Jazyk C# a platforma .NET

Katedra softwarového inženýrství
Fakulta informačních technologií
České vysoké učení technické v Praze

Úvod
BI-DNP

Platforma .NET

Programy napsaná jazyce C# (a dalších programovacích jazycích) jsou určeny pro spouštění na platformě .NET (.NET Framework) – runtime (či prostředí), které poskytuje základní infrastrukturu pro běh programů (též **CLI** – Common Language Infrastructure – standard a **CLR** – Common Language Runtime – jeho realizace).

Překlad programů

.NET je „inspirován" jazykem Java. Podobně jako v Javě nejsou programy (nejen v C#) překládány přímo do strojového jazyka cílového počítače, ale do jakéhosi mezijazyka (virtuálního jazyka) – **CIL** (Common Intermediate Language - dříve IL nebo MSIL). Jde o nízkoúrovňový objektově orientovaný jazyk zásobníkového typu (stack-based) – obdoba bytecode Javy. Není však interpretován, ale při zavádění programů do paměti (resp. při prvním volání metod objektů) je kompilován pomoci **JIT** (Just In Time compiler) do strojového kódu cílového počítače a teprve potom je program spouštěn (obvykle).

Programy v .NET jsou (standardně) tzv. typově bezpečné (**type-safe**) – během JIT kompilace se kontroluje (pomocí tzv. metadat - viz dále) např. to, zda akce prováděné kódem programu odpovídají deklaracím apod. Konkrétní příklady: zda indexy pole nejsou mimo rozsah, zda se volají pouze definované metody tříd, zda metody používají deklarované počty a typy parametrů atd. (Tyto typové kontroly ovšem mohou být vypnuty, i když běžně se to nedělá.)

Důsledkem je to, že jeden takový program nemůže „ublížit" druhému. Díky tomu je možné v rámci jednoho klasického procesu Windows spustit více .NET programů (vytvořit více tzv. **aplikačních domén**). Výhodou je to, že pro typově bezpečné programy je daleko jednodušší a efektivnější jejich řízení operačním systémem – z jeho hlediska jde o jediný proces. Naopak pro různé klasické samostatné procesy je např. přepínání mezi nimi daleko „nákladnější".

Programy, které využívají .NET (jsou překládány do CIL v rámci tzv. assemblies a používají služeb platformy .NET – oboje viz dále) tvoří tzv. **managed** (spravovaný, řízený) kód. Ovšem je možné, aby spolupracoval řízený i neřízený (**unmanaged**) kód. V neřízeném kódu bývají psány starší programy, nebo komponenty, u kterých se vyžaduje zvláště vysoká efektivita.

Běhové prostředí (Framework)

.NET Framework poskytuje programům nejrůznější služby (jde o obdobu Java Virtual Machine - JVM). V zásadě jde o ty služby, které programům poskytují jednak operační systémy, jednak o ty, které poskytují knihovny klasických programovacích jazyků. Výhodou je ale to, že tyto služby jsou jednotné – nezáleží na operačním systému ani na použitém jazyku. .NET je běžně používán na Windows – v jejich novějších verzích je přímo jejich součástí (i když často nejde o poslední verze .NET). Ovšem existuje i free implementace .NET (není vytvořena Microsoftem) – tzv. **Mono**, která existuje (s určitými omezeními) i pro Unix a Linux. Takže tentýž program může pracovat beze změny na Windows i Linuxu.

Některé ze služeb .NET jsou realizovány samostatným kódem, jiné tvoří jednotné rozhraní ke službám operačního systému. .NET tedy poskytuje správu procesů a threadů, správu paměti, podporu ladění …

Dále je uvedena základní struktura .NET Frameworku. Rozhodně nejde o kompletní přehled - .NET obsahuje tisíce tříd, struktur, interfaces … (o metodách, vlastnostech a událostech ani nemluvě). Jde jenom o základní přehled, co asi .NET obsahuje (a samozřejmě autor každé publikace o .NET zahrnuje ty části a detaily, které považuje za nejdůležitější).

Vlastní aplikace						**Názvy vrstev:**
Console	WinForms	WebForms (ASP.NET)		XML Web Services	…	**User/program interface**
ADO .NET (práce s databázemi)						
Math (sin,cos,…)	Stringy	Datum/čas	Kryptografie	XML	…	**Framework Class Library**
JIT compilátory	Správa procesů a threadů	Správa paměti (GC)	Security	Datové typy (CTS)	…	**CLR**
Hostitelský operační systém (většinou Windows)						

Tato tabulka představuje strukturu .NET Frameworku zhruba pro verzi 2.0. Od té doby byla přidána mnohá rozšíření, z nichž nejdůležitější jsou uvedena v další podkapitole.

Vysvětlivky k tabulce:
CLR – Common Language Runtime – základní část frameworku

CTS – Common Type Systém – zabudované a definovatelné datové typy (zahrnuty přímo do základu .NET) CTS obsahuje prakticky všechny „rozumné“ datové typy. Jednotlivé jazyky, pracující na platformě .NET, z nich používají určitou podmnožinu, přičemž mohou daný typ nazývat tak, jak je v daném jazyku zvykem. Např. typ Int32 (přesně System.Int32) je v C# a C++ nazýván int, ve Visual Basic .NET Integer. (V jednom programu lze dokonce použít oboje (což není příliš rozumné): int i; int32 k;

CTS umožňuje vytvářet snadno aplikaci, jejíž jednotlivé části jsou psány v různých jazycích. Ale protože jednotlivé jazyky (i části .NET) nemusejí podporovat všechny typy a možnosti, které .NET poskytuje, byla vytvořena řada pravidel, která zajišťuje bezproblémovou spolupráci programů – **CLS** (Common Language Specification). Kompilátor přitom umí upozornit warningem na porušení pravidel CLS (pro to, aby kompilátor tyto warningy vypisoval, se používá atribut [assembly: CLSCompliant(true)]).

Další významná rozšíření .NET frameworku (popř. jazyků)

V určité době se Microsoftům zlíbilo ve slově Foundation, takže vznikly:

Windows Commnunication Foundation (WCF) – jakýsi jednotný „obal“ nad komunikačními protokoly a službami. WCF je z velké části řízeno deklarativně pomocí konfiguračních souborů, takže se programátor nemusí zabývat detaily komunikace. (V praxi to ale výhoda spíše pro administrátory. Pro programátora to zase taková výhra není: často je daleko snazší zjistit chybu v kódu než nějaký překlep v konfiguráku.)

Windows Workflow Foundation (WF) – (mělo by být WWF, ale tuto „značku“ již má někdo rezervovánu a ty velké firmy se chovají jako malé děti). WF umožňuje vytváření programů grafickým způsobem: algoritmus se „namaluje“ (na plochu se umístí např. rozhodovací blok, do vlastností se vloží podmínka – např. x > 0 – a větve Ano a Ne se spojí s ostatními částmi grafu). Z takto vzniklého „programu” se vygeneruje CIL, který je možno spustit. Tímto způsobem je možno vytvářet nejenom grafy typu vývojových diagramů (blokových schémat), ale i ve tvaru konečných automatů (propojené uzly a přechod mezi uzly na základě vstupů).

Windows Presentation Foundation (WPF) – asi nejvýznamnější ze všech „foundations“. Jde o tvorbu „tlustých“ klientů, které je opět možno vytvářet vizuálními nástroji (jako u WinForms), ale formulář (obrazovka) je uložen ve formě XML (resp. XAML), nikoli ve formě kódu používaného jazyka, jak je tomu u WinForms. Nové „formuláře“ jsou tedy přenositelné mezi různými jazyky (jde opět o „inspiraci“ u GLADE a podobných systémů). Samozřejmě je možno formuláře vytvářet (a ukládat) i bez vizuálních návrhářů a XML – čistě programově. Pro WPF pochopitelně existují třídy, které lze v programu používat. Takový způsob je ale podstatně pracnější.

Další „inspiraci“ MS získal u Flash animací. Je možno (deklarativně) určit, že nějaký prvek na obrazovce se bude pohybovat po určité křivce, proměnnou rychlostí danou zvoleným vzorcem, přitom bude měnit barvu a vydávat nějaké zvuky. Tyto údaje se opět uloží v XAML. (Aby bylo možno oddělit práci návrhářů (grafiků) a programátorů, dodávají MS ještě další „vývojové prostředí“ – Blend for Visual Studio. V něm se provádí grafický a animační návrh aplikace, zatímco programování se provádí ve Visual Studiu. Samozřejmě VS umí zpracovat výstup Blendu a naopak.)

Silverlight – původně měl vhodnější název WPF/e (e je od everywhere), což ale nebylo dost vznešené. Jde o „ořezanou“ verzi WPF (např. dvourozměrná místo trojrozměrné grafiky) a .NET frameworku tak, aby se vešla do několika megabyte. Podstatné je ale to, že lze spouštět v rámci webové stránky (browser tvoří okno aplikace) malým úvodním JavaScriptem. Ten stáhne a nainstaluje vlastní Silverlight (není-li již instalován) a potom v něm spustí požadovanou Silverlight aplikaci. Jde o v podstatě standardní .NET programování v okně browseru (nejde tedy o web aplikaci). Vzhledem k tomu, že existuje i verze pro Mac a další pro Linux (zvaná **Moonlight**, vytvořená autory Mono), jde skutečně o multiplatformní programování. Na druhé straně je třeba uvést, že Silverlight má mnoho omezení, daných jeho malým rozsahem a hlavně bezpečnostními hledisky. A konečně je vhodné zdůraznit, že Silverlight lze používat i jiným způsobem než je výše uvedeno. Lze např. vkládat malé Silverlight prográmky do standardní webové stránky (ve stylu ohyzdných Flash reklam), popřípadě používat Silverlight pro tvorbu samostatných aplikací, které se nespouštějí v rámci browseru.

ASP.NET MVC – web aplikace se nejčastěji vytvářejí pomocí jedné ze tří technologií: jednak vkládáním skriptu mezi HTML tagy (typicky PHP, repektive jemu velmi podobné původní ASP). Potom přišel Microsoft s tím, že zavedl technologii ASP.NET, která (zhruba) simuluje na webu tvorbu „tlustých“ klientů. Časem se ale zjistilo, že (hlavně pro rozsáhlé aplikace) přináší ASP.NET určité problémy – zejména přenášení velkých viewstates a horší možnosti ladění programů. Mezitím ostatní svět (Ruby, Python, PHP, …) přešel na architekturu MVC (Model, View, Controller) – rozdělení programu na části, které realizovaly vlastní logiku aplikace (Model), definovaly zobrazení (View) a zpracovávaly vstupy od uživatele (controller). MS se tedy opět „inspiroval“ a vytvořil na základě původního ASP.NET technologii MVC. Tvrdí však, že bude i nadále podporovat původní ASP.NET (což říkal již mnohokrát u různých starších technologií).

Web API – v rámci práce na ASP.NET MVC Microsoft vytvořil i další technologii – možnost tvorby aplikací, jejichž rozhraní (API) tvoří v podstatě HTTP protokol. (Ostatní svět to nazývá REST – popř. RESTful – technologií. Ovšem nebyl by to ani Microsoft, kdyby do standardních a všeobecně přijatých pojmů nevnášel zmatek.)

Zajímavé je ale to, že MS zřejmě v příští verzi sjednotí Web API a ASP.NET MVC (což je dost logické – navíc již dnes mají tyto programy podobnou strukturu). Nebo spíš, že založí MVC na Web API.
A zvlášť pozoruhodný je fakt, že plánují oddělení Web API od úděsného IIS, takže by fungovalo samostatně. A dokonce Web API chtějí vyvinout pro různé platformy a snad i nabízet jako free software! (Tyto informace jsou ovšem nezaručené a předběžné.)

Další rozšíření souvisejí se spoluprací programu s databází. S databázemi se v programu pracuje poněkud „offline“: program vytvoří řetězec SQL příkazu (někdy parametrického příkazu, stored procedury apod.), ten pošle na databázi a výsledky (pokud je daný příkaz vrací) zpracuje. To ovšem u fanatických vyznavačů objektového programování vyvolává téměř záchvaty: „Paní, slyšela jste - dyť ono to SQL je řetězec, ono to nemá metody, vlastnosti, ani události – s tím se prostě nedá žít!“

Takže se hledaly způsoby, jak zaintegrovat práci s databází do jazyků a zejména do objektového přístupu. Platforma .NET sice od začátku obsahuje technologii ADO.NET, která používá objekty pro přístup k databázím, ale snaží se ve svých objektech namodelovat základní databázové struktury (tabulky, views, relace, …) a to opět bylo považováno za velmi fuj (i když jde o značně efektivní přístup).

U Microsoftu – kde zřejmě vzhledem k jeho velikosti vzniká podobná situace jako ve státní správě (totiž to, že pravice neví, co dělá levice a tedy se pracuje velmi neefektivně) – vznikly dvě naprosto odlišné technologie pro práci s databázemi – LINQ a Entity Framework. Dlouhou dobu panoval absolutní zmatek – nebylo vůbec jasné, co vlastně MS bude používat pro práci s databázemi. Teprve později se ukázalo, že se přiklonil k daleko těžkopádnějšímu Entity Frameworku (typické pro MS), ale LINQ ponechal jako univerzální dotazovací prostředek, který lze používat pro konstrukci dotazů – i v rámci Entity Frameworku (tzv. LINQ to Entities).

LINQ (Language INtegrated Query) – je univerzální dotazovací „jazyk“. Je součástí (rozšířením) existujícího jazyka (např. C#) a hodně vychází ze syntaxe SQL. (Pravděpodobně jde o jeden z velmi mála případů originálního výtvoru firmy Microsoft.) Za hlavní přínos LINQ považuji to, že v programovacích jazycích umožňuje zadávat „co požadujeme“, nikoli krok po kroku popisovat, jak toho dosáhnout (což se dělá v klasických programovacích jazycích). Je to podobné jako u SQL: tam také zadáváme (v klauzuli WHERE), že chceme např. výpis všeho zboží, kde počet kusů na skladě je větší než sto. Ale nikde se neříká, zda se má vyhledání provést postupným procházením celé tabulky, použitím indexu či jinak. O tom rozhoduje až optimalizátor v databázi. LINQ má několik podvariant (včetně výše uvedené LINQ to Entities):

- LINQ to Objects – umožňuje dotazování do kolekcí objektů v RAM
- LINQ to XML – dotazování XML dat
- LINQ to ADO.NET - dotazování do objektových struktur ADO.NET – méně používané
- LINQ to SQL – dotazování do databáze (původní cíl LINQ, docela dobře udělané; přednost získal ale Entity Framework)

LINQ existuje i v paralelní verzi – PLINQ, která umožňuje u vícejádrových (víceprocesorových) systémů rozdělit vykonávání dotazu na více částí, které se paralelně provádějí na různých jádrech (procesorech) – podobně jako ve velkých databázových systémech.

Entity Framework - (dále EF) patří do takzvaných ORM (Object-Relational Mapping) systémů. Byl „inspirován“ NHibernate a dalšími ORM systémy. V principu se v programu pracuje pouze s objekty, které je ale možno načítat a ukládat z/do odpovídajích databázových struktur. V programu se vůbec nemusí pracovat s SQL jazykem – potřebné SQL příkazy generuje EF sám (což ale může vést k neefektivitám – i když se situace poněkud zlepšuje). EF lze používat třemi hlavními způsoby:

- Database First - vychází se z existující databáze. Nástroje EF z ní „vytáhnou“ používané struktury a na jejich základě se vytvoří tzv. Entity Model a odpovídající objekty (vybavené pro práci s databází)
- Model First – vytvoří se (většinou vizuálními nástroji) model používaných struktur (objektů a jejich vazeb) a na jeho základě lze vygenerovat jednak databázi a jednak kód pro objekty
- Code First – vytvoří se standardní třídy, tzv. kontextová třída (práce s DB) a definuje se mapování tříd (resp. vlastností) na databázové tabulky. Je možno i generovat SQL pro vytvoření databáze.

Struktura překládaných programů (assembly)

Překladem zdrojového kódu programu, určeného pro platformu .NET, vzniká tzv. assembly. (Nebudu si hrát na obrozence a nazývat to sestavením, montáží ani břinkotlapkošlapkou.) Assembly nemusí být tvořena jedním souborem, ale může se skládat z několika částí – tzv. modulů. (Toto se ale používá spíš v situaci, kdy program překládáme řádkovým překladačem csc a používáme assembly linker al.)

Assembly se může skládat z následujících částí (které kromě manifestu jsou nepovinné):

- vlastní **kód** (CIL) – nemusí být, jde-li např. jenom o deklarace
- assembly **manifest**, který obsahuje:
  - jméno assembly
  - číslo verze
  - jazyk (CZ, EN, …) – nikoli programovací jazyk
  - seznam odkazů (referencí) na jiné assembly, které tato assembly používá (důležité)
  - seznam souborů, které assembly tvoří …
- assembly **metadata** – popis struktury assembly:
  - třídy, struktury, … definované v assembly
  - jejich metody
  - parametry – typy a pořadí – a vracené hodnoty metod …
- **resources** – externí nebo interní „zdroje“ programu – např. texty, obrázky, … v různých jazycích

Assembly mohou být tzv. **private** (běžný výstup kompilace) nebo **strong name** – digitálně podepsané. (Profesionální aplikace by měly být vždy strong name – už jenom z toho důvodu, že při zavádění porušené strong name assembly (disková chyba, virus apod.) je ohlášena chyba a program se nespustí.)

Assembly může být vytvořena jako .exe (obvykle tehdy, když s ní přímo komunikuje uživatel), nebo jako .dll (je-li určena k volání jinými assembly). Zdůrazňuji, že nejde o klasická .exe a .dll – neobsahují např. stojový kód, ale CIL (viz výše) – přesněji obsahují assembly. Těžkou chybou Microsoftu je to, že pro tyto .NET aplikace nezvolil jinou koncovku, když i pro obyčejný Office po změně struktury požívá .xls->.xlsx …

Instalace assembly: na rozdíl od katastrofálního způsobu instalace starších Windows aplikací stačí assembly .NET aplikace nakopírovat do zvoleného adresáře (s případnými dalšími používanými soubory) – tedy žádné zápisy do registry a jiné obludnosti. Musí jít ovšem o čistou .NET aplikaci, která nepoužívá unmanaged kód. (Pro aplikaci, skládající se z více assembly, si však přečtěte také další bod.) Samozřejmě na Windows, kde je aplikace instalována, musí být nainstalován .NET Framework (požadované verze).

Tento způsob instalace se nazývá **XCopy instalace** (to proto, že často je aplikace uložena ve struktuře vnořených adresářů a ty je potřeba nakopírovat na cílový počítač – buď pomocí Windows Exploreru, nebo řádkovým příkazem XCopy). Je samozřejmě možné vytvořit kompletní instalačky (soubory .exe nebo .msi), ale to je spíš věc jednoduchosti a pohodlí instalace pro uživatele – není to nezbytnost, jakou byla instalace u starších Windows technologií. Visual Studio poskytuje možnost jak kompletní tvorby instalačního programu, tak (pro jednoduché aplikace) tzv. **ClickOnce instalaci** (kdy uživatel provádí instalaci jedním kliknutím na hyperlink nebo soubor).

Aplikace, složená z více assembly

Při vytváření aplikace, složené z více assembly, musí být především v té assembly, která volá metody tříd, definovaných ve druhé assembly (popř. používá jejich vlastnosti apod.) definovány reference na „volanou“ assembly (viz assembly manifest. Volaná assembly se často nazývá komponenta.) Potom stačí obě assembly (volanou i volající) nakopírovat do stejného adresáře a aplikace bude fungovat. (.NET dovoluje i jiné možnosti – např. kopírovat volanou do speciálně pojmenovaného podadresáře (viz help), ale toto je nejjednodušší varianta.)

Kromě toho je možné umístit private assembly (viz výše) do libovolného **pod**adresáře toho adresáře, v němž je uložena volající assembly. Ovšem v tom případě je třeba použít tzv. **konfigurační soubor**, ve kterém se určí, kde se má volaná assembly najít.

Pro strong name assembly (viz výše) je možné umístění kamkoli – třeba i na jiný disk (rozhodně nemusí být v podadresáři adresáře volající assembly). Ale i v tomto případě je třeba vytvořit konfigurační soubor – pro tuto situaci navíc poměrně složitější.

Global Assembly Cache (GAC)

Výše uvedená řešení jsou dost nepraktická např. pro assembly, sdílené více aplikacemi (a pro private assembly nepoužitelná) – je totiž nutné vytvářet konfigurační soubory. Totéž je nepříjemné pro dodavatele „knihoven“ apod. Proto byla definována **GAC** (**Global Assembly Cache**). GAC je adresář zvaný assembly, standardně umístěný pod adresářem Windows.

Je-li ve volající assembly použita třída, její metoda apod., která v dané assembly není definována, je jedno z prvních míst (vedle aktuálního adresáře a míst, určených konfiguračním souborem – přesné pořadí viz help), kde je volaná assembly hledána, právě GAC. Tedy je-li assembly (obvykle typu .dll – komponenta) umístěna v GAC, bude automaticky nalezena bez jakýchkoli konfiguračních souborů či kopírování do adresáře, kde je umístěna volající assembly. (Např. i komponenty vlastního .NET Frameworku jsou uloženy do GAC.)

Ovšem pro GAC platí některá pravidla: **vkládat** assembly do ní (např. pomocí řádkové utility gacutil nebo prostřednictvím instalačního programu) může **jenom uživatel s administrátorskými právy**. Dále **assembly**, která je do GAC kopírována, **musí být strong name** (kvůli bezpečnosti a efektivitě – digitální podpis se např. kontroluje jenom při ukládání assembly do GAC, nikoli při jejím spouštění. GAC se považuje za dostatečně chráněnou.)

Soustředěním sdílených assembly (komponent) do jednoho adresáře však hrozí nebezpečí, známé z klasických Windows jako „dll hell“. Jde o to, že pokud se nahradí nějaké .dll novější verzí, mohou aplikace, které používaly starší verzi, začít „padat“. Přitom třeba nová verze napravila některé chyby starší, ale volající programy mohly právě tyto chyby nebo nepřesnosti starší verze (třeba nevědomky) využívat, ale s novou – byť lepší – verzí padají.

Aby se tomuto problému předešlo, jsou **assembly v GAC odlišovány dle jména, verze,** jazyka (cz, en, …) – tzv. **culture** a **hash z public key** (musejí být digitálně podepsané – viz výše). V GAC tedy mohou být různé verze – starší a novější – a nepřepíší se. Přitom v referenci volané assembly je uvedena verze volané assembly (ta, se kterou byla volající kompilována), takže si i při více verzích v GAC volající vybere „tu svoji“. Je ovšem možné vytvořit konfigurační soubor, který přinutí volající assembly používat jinou verzi (je-li např. vhodnější).

GAC se jeví jako „plochá struktura“, ve které jsou assembly rozlišovány právě podle čtyř výše uvedených atributů, zobrazených ve čtyřech sloupcích. Něco takového ale není v souborovém systému možné. Jde jenom o rozšíření Windows Exploreru („Průzkumníka“), který GAC takto zobrazuje. Ve skutečnosti je GAC tvořen sadou podadresářů, jejichž jména obsahují verzi, culture a hash z public key a to právě umožňuje ukládat do GAC různé verze assembly se stejnými jmény.

Namespaces

Díku tomu, že je v jedné assembly snadno možné používat např. třídy z jiné assembly (jak je uvedeno výše, stačí ve „volající“ assembly uvést reference na „volanou“ a obě např. nakopírovat do stejného adresáře), vzniká problém s **kolizí stejných jmen v různých assembly**. Příklad: budu psát aplikaci, která bude využívat dvě komponenty – jednu pro účetnictví (např. Ucto.dll) a druhou pro skladové hospodářství (Sklad.dll). Ale v obou bude stejnojmenná třída Položka. Ovšem jedna bude účetní položka, druhá skladová, takže se mohou lišit ve vlastnostech, metodách i událostech, které vyvolávají. Tím vznikne problém při překladu. Snadno jej vyřeším, jestliže jsem komponenty psal sám. Potom prostě jednu (nebo obě třídy) přejmenuji. Toto ovšem nelze provést, pokud mám (např. koupené) komponenty bez zdrojových kódů. Právě k řešení tohoto problému jsou určeny namespaces.

Abstraktně je **namespace** prostor, ve kterém jsou **všechna jména různá**. Ale ve dvou různých namespace mohou být použita stejná jména. Odpovídá to adresářům – v jednom adresáři nemohou být soubory stejných jmen, ve dvou různých adresářích ano.

Technicky jsou **namespaces řešeny** pomocí tzv. **prefixů**. Takže pro Ucto.dll zvolíme např. prefix ucto, pro Sklad.dll prefix skl. Potom budeme psát ucto.Polozka nebo skl.Polozka a kolize nenastane (prefixy se liší, je jasné, o kterou položku jde). Poznámka pro ty, kteří znají XML a jeho namespaces: abstraktně jde o totéž, ale technicky se realizace velmi liší: v .NET prefix již jednoznačně určuje namespace – nepřísluší k němu již nic dalšího!

Okamžitě se ale naskytne námitka, že zrovna tak se může vyskytnout i stejný prefix. Proto doporučuje MS používat jako prefix webové jméno firmy obráceně – např. cz.nasefirma.www.Polozka. A pokud by firma měla dejme tomu více poboček, lze použít třeba cz.nasefirma.praha.Polozka, cz.nasefirma.brno.Polozka atd. Podobně by se řešila situace, kdy potřebujeme různé namespaces pro více aplikací. (Samozřejmě takovéto namespaces mají smysl jen tehdy, když danou komponentu určíme k použití ve více programech – tedy ne ty, které např. nejsou sdílitené (.exe)).

Assembly a namespaces. Vztah mezi assembly a namespaces je v postatě libovolný. Mohou nastat tyto varianty (namespace se definuje pomocí složených závorek, kdekoli je potřeba – namespace xxx { … }):

- **celá assembly je jeden namespace** (všechen kód – kromě directiv using – je vnořen do obklopujícího namespace). Takto vytváří strukturu programu např. Visual Studio – celý kód vnoří do namespaspace, zvaného podle názvu projektu (projekt Test, namespace Test). Obecně assembly nemusí mít obklopující namespace (použije se implicitní), ale velmi se obklopující namespace (nejvyšší) doporučuje.
- **jedna assembly obsahuje více namespace** – toho se snadno dosáhne díky tomu, že začátek a konec assembly je určen závorkami { a }. Takže část kódu assembly se vnoří do bloku jednoho namespace, další část do druhého atd. Při tomto přístupu je poněkud obtížnější programování, protože např. voláme-li v jednom namespace metodu definovanou ve druhém, musíme uvést prefix druhého. Někdy to ale může být vhodné.
- **jeden namespace obsahuje více assembly** – opět velmi jednoduché, protože stačí v každé z příslušných assembly nadefinovat stejný obklopující namespace

Vnořené namespaces. Definice namespaces je možno vnořovat (díky závorkové struktuře):

```
namespace vnejsi{
   // …
   namespace stredni{
      // …
      namespace vnitrni{
         // …
         class Polozka{
            // …
         }
         // …
      }
      // …
   }
   // …
}
```

Potom přístup zvnějšku ke třídě Položka bude: **vnejsi.stredni.vnitrni**.Polozka

Poznámky: tečka ve jménu namespace nemusejí znamenat vnořený namespace. Tečka je prostě přípustný znak ve jménu namespace – viz výše cz.nasefirma.www. Dále: vnořené namespaces se používají hlavně v rozsáhlých assembly, aby se strukturoval a zpřehlednil jejich obsah (třeba v komponentách .NET).

Direktiva using. Aby nebylo nutné psát vždy znovu dlouhé prefixy při používání např. tříd z jiných namespaces, je možno použít direktivu (direktivy) using. Píší se na začátku kódu a fungují trochu jako path v adresářích. Tedy: napíši-li na začátku kódu **using cz.nasefirma.www;** lze potom dále psát pouze Polozka místo cz.nasefirma.www.Polozka. Takto samozřejmě nelze postupovat v případě kolize jmen (ovšem using dovoluje také jména přejmenovat – viz dále).

Pozor ovšem na to, že using nepůsobí rekursivně: napíšeme-li např. using System; nebude překladač (zřejmě z důvodů efektivity) při hledání jména prohledávat vnořené namespaces. Musíme tedy použít using System.Data.SqlClient; pokud chceme pracovat např. se třídou SqlConnection z tohoto namespace.

Ještě jedno upozornění: using je v je C# také příkaz, používaný v kódu a je to něco zcela jiného než directiva using, používaná na začátku kódu.

Použití **using** pro **přejmenování**: např. v našem prvním příkladu pro namespaces bychom mohli napsat:

```
using sPolozka = skl.Polozka;
using uPolozka = ucto.Polozka;
```

Potom v kódu můžeme používat nové názvy jako sPolozka místo skl.Polozka. To může být výhodné např. pro řešení kolizí jmen pro případ dlouhých názvů namespaces.

Přejmenovávat lze také vlastní namespaces. Např. můžeme použít (z předchozího příkladu):

```
using slozeny = vnejsi.stredni.vnitrni;
```

Vývojové prostředky pro C#

Vývoj můžeme provádět pomocí editoru typu Notepad a **řádkových kompilátorů a linkerů csc a al** (oboje zadarmo stažitelné z webu MS). Takový vývoj je ale pro vývojáře značně náročný a používá se spíš pro závěrečné kompilace aplikací, které potřebujeme kompilovat nějakým speciálním způsobem.

Dále můžeme použít **MonoDevelop** (www.monodevelop.org), které je free a multiplatformní (použitelné na Windows i Linuxu), na druhé straně neobsahuje všechny (hlavně nadstavbové) možnosti Visual Studia a je vždy o něco pozadu vůči poslední verzi Visual Studia. Pokud bychom chtěli používat C# programy na Linuxu (ať už vyvinuté v MonoDevelop nebo Visual Studiu), potřebujeme na Linuxu napřed nainstalovat platformu .NET. Tu nalezneme jako free aplikaci, zvanou **Mono** (www.mono-project.com).

Visual Studio je hlavní vývojové prostředí firmy Microsoft, které budeme používat i v tomto předmětu. V současnosti je jeho poslední verzí Visual Studio 2013. Vyskytuje se v různých podverzích – od Visual Studio **Expresss**, které je free (poněkud ořezané, ale dost kompletní) až po **Ultimate** (resp. Enterprise v některých verzích – stálé změny názvů přispívají k většímu zmatku). Ultimate verze obsahuje např. i nástroje pro systémové architekty (vedoucí softwarových týmů, kteří již zapomněli programovat) a které jde daleko za hranice všedních cen. Pro většinu vývoje stačí verze Visual Studio **Proffessional**, které však také není zvláště levné, takže některé (i profesionální) vývojářské týmy používají verzi Express (je otázka, je-li to právně v pořádku).

Programy v C# potřebují nainstalovanou platformu .NET (jak bylo uvedeno na začátku). Většina novějších Windows .NET obsahuje, ale obvykle ne v poslední verzi (ta je dnes 4.5.1). Novější verze Visual Studia ale dovolují nastavit verzi .NET, pro kterou je program kompilován. Doporučuji být zde konzervativní (pokud v programu nevyužíváte výstřelky poslední verze .NET) a nastavit nějakou nižší verzi – ušetříte si instalaci nového .NET u zákazníků.

Programovací jazyky podporované Visual Studiem
Verze Expres slouží vždy pro jednu oblast. Potřebujeme-li pracovat ve více oblastech, musíme stáhnout několik verzí expressu: Visual Studio Express C#, Visual Studio Express Visual Basic, Visual Studio Express Web Developer, … Vyšší verze (Proffessional, Ultimate apod.) obsahují standardně podporu těchto jazyků:

- **C#** - probíraný na tomto předmětu; nesmírně silná „inspirace“ Javou
- **C++** - standardní C++ s doplňky pro práci na .NET; na rozdíl od ostatních jazyků dovoluje generovat unmanaged kód (klasická .exe a .dll) - důležité pro tvorbu driverů, základů operačních systémů, …
- **Visual Basic .NET** – plně objektová verze jazyka Visual Basic (i když s poněkud nestandardní terminologií). Je silně nekompatibilní s před-netovskými verzemi (Visual Basic 6.0 resp. Visual Basic for Applications – VBA), dodnes používanými při programováni Office (Word, Excel, …)
- **F#** - funkcionální jazyk (ne úplně „funkcionálně čistý“)
- **JScript** – MS verze JavaScriptu. Zatímco v ostatních jazycích lze vytvářet kompletní aplikace, slouží JScript hlavně pro skriptování ve webových stránkách (často generované nástroji Visual Studia)

Kromě těchto „standardních“ jazyků existují další rozšíření Visual Studia – některé přímo od Microsoftu, jiné vytvářené třetími stranami. Např.

- **TypeScript** – typovaná verze JavaScriptu, „překládaná“ do „čistého“ JavaScriptu
- **IronPython** – verze Pythonu pro .NET
- **PHP Tools for Visual Studio** – vývojové prostředí pro tento PHP (placené)
- …