

Vytvoříme kostru, která bude pro všechny pluginy společná. Do této kostry pak vložíme vlastní funkce, metody nebo formuláře.

```
Vytvoříme třídu pomocí které bude plugin komunikovat s jádrem. Tato třída
implementuje rozhraní KomAplikaceMain
public class Plugin : KomPluginMain
{
    Vytvoříme statická instance odkazující se na metody jádra..
    public static KomAplikaceMain aplikaceMain;
    Vytvoříme statický seznam načtených pluginů
    public static ArrayList plugins;

    public Plugin()
    {
        Inicializujeme seznam plugins
        plugins = new ArrayList();
    }
```

Pomocí této metody určíme, má-li se plugin načítat automaticky po spuštění aplikace. Pokud nechceme, aby se plugin spustil sám, nastavíme hodnotu na **false**. V opačném případě nastavíme hodnotu metody na **true**.

```
public bool AutoStart
{
    get
    {
        Nastavíme automatické načítání pluginu(true-ano, false-ne)
        return false;
    }
}
```

Pomocí této metody identifikujem jednotlivé pluginy. Název by měl být stručný a výstižný aby uživatelům aplikace byla na první pohled patrná funkce pluginu.

```
public string Nazev
{
    get
    {
        Nastavíme jméno pluginu, které může být zobrazeno aplikací
        return "Maximalizuj";
    }
}
```

Pomocí této metody získáme doplňující informace o pluginu. Tyto informace pak mohou bít načteny aplikací aby byl uživatel obeznámen s funkcí tohoto pluginu

```
public string Titulek
{
    get
    {
        Nastavíme popis pluginu, který může být zobrazen aplikací
        return "Maximalizuje okno aplikace";
    }
}
```

Metoda kterou zavolá jádro předává instanci

```
public void Load(KomAplikaceMain aplJadro)
{
        Předá instanci jádra
        aplikaceMain = aplJadro;
}
```

Tato metoda předá pole všech pluginů. Díky tomu může probíhat komunikace mezi jednotlivými pluginy.

```
public void PluginsLoaded(KomPluginMain[] pluginy)
{
        plugins.AddRange(plugins);
}
```

Pomocná metoda1 pomocí které mohou jednotlivé pluginy vzájemně komunikovat. Tato metoda slouží pouze pro speciální funkce, které pak mohou využívat ostatní pluginy.

```
public object Metoda1(object param)
{
    Nastaveno null-tedy tuto metodu nebude jádro nikdy volat
    return null;
}
```

Pomocná metoda2 pomocí které mohou jednotlivé pluginy vzájemně komunikovat. Tato metoda slouží pouze pro speciální funkce, které pak mohou využívat ostatní pluginy.

```
public object Metoda2(object param)
{
    Nastaveno null-tedy tuto metodu nebude jádro nikdy volat
    return null;
}
```

Metoda, která se spustí v případě zavolání jednotlivého pluginu a provede definované operace či funkce.

```
    public void Proved()
    {
        Zde bude kód funkcí, které bude plugin provádět v případě spuštění.
        Jednotlivé funkce obsažené v ukázkových modulech jsou uvedeny dále.
    }
}
```

7.6.2 Použité funkce v ukázkových pluginech

Jednotlivé funkce pluginů jsou již v rukách programátora. Záleží však na tom, co aplikace umožní pluginu řídit. V případě ukázkové aplikace, můžou být funkčně ovlivněny dvě metody-Maximalizuj a Normal.

V důsledku není těžké, pracovat s metodami, jsou li dobře navrženy již v samotném jádru aplikace. V případě Maximalizuj() a Normal() zavolá plugin z celého složitého kódu aplikace jednodše metodu.

Touto funkcí plugin maximalizuje okno

```
aplikaceMain.Maximalizuj();
```

Touto funkcí plugin obnoví okno z maxima

```
aplikaceMain.Normal();
```

Tato funkce spustí všechny přístupné pluginy

```
foreach(KomPluginMain plugin in plugins)
{
   Ověříme zdali se nejedná o náš plugin a o plugin načítaný automaticky
   if(plugin.Nazev != Nazev && plugin.AutoStart!=
true)
   {
```

```
   Zobrazíme informaci o aktuálně spouštěném pluginu a zavoláme jeho funkci
   MessageBox.Show("Nyní se spustí plugin:
"+plugin.Nazev);
   plugin.Proved();
   }
}
```

7.6.3 Plugin s formulářem

Protože plugin nemusí fungovat pouze jako funkcionální prvek, obsahuje ukázková aplikace také variantu, která po zavolání spustí nový formulář, který obsahuje jednoduché internetové rádio(AB-radio.cz).

Abychom mohli pluginem otevřít nový formulář, který bude vykonávat nějakou činnost, je potřeba vytvořit jeho vlastní třídu.

V případě ukázky se třída obsahující funkce a vzhled formuláře jmenuje RadioGui.cs.

Třídu pro formulář s metodami a funkcemi je možné vytvořit jako nový projekt a pak přidat tuto existující referenci do projektu pluginu. Je také bezpečnější zadávat design formuláře přímo to třídy. Stačí po dokončení navrhování zkopírovat zdroj ze souboru „nazevTridy.Designer.cs“.

Protože otevíraný formulář může obsahovat libovolný obsah, nebudu jej více rozebírat a přejdu přímo ke způsobu jak pomocí pluginu zavolat požadovanou třídu s formulářem.

Metoda proved() vykoná jednoduchou funkci, která spustí třídu s formulářem obsahující přídavnou aplikaci.

```
public void Proved()
{
    Inicializujeme  třídu RadioGui a pojmenujeme jí „radio“
    RadioGui radio = new RadioGui();

    Zavoláme metodu Show která nám otevře požadovaný form
    radio.Show();
}
```

7.7 Shrnutí

- Definice rozhraní řídícího modulu, který umožní komunikaci mezi pluginy a jádrem aplikace.
    - Po kompilaci získáme soubor Main_plugin.dll
- Vytvoření třídy jádra pomocí které mohou pluginy komunikovat s aplikací.
- Vytvoření hlavní třídy aplikace a inicializace jádra.
- Definování metod pro načítání a práci s pluginy.
    - Po kompilaci získáme řídící aplikaci která pomocí jádra a řídícího modulu pracuje s pluginy ve složce Plugins
- Vytvoření jednotlivých pluginů pro aplikaci, jejíž funkce mohou ovlivni nebo rozšířit.
    - Po kompilaci získáme pluginy, které je potřeba nahrát do složky programu – Plugins
    - Běžné pluginy je možné přidávat a načítat za běhu aplikace
    - Automaticky spustitelné pluginy se projeví až po restartování aplikace

8 Závěr

Jak na závěr shrnout problematiku spojenou s využíváním využívání DLL knihoven? Pluginy ve formě DLL knihoven jsou bezesporu dobrou myšlenkou. Je sice pravda, že je nelze využít vždy, všude a na všechno, nicméně jejich nasazení může programátorům práci zefektivnit a zjednodušit. Že se jedná o kvalitní myšlenku naznačuje i fakt, že tyto knihovny dnes využívá převážná většina aplikací a to i přes veškeré nedostatky s nimi spojené. Metody na odstraňování těchto nedostatků jsou navíc neustále zlepšovány a tak mají pluginy ve formě DLL knihoven před sebou slibnou budoucnost.

Reference

[1] KADLEC, Václav. Programovat DLL knihovny může úplně každý. Živě.cz [online]. 2001 [cit. 2008-03-03]. Dostupný z WWW: <http://www.zive.cz>.

[2] FAJFR, Stanislav. C# aplikace s podporou pluginů. Programujte.cz [online]. 2006 [cit. 2008-03-11]. Dostupný z WWW: <http://www.programujte.cz>.

Použité zdroje

Internet:

(1) http://www.codersource.net *24.4.2007*

(2) http://www.codeproject.com *17.10.2007*

(3) http://www.devhood.com *21.12.2007*

(4) http://www.pcsvet.cz *21.12.2007*

(5) http://www.codeproject.com *12.01.2008*

(6) http://msdn2.microsoft.com/cs-cz/vcsharp/default(en-us).aspx *12.01.2008*

(7) http://www.zive.cz *3.3.2008*

(8) http://www.programujte.com *11.3.2008*