

**Nové technologie  
programování nejen na Internetu**

**Microsoft .NET**

**&**

**Java 2, Enterprise Edition**

**Tomáš Matoušek**

**2002**

# 1 Obsah

1	Obsah.....	2
2	Úvod.....	3
3	Microsoft .NET.....	6
3.1	.NET Framework.....	6
3.1.1	Common Language Runtime.....	6
3.1.2	Assemblies.....	8
3.1.3	Atributy a metadata.....	9
3.1.4	Class Library .....	9
3.2	Desktopové aplikace .....	10
3.3	Client-side aplikace na Internetu .....	10
3.4	Server-side aplikace na Internetu .....	12
3.4.1	ASP.NET.....	12
3.4.2	Práce s web stránkami: Web Forms, Web Controls .....	12
3.4.3	Web Services.....	15
4	Java 2, Enterprise Edition .....	18
4.1	Client-side aplikace na Internetu .....	18
4.2	Server-side aplikace na Internetu .....	19
4.2.1	Servlets & Java Server Pages .....	19
4.2.2	Enterprise JavaBeans .....	19
5	Další informace.....	21

## 2 Úvod

Na Internetu lze dnes najít mnoho informací. Ty jsou publikovány především v jazyku HTML a prohlíženy prohlížeči, kteří tento jazyk reprezentují v grafické podobě. Komunikace mezi servery poskytujícími informace a klienty (prohlížeči) probíhá na nejvyšší úrovni pomocí protokolu HTTP, jenž umožňuje přenášet hypertextové stránky. Není to samozřejmě jediná možnost komunikace, ale ostatními se dále zabývat nebudeme.

Možnosti stránek v jazyce HTML se za dobu své existence rychle zvětšovaly. Dříve byly stránky statické a textové. Dokument byl tvořen několika soubory obsahujícími prostý text doplněný možnostmi vzájemných odkazů mezi těmito soubory či jinými dokumenty na Internetu. Postupem času jazyk HTML prodělal vývoj. Dnes je možno do dokumentů vkládat obrázky, sekvence filmů i zvuky, přidávat k textu i tzv. kaskádové styly, jež umožňují navrhovat pokročilý design stránek, či stránky oživit skriptovacím jazykem. Ten je interpretován prohlížečem a umožňuje dynamickou manipulaci s objekty v dokumentu. Dále je možné, aby prohlížeč posílal na server data zadaná uživatelem do tzv. formulářů a aby server předal tato data programům, které na něm běží. Tyto programy podle přijatých dat vygenerují stránku, jenž se má prohlížeči poslat jako odpověď. Toto je umožněno díky schopnosti severu generovat stránky programově.

Na web serveru, který prezentuje stránky v jazyce HTML a posílá je klientovi, je možno po dotazu klienta na stránku spustit buďto přímo program resp. skript (CGI), jehož výstupem je požadovaná stránka, nebo program, který přečte již existující stránku a pozmění ji (např. PHP, ASP). Využijeme-li druhou uvedenou možnost, můžeme do stránek psaných v HTML vepisovat kousky skriptů v odpovídajícím skriptovacím jazyce (PHP, ASP), které uzavřeme do „závorek“, např. `<? kód ?>`, `<?php kód ?>`, `<% kód %>`, apod. Obsah těchto závorek (tj. kód) bude při dotazu klienta na stránku rozparsován, interpretován a případně nahrazen svým výstupem. Soubory vzniklé po tomto nahrazení jsou následně zaslány klientovi, který v principu nepozná, jaké části se generovaly nově a jaké již byly uloženy v původním souboru.

Tato metoda se používá např. při prezentaci dat uložených v databázi. Do stránky, která bude sloužit pro zobrazení dat nějaké tabulky, vložíme kód, jenž se připojí na databázový server, provede dotaz pomocí jazyka SQL, a získané výsledky zformátuje např. do tabulky v HTML a vloží je místo sebe.

Metoda nahrazování kódu interpretovaného na serveru je dnes velmi rozšířena. Způsob vývoje stránek však zdaleka není ideální a zdrojové kódy těchto stránek jsou mnohdy dosti nepřehledné. Je to způsobeno především častým proplétáním kódu jednotlivých použitých jazyků (někteří autoři používají pojmu *spagetti code*). Vždyť ve zdrojovém souboru stránky prezentující data databáze je často třeba použít i čtyři jazyky: PHP/ASP, SQL, HTML + CSS a JavaScript! Cílem by měl být ale kód, ve kterém se lze snadno orientovat. Toho lze jistě dosáhnout tím, že se tvůrci těchto stránek při jejich psaní zaměří nejen na vlastní obsah a funkčnost, ale také na čistotu, eleganci a znouvopoužitelnost kódu. Tím jsou ale často příliš odkláněni od podstaty problému, který řeší. Jsou nuceni zabývat se relativně nepodstatnými detaily, se kterými se v jiných programovacích jazycích jako je např. Java nesetkají. Z vlastní zkušenosti mohu potvrdit, že snaha o čistotu a znouvopoužitelnost kódu je někdy vykoupena nezanedbatelnými časovými ztrátami.

Podstatnou nevýhodou je také určitá nemožnost vzájemné spolupráce, má-li stránky psát skupina vývojářů. Pokud tvoří aplikaci prezentující data databáze a ne každý vývojář ovládá grafický design, HTML,

programování skriptů, SQL a server-side programování např. v PHP, není prakticky možné spolupracovat na jedné stránkách. Kvůli efektu spaghetti code se totiž vše prolíná se vším a většinou nic nelze bez obtíží oddělit zvlášť. Dále popisované technologie umožní znatelné zlepšení.

Od stránek se také čím dále tím více očekává, že se budou chovat jako standardní desktopové aplikace, což je do značné míry rozumný požadavek. Hlavním rozdílem mezi aplikací postavenou na protokolu HTTP a jazyku HTML oproti grafické aplikaci běžící přímo na operačním systému je problematičnost webové aplikace uchovávat si stav, ve kterém se nachází. Uvedu praktický příklad.

Nechť máme databázi studentů středních škol. U každého studenta udržujeme kromě jeho osobních údajů i identifikátor školy, na které studuje. Mějme pro jednoduchost stránku, na které je možné ve formuláři upravovat informace o zvoleném studentovi. Jelikož škol je mnoho, nechceme je všechny načítat při zobrazení této stránky. Pokud tedy chceme studentovi změnit školu, musí se nám v prohlížeči zobrazit seznam těchto škol a to do stejného okna prohlížeče, kde byly informace o studentovi (toto není jediná možnost, problém lze obejít jistými triky<sup>1</sup>). Poté, co si vybereme některou ze škol, je opět načtena původní stránka studenta s vyplněnou kolonkou „škola“. Ostatní údaje o studentovi musí být také staženy ze serveru. Abychom neztratily informace, které uživatel mohl již do formuláře doplnit, musíme při odeslání požadavku na stránku se seznamem škol na server poslat i tato data. Ta lze na serveru uložit a poslat zpět při návratu na stránku.

Všechny operace uvedené v příkladu lze naprogramovat pomocí stávajících prostředků na straně serveru (PHP, ASP) nebo na straně klienta programováním v JavaScriptu (zmíněné triky). Mohlo by se zdát, že je tím problém vyřešen. Řešení existuje, ale je s ním spojena opět ona režie, které pouze trápí programátora a znepráhledňuje zdrojový kód stránek. Dalším rozumným požadavkem by mohla být možnost otvírat stránku pro výběr školy nejen ze stránky studenta, ale také ze stránek jiných (např. učitele apod.). Pak je třeba do stránek vložit složitější kód zajišťující univerzalitu stránky. Přitom se tento problém dá vyřešit tak, že se část potřebné funkcionality zakomponuje přímo do web serveru. Tak je to také účinně při použití technologií popisovaných dále.

Možnosti Internetu se však nezužují jen na generování a zobrazování dokumentů či používání webovských aplikací jako front-end k databázovým systémům. Uvedme další praktický příklad. Budeme vyvíjet aplikaci, které bude zaměřena na nějakou finanční problematiku. Chceme nyní, aby aplikace znala aktuální kurzy měn podle jisté banky. Jak toho dosáhnout? Možným řešením bez použití nových technologií je umístit nějaký CGI skript či PHP/ASP stránku na server banky a odpovídat na dotazy klientů formou textového řetězce. Formát tohoto textu by definoval programátor banky a popsal by ho na webovských stránkách banky. Pak bychom se mohli, jako programátoři klientské aplikace, podívat na tyto stránky a implementovat poskytnuté rozhraní do naší aplikace. Takovéto řešení bude však nestandardní a každá taková služba bude mít

---

<sup>1</sup> Jeden z možných triků, který však využívá rozšíření poskytnutá Microsoft Internet Explorerem, spočívá v použití metody `window.showModalDialog`, jímž zobrazíme stránku pro výběr škol v novém okně prohlížeče, avšak IE nedovolí přístup do okna s původní stránkou studenta. Navíc, což je nejdůležitější, umožní předání dat mezi těmito dvěma okny pomocí proměnných `window.dialogArguments` a `window.returnValue` bez účasti serveru. Toto je asi nejelegantnější řešení vůbec (nemusí se ani žádná data ukládat na server a stahovat zpět ke klientovi), pokud můžeme použít rozšíření IE. Chceme-li, aby stránky splňovaly standardy W3C, lze nepřítomnost zmíněné metody obejít pomocí tagu `IFRAME`, avšak výsledek nebude již tak elegantní.

pravděpodobně jiné rozhraní, jiné formáty dat apod. Vyvíjet aplikaci, která využívá více zdrojů informací bude problematické už jen kvůli těmto odlišnostem. Právě standardizace komunikace mezi službami poskytovanými na Internetu je jedním z hlavních cílů nových technologií, jimiž se budeme zabývat.

Takovýchto dílčích problémů lze najít více (např. cachování dat, komunikace s databázovými servery, atd.), což vedlo softwarové architektury ke snaze navrhnout jejich efektivní a co nejjednodušší řešení. Významnými výsledky této snahy jsou architektury **Microsoft .NET** (dále .NET) a **Java 2, Enterprise Edition** (dále J2EE) navržená firmou Sun. Ty jsou také tématem dalšího textu, ve kterém se zaměříme především na Microsoft .NET. Mnohé z toho, co tyto architektury přináší lze naprogramovat i bez jejich použití. Avšak, jak je naznačeno v předvedených příkladech, je třeba více práce a úsilí, výsledný kód bude méně přehledný a vzniklá řešení často nebudou schopna jednoduše komunikovat s ostatními aplikacemi.

Samotné .NET i J2EE se skládá z mnoha myšlenek a technologií a není možné v tomto prostoru rozebrat vše. Zaměříme se jen na jejich základní prvky a na některé aspekty programování na Internetu (především na řešení zmiňovaných nesnází). Jak sama firma Microsoft přiznává, mnohé myšlenky z .NET jsou převzaty z existujících systémů (a také z J2EE). Příkladem může být nový jazyk C#, jež je velmi podobný jazyku Java firmy Sun. Přesto jako první prozkoumáme architekturu .NET, kterou poté porovnáme s J2EE. Ačkoliv je firma Microsoft známa nestandardností svých produktů, je na jejím novém produktu již poznat jistá snaha o standardizaci. Na Internetu totiž nelze bez standardizace uspět v širším okruhu jeho uživatelů, jelikož je velmi heterogenní a spojuje mnoho různých technologií. Obě architektury jistě mají svoji budoucnost a jsou připraveny kooperovat s již zavedenými technologiemi i spolu navzájem. Na možnost použití již existujících programových vybavení kladou obě velký důraz, aby nebyly zmařeny dosud investované prostředky.

#### **Poznámka**

V této oblasti je mnoho anglických termínů, které nemá smysl překládat do češtiny, jelikož mnohdy neexistuje ani jejich jednoduchý ekvivalent a bylo by nutné vymýšlet nové pojmy. V textu tedy bude překlad uveden jen někdy a většinou budou používány anglické pojmy.

## 3 Microsoft .NET

Microsoft .NET je souhrnný název pro množinu technologií programování desktopových aplikací i aplikací pro Internet. Jejím základem je *.NET Framework*, což je prostředí pro vývoj a běh programů. Jeho relativně stručný popis uvedeme v následující části. Dále se pak zaměříme na nové možnosti programování desktopových aplikací a aplikací typu klient-server využívající možnosti Internetu.

### 3.1 .NET Framework

.NET Framework zajišťuje prostředí pro běh nového typu aplikací. Zatím je implementováno jen pod operačním systémem Windows, ale není z principu omezeno jen na tuto platformu<sup>2</sup>. Není totiž součástí operačního systému, ale jeho rozšířením. Fyzicky (v operačním systému Windows) je tvořeno sadou assemblies (viz dále), dynamicky linkovaných knihoven, služeb, programů a konfiguračních souborů, které lze do systému přidat a opět odebrat bez újmy na funkčnosti systému (bez něj pouze nepůjdou spustit aplikace pro něj napsané).

#### 3.1.1 Common Language Runtime

Chceme-li vyvíjet libovolnou aplikaci, musíme samozřejmě napsat její zdrojový kód v nějakém jazyce. V případě platformy .NET je možno zvolit mezi několika jazyky, jejichž kompilátory jsou dodávány firmou Microsoft (např. C#, Visual Basic, C++) nebo zvolit jiné jazyky, jejichž kompilátory vyvíjejí jiné firmy a které jsou určeny pro .NET (tímto směrem je platforma .NET otevřena).

Většina aplikací potřebuje ke svému běhu knihovny poskytnuté vývojovým prostředím či přímo jazykem, v němž jsou napsány. Množina těchto knihoven je tzv. *runtime* aplikace. V .NET je tímto běhovým prostředím *Common Language Runtime (CLR)*. CLR zajišťuje spuštění a ukončení programu, spravuje paměť, thready, zajišťuje bezpečnost běhu aplikací, atp. Již jeho název napovídá, že se jedná o prostředí společné všem jazykům, které lze použít pro vývoj aplikací na platformě .NET.

Když napíšeme zdroj programu ve zvoleném jazyce, kompilátor tohoto jazyka přeloží tento kód do jazyka *Microsoft Intermediate Language (MSIL)* a navíc k tomuto kódu připojí další informace – tzv. *metadata*. Takto přeloženému kódu se říká metadata řízený kód (*managed code*). O jeho provádění se stará CLR využívajíc přidaná metadata (viz dále). U některých jazyků (např. C++) je také možno zdroj programu zkompileovat přímo do nativního kódu procesoru, na kterém kompilátor běží. Pak není třeba přidávat metadata a takový kód je tudíž neřízený (*unmanaged code*). Neřízený kód nepotřebuje ke svému běhu .NET Framework, je však procesorově závislý – jde o stejný kód, který je generován standardními překladači (tedy překladači neurčenými pro .NET).

MSIL je jazyk velmi blízký assembleru, avšak narozdíl od assembleru je procesorově nezávislý a umožňuje vyšší úroveň programování. Kromě instrukcí jako jsou například instrukce realizující aritmetiku, přesuny paměti, řízení toku programu, které má každý assembler, jsou v MSIL také instrukce řízení výjimek, práce

---

<sup>2</sup> Firma Ximian, Inc. (<http://www.ximian.com>) již pracuje na projektu „Mono“ (viz <http://www.go-mono.com>), jenž má být open-source implementací .NET Framework pro Linux.

s objekty, volání virtuálních metod či přímá manipulace s prvky polí. MSIL je navržen tak, aby šel jednoduše přeložit do assembleru procesoru, kterým má být program vykonáván.

Žádné programy na platformě .NET nejsou interpretovány. Kód v MSIL musí být vždy přeložen do nativního kódu a pak vykonáván přímo procesorem. Tento překlad je realizován tzv. *just-in-time* kompilátorem (zkráceně *JITter*), jež je přímo součástí CLR. Překlad aplikace má tedy dvě fáze: (a) překlad z vyšších programovacích jazyků do MSIL a (b) překlad z MSIL do strojového kódu cílového procesoru. Fáze (a) probíhá na počítačích, kde je program vyvíjen programátory. Fáze (b) však probíhá buďto při instalaci programu na cílový počítač nebo teprve při spuštění programu na tomto počítači a má ji na starosti právě *JITter*. Jakou metodu překladu programátor zvolí záleží také na jeho požadavcích na rychlost spuštění aplikace (kompilace při spuštění větších programů trvá nezanedbatelnou dobu). Z důvodů úspory času a paměti je také možno nepřekládat celou aplikaci hned při spuštění, ale přímo při jejím běhu. Jelikož aplikace na počátku svého běhu často nevyužívá všech svých modulů a funkcí, je možné, aby *JITter* zkompiloval nejprve inicializační část aplikace a poté přidával jednotlivé metody podle toho, jak jsou v průběhu práce aplikace používány.

Aby bylo možné překládat různé jazyky do jednoho a navíc zaručit jejich bezchybnou spolupráci (mezi-jazykové volání a zachytávání výjimek, dědičnost, atd.), je třeba, aby tyto jazyky splňovaly některé základní požadavky, jež jsou zahrnuty v tzv. *Common Language Specification*. Tato specifikace obsahuje definici typů *Common Type System*, kterou musí shodně implementovat všechny jazyky, dále definici způsobu vyvolání a zachycení výjimek, způsobu přístupu k objektům, způsobu volání metod apod.

Některé jazyky v .NET umožňují více, než je vyžadováno CLS. Pak ale nelze tyto nadstandardní schopnosti užívat, pokud je třeba spolupracovat s kódem napsaným v jiném jazyce. Pokud například v jednom jazyce je použit typ, který není v jiném, nelze metodu používající onen typ volat z obou jazyků. Naopak v metodách, které jsou privátní a nelze je volat z jiných modulů, jež by mohly být programovány v jiném jazyce, lze používat všech rysů daného jazyka. Kód splňující CLS bývá označován jako *CLS-Compliant Code*. Kontrolovat platnost pravidel CLS nemusí programátor – dělá to za něj kompilátor, je-li část veřejného kódu označená atributem `CLSCompliantAttribute` (o attributech se dozvíme v části 3.1.3 Atributy a metadata).

#### **Poznámka**

Je třeba říct, že možnost použití mnoha jazyků je jistě výhodou, avšak ne všechny jazyky, které jsou předělávány tak, aby splňovaly požadavky CLS, budou po předělání vypadat stejně jako před ním. Například VB se dosti liší. Pokud původní jazyk není ani objektový (tak tomu bylo u VB), nemyslím si, že je vhodné ho násilím předělat do objektového a pak navíc do CLS kompatibilního. Navíc starší aplikace napsané v tomto jazyce v jeho nové verzi pravděpodobně nebude možné přeložit. Možnost pracovat s více jazyky by se také neměla nadměrně využívat v rámci jednoho projektu. Výhodu vidím spíše v možnosti používat různé produkty třetích stran, které mohou být programovány v různých jazycích (takových, které jejich vývojáři znají a na které jsou zvyklí) a přitom plně využít výhody společného jazykového základu, mezi něž například patří použití mechanismu výjimek tak, jako by byla celá aplikace naprogramována v jednom jazyce.

## 3.1.2 Assemblies

Každý větší program je kvůli přehlednosti nutné rozdělit do několika modulů. Ty mohou být linkerem slinkovány do jednoho výsledného souboru nebo je možné vytvořit dynamicky linkované knihovny apod. Aby bylo možno sdílet kód mezi různými programy či výrobci, vytvářejí se komponenty, jež poskytují přes své rozhraní rozmanité služby. Tato technologie se rozšířila ve Windows pod názvem COM a postupně se vyvíjela. Doposud však nebyl uspokojivě vyřešen problém verzování a instalace komponent (někdy pojmenovávaný DLL Hell). Spočívá v nepřítomnosti správy verzí sdílených komponent a jejich vzájemnému přepisování, čímž vznikají problémy se spuštěním aplikací využívajících různé verze téže komponenty nebo různé komponenty náhodou shodně pojmenované.

CLI definuje tzv. *assembly*, což je kolekce souborů s kódem nebo daty, jenž je vždy doplněna tzv. *manifestem*. V manifestu jsou uložena metadata popisující assembly. Musí zde být přesně specifikovaná verze assembly, její kultura (tj. národní prostředí), seznam obsažených souborů a seznam jiných assemblies, jejichž služeb je využíváno, a to včetně jejich kultur a verzí. Existují dva druhy assembly: *soukromé* a *sdílené*. Soukromé jsou vytvářené bez úmyslu poskytovat služby cizím programům. Tyto assembly nejsou nikde registrovány a není třeba, aby se systém zabýval jejich verzováním, jelikož pocházejí od jednoho autora. Naopak sdílené assembly jsou poskytovány jiným vývojářům a proto musí mít celosvětově jednoznačná jména a systém musí umožnit správu jejich verzí. Sdílené assembly jsou uloženy v *Global Assembly Cache (GAC)*, kam jsou nahrány při instalaci aplikace. Jelikož mají jednoznačná jména a verze, nedochází k přepisování existujících assembly a tím odpadá problém DLL Hell.

### Poznámka

Čísla verzí nejsou vytvářena nahodile tak, jak se autorům usmyslí, ale podle politiky, která umožňuje automatické nalezení vhodné verze komponenty, není-li k dispozici verze požadovaná nebo je-li k dispozici verze novější. Číslo verze se skládá ze čtyř částí: (major version).(minor version).(build number).(revision number). Pokud jsou v komponentě provedeny změny, jež nejsou kompatibilní se starší verzí, je třeba změnit major či minor verzi. Jsou-li provedeny jen opravy, které nemění význam a způsob volání jednotlivých poskytovaných služeb, mění se build nebo revision number. Jako kompatibilní se považují assemblies mající stejné major a minor verze. Při hledání assembly, která se má použít, systém vybere tu verzi, jež je kompatibilní a jež má nejvyšší build a revision number. Pokud není nalezena kompatibilní verze, je ohlášena chyba a aplikaci nelze spustit. Toto je implicitní chování, které může aplikace přepsat ve svém konfiguračním souboru, nevyhovuje-li.

Další zajímavou možností assembly v .NET je ochrana vlastního kódu. Při vytváření assembly jsou spočítány hash hodnoty ze souborů, které obsahuje, a jsou umístěny do manifestu. Obsah souboru nesoucí manifest je pak opět hashován. Výsledná hodnota je šifrována privátním klíčem vygenerovaným autorem a uložena do manifestu (do části, která není hashována). Spolu s privátním klíčem si autor vygeneruje i veřejný klíč, který je také umístěn do nehashované části manifestu. Tak je CLR umožněno při spuštění aplikace zkontrolovat, zda nebyl měněn její kód či data. Pomocí veřejného klíče se také generuje statisticky jednoznačné jméno sdílené assembly.

### 3.1.3 Atributy a metadata

Všechny jazyky splňující standard CLS musí umožňovat popis kódu tzv. *atributy*. Atributy jsou metadata přidaná kompilátorem ke kódu MSIL. Podle hodnot některých atributů se pak řídí běh aplikace v CLR, některé atributy jsou také využívány kompilátory. Programátor má taktéž možnost s metadatami pracovat za běhu programu. Atributy lze popisovat různé části kódu – celou assembly, třídu nebo jednotlivé metody. Atributy užívané pro celou assembly nesou např. jméno autora, verzi, popis assembly, apod. Příkladem atributu používaného na metody je atribut `Obsolete`. Pokud je tímto atributem označena metoda, kterou někde voláme, kompilátor nám u jejího volání zobrazí varování, že voláme zastaralou metodu.

Každý atribut je definován třídou odvozenou ze základní třídy `System.Attribute`. Lze tedy používat nejen předdefinovaných atributů – potomků této třídy, ale je také možno definovat vlastní atributy vytvořením nového potomka této třídy. Pomocí speciálního atributu použitého na nového potomka lze navíc definovat, na jakou část kódu může být atribut použit a jeho další vlastnosti.

Metadata umožňují, aby byl kód sebe-popisující. Narozdíl od technologií jako COM či CORBA není tedy třeba popisovat kód zvlášť např. pomocí IDL (Interface Definition Language), záznamů v nějaké databázi komponent (ve Windows v registru) apod.

### 3.1.4 Class Library

Již víme, že lze aplikace pro .NET psát v různých jazycích, které jsou překládány do jazyka MSIL. Při psaní jejich zdrojového kódu je však také podstatné, jak program může komunikovat se svým prostředím. Jakou funkci má programátor použít např. pro vypsání textu na konzoli, programuje-li konzolovou aplikaci? Pokud bychom volali přímo API funkce konkrétního systému, nebyl by kód přenositelný na jiný operační systém. Také proto je další důležitou součástí .NET Framework knihovna tříd (*class library*), jež obsahuje množství tříd a rozhraní. Některé z nich jen obalují API funkce systému, jiné poskytují rozšířenou funkcionalitu. S přechodem na jiný operační systém, na kterém je implementován .NET Framework, pak aplikace nebude třeba přepisovat ale ani znovu překládat, neboť již budou přeloženy do MSIL a budou volat stále stejné metody knihovny tříd. Jde o podobnou myšlenku, která je již také využita v jazyce Java či dvojici prostředí Delphi-Kylix firmy Borland<sup>3</sup>.

Pomocí MSIL a class library je dosaženo nezávislosti aplikací na operačním systému a použitém hardwaru. Stačí, aby na cílovém systému a hardware byl implementován .NET Framework. Jde o analogii konkurenčního prostředí Javy, kde roli .NET Frameworku hraje Java Virtual Machine. Programy pro .NET Framework by však měly běžet rychleji, jelikož nejsou interpretovány, ale jsou vykonávány přímo procesorem. Navíc lze při JIT kompilaci provádět optimalizaci kódu na daný procesor, což dále zvyšuje její běhovou rychlost, avšak za cenu zpomalení kompilace (vyplatí se tedy především u kompilace při instalaci).

Všechny třídy a rozhraní jsou v class library dostupné přes *prostředí jmen (namespace)* všem druhům aplikací bez ohledu na jazyk, ve kterém jsou napsány. Pro ilustraci uveďme několik příkladů. Pokud chceme v konzolové aplikaci vypsát text na konzoli, voláme metodu `WriteLine` třídy `Console` v namespace

---

<sup>3</sup> Hlavní součástí prostředí Delphi běžících na systému Windows je od verze 6 knihovna komponent VCL, jejíž část pojmenovaná CLX je dostupná i v prostředí Kylix, který běží na systému Linux. Jde vlastně také o runtime.

`System` (zapsáno v tečkové notaci jazyka C#: `System.Console.WriteLine`). Chceme-li vytvořit webovskou službu, vytvoříme potomka třídy `System.Web.Services.WebService`). Vytváříme-li webovskou stránku s kódem na straně serveru v ASP.NET, bude odvozena od třídy `System.Web.UI.Page`. Třídy pracující s databázemi (ADO.NET) jsou obsaženy v namespace `System.Data` a tak dále. Kompletní reference hierarchie tříd je uvedena v dokumentaci MSDN [3]. Na samém vrcholu této hierarchie je třída `System.Object`, ze které jsou odvozeny všechny ostatní třídy. Tato hierarchie také umožňuje pojmenovávat nové třídy a rozhraní, aniž bychom se dostali do konfliktu s existujícími jmény.

#### **Poznámka**

Common Language Runtime, Common Language Specification, Common Type System, části Class Library, assembly a další jsou standardizovány společností ECMA ve standardu ECMA-335: *Common Language Infrastructure* [14].

## **3.2 Desktopové aplikace**

Novinkou v oblasti vývoje desktopových aplikací s grafickým uživatelským rozhraním (GUI) poskytovanou prostředím .NET Framework jsou tzv. *Windows Forms (WinForms)*. Jejich funkcionality je dostupná přes třídy obsažené v namespace `System.Windows.Forms`.

WinForms poskytuje prostředí pro tzv. *RAD (Rapid Application Development)* GUI aplikací, skombinuje-li se navíc s vhodným nástrojem pro vizuální návrh. Cílem RAD je umožnit rychlý návrh designu aplikací tak, aby se programátor mohl soustředit na vlastní funkcionality programu a přitom ji mohl vhodně graficky prezentovat. WinForms obsahují třídy ovládacích prvků (tlačítka, tabulky, textová okna, apod.) a třídy kontejnerů (formuláře, menu, atd.) nesoucí vždy několik ovládacích prvků. Jednotlivé komponenty mezi sebou komunikují pomocí událostí, jak je tomu v této oblasti zvykem.

WinForms nabízejí podobné prostředky jako již několik let nástroje Delphi a C++ Builder firmy Borland prostřednictvím *VCL (Visual Component Library)* či Java ve svém balíku *AWT (Abstract Window Toolkit)*. Hlavním tématem tohoto textu je však programování pro Internet, nebudeme se tedy pouštět do detailního rozboru struktury WinForms.

## **3.3 Client-side aplikace na Internetu**

Od dob Internet Exploreru 3.0 je možno do HTML stránek vkládat tzv. *ActiveX* objekty. Jsou to objekty postavené na technologii COM a proto jsou také omezeny jen na platformu Windows. Prakticky jakákoliv aplikace může být vytvořena v podobě ActiveX objektu. Lze implementovat jednoduché objekty zobrazující prostý text i složité objekty typu tabulek programu Excel apod. Známým příkladem ActiveX objektu hojně využívaným na Internetu je Macromedia Flash. Ten umožňuje vytvářet animované stránky s pokročilým grafickým designem.

Na stránku můžeme ActiveX objekt umístit pomocí tagu `<object>`, jemuž nastavíme parametr *classid*. Ten obsahuje hodnotu *GUID (Global Unique Identifier)* – jednoznačné hexadecimální číslo identifikující COM objekt. Seznam objektů se na klientovi uchovává v systémovém registru Windows. Pokud na cílovém

počítači ještě není přítomen objekt, na který se stránka odkazuje, je prohlížečem stáhnut z adresy dané parametrem CodeBase tagu `<object>` a nainstalován, tzn. jsou přidány záznamy do registru systému. Pak již lze COM objekt spustit a zobrazovat jeho grafický výstup v prohlížeči.

Některé ActiveX objekty slouží k zobrazování a práci s formáty souborů, jež nejsou zobrazitelné v prohlížeči. Jsou jimi zmiňované objekty Macromedia Flash, dokumenty *PDF (Portable Document Format)* a mnohé další. Aby tyto vložené objekty nebyly omezeny jen na platformu Windows, existuje v prohlížečích jako je např. Mozilla možnost nainstalovat tzv. *plug-in*. Je to knihovna, kterou prohlížeč načte při svém spuštění a která modifikuje vnitřní chování prohlížeče. Tímto způsobem přidává prohlížeči novou funkcionalitu a to různého typu. Některé plug-iny lze asociovat s MIME typem. Je-li pak pomocí tagu `<embed>` vložen do HTML stránky soubor určitého MIME typu, je spuštěn asociovaný plug-in.

ActiveX nabízí takto podobnou funkcionalitu jako applety, o nichž budeme mluvit později. S použitím této technologie je však spojeno poměrně velké bezpečnostní riziko. ActiveX objekty totiž nejsou příliš omezovány ve své činnosti a proto může jejich autor na cílovém počítači i škodit. Kromě omezenosti na platformu Windows je toto jejich další nevýhoda.

Možnosti poskytované ActiveX objekty byly v .NET nahrazeny *WinForms*. WinForms umožňují vytvářet uživatelské komponenty (ovládací prvky) implementací potomka třídy `System.Windows.Forms.Control`. Tyto mohou být vloženy do jakékoliv aplikace užívající WinForms a chovají se stejně jako zabudované komponenty. Navíc je lze přidat do HTML stránky podobně jako ActiveX objekty. Stačí vložit zmiňovaný tag `<object>`, jehož atribut `classid` však nebude obsahovat GUID (ani nemůže, protože nové komponenty nejsou COM objekty), ale odkaz na assembly, ve které je komponenta implementována, a plné jméno třídy komponenty. Chceme-li například na stránku vložit komponentu implementovanou třídou `Komponenta1` v namespace `MojeKomponenty` zkompilovanou do assembly `Komponenty.dll`, přidáme do HTML kód:

```
<object classid="Komponenty.dll#MojeKomponenty.Komponenta1"></object>
```

Dále lze také přidat parametry a přistupovat k nim na HTML stránce přes JavaScript.

Oproti ActiveX mají takto vytvořené komponenty několik výhod. Nejdůležitější je asi bezpečnost. CLR implicitně zakazuje komponentě přistupovat na jiné soubory, než na ty, které se nacházejí v místě umístění její assembly (tj. na serveru). Dále jsou zakázány přístupy do klientského systému apod. Pokud klient komponentě důvěřuje, může některá omezení zrušit. Další výhodou je velice jednoduchý vývoj komponent narozdíl od komponent COM. Stačí pouze implementovat třídu komponenty a to je vše. Nevýhodou je nutnost přítomnosti .NET Frameworku na klientském počítači. Narozdíl od ActiveX však .NET Framework není omezen na platformu, tudíž na ni nejsou omezeny ani WinForms komponenty.

#### **Poznámka**

Jelikož existuje mnoho již fungujících ActiveX komponent, je ActiveX dostupné i přes .NET Framework. Stejně tak lze pracovat i s jinými technologiemi založenými na COM.

## 3.4 Server-side aplikace na Internetu

### 3.4.1 ASP.NET

ASP.NET je souhrnný název pro prostředky, které poskytuje platforma .NET pro vývoj aplikací běžících na serveru. Jeho součástmi jsou tzv. *Web Forms*, *Web Controls* a *Web Services*.

#### Poznámka

Podle názvu ASP.NET by bylo možné usoudit, že se jedná o další verzi ASP. Web stránky napsané v ASP skutečně budou i nadále fungovat. ASP.NET však představuje mnohem komplexnější řešení zahrnující všechny aspekty programování na serveru. Těmi jsou zejména prostředí pro vývoj web služeb, web stránek, udržování stavu aplikací, cachování, vazba na databáze, atd.

### 3.4.2 Práce s web stránkami: Web Forms, Web Controls

Jak jsme se zmínili v úvodu, nejsou zdrojové kódy současných stránek psaných v PHP či ASP kvůli prolínání různých jazyků příliš přehledné. Také je třeba ručně implementovat uchovávání stavu stránek na serveru.

Nejprve si popíšeme tzv. *session management*. Je to obecná technika, která obchází nestavovost protokolu HTTP. Session management umožňuje uchovávat na serveru proměnné, které jsou spojeny s klientem procházejícím naše stránky. Klient nejprve navštíví první stránku, kde dostane přiděleno jedinečné číslo identifikující jeho relaci (tzv. *session*). Toto číslo (*session-id*) se poté protokolem HTTP přenáší od serveru ke klientovi a zpět s každým dotazem a odpovědí tak, aby bylo na serveru k dispozici vždy, když je generována stránka. PHP/ASP skript má pak k dispozici všechny proměnné uložené na serveru patřící dotyčnému klientovi (jaké proměnné to jsou se pozná právě pomocí *session-id*).

Přenos *session-id* se provádí dvěma způsoby. První způsob využívá možnosti přidávání parametrů za URL nebo jako součást cesty v URL. Na severu pak musí docházet k doplňování *session-id* do všech odpovídajících odkazů. Druhou možností je využít cookie v HTTP. *Session-id* se uloží do cookie na klientském počítači a je automaticky prohlížečem posíláno zpět na server. Prohlížeče však umožňují zakázat práci s cookie, což nutí programátory umožnit jiný způsob přenášení *session-id*.

Pomocí *session* proměnných můžeme udržovat informace na serveru po dobu celé relace. Pomocí této techniky vyřešíme první příklad z úvodu. Předpokládejme, že uživatel navštívil stránku s informacemi o studentovi a má již přiděleno *session-id*. Chce-li uživatel vybrat školu, pošle přes formulář na server aktuální data studenta ze stránky a také nastavený příznak, že chce vybírat školu. Data posíláme stejnému skriptu, který generoval stránku se studentem. Tento skript uloží data do *session* proměnných a vygeneruje stránku se seznamem škol. Výběrem školy uživatel odešle na server opět stejnému skriptu zvolenou školu a příznak, že byla vybrána škola. Skript znovu vygeneruje stránku studenta, doplní jeho osobní údaje ze *session* proměnných a název školy z dat, která poslal klient.

Podobně by mohlo vypadat řešení bez ASP.NET. Toto řešení splňuje i náš požadavek na univerzalitu kódu realizujícího výběr školy (kdyby to nebylo třeba, mohli jsme si ušetřit práci se *session* proměnnými). Je ale vidět (zvláště z kódu, který je třeba napsat), že toto řešení nebude asi příliš přehledné, budeme-li chtít vytvořit komplikovanější aplikaci, kde je podobných výběrů či jiných operací na jedné stránce potřeba více.

Dále si popíšeme, jaká vylepšení přináší ASP.NET a jak lze podobné situace řešit s jejich pomocí. Nečekejme nějaké zázračné změny. Řešení bude podobné, nicméně část kódu, kterou jsme museli nyní programovat sami, zajistí automaticky server a vzniklé řešení bude navíc přehlednější a snadněji rozšiřitelné.

ASP.NET nadále umožňuje vkládat do zdrojového HTML kódu stránek „závorky“ `<? ?>`. Do nich lze však doplnit kód psaný v libovolném jazyce dostupném na platformě .NET (například C#). Tento způsob se však nedoporučuje hojně používat, jelikož vede k nepřehlednému kódu. Místo toho se do HTML vkládají speciální tagy s nastaveným atributem `runat` na hodnotu „server“. Tyto tagy jsou při generaci stránky nalezeny, zpracovány a místo nich se může vygenerovat nějaký HTML kód. Aby se soubory s novým způsobem zpracování odlišily od stránek ASP, přidává se za jejich jméno přípona `.aspx` místo původní přípony `.asp`.

Tagy obsažené v `.aspx` stránce můžeme rozdělit do tří skupin. První skupinou jsou stávající tagy HTML, druhou nové tagy implicitně zabudované do ASP.NET a poslední skupinu tvoří uživatelsky definované tagy. Tagy zpracováváné na serveru se nazývají *ovládací prvky* (*Server Controls*). Každý ovládací prvek je instancí třídy, která implementuje jeho vlastnosti a chování. Tento objektově orientovaný přístup umožňuje vytvořit objektový model stránky a třídu obecné stránky (kontejner) zahrnout do hierarchie tříd class library (konkrétně jde o třídu `System.Web.UI.Page`). Z této třídy jsou pak odvozeny všechny uživatelem vytvořené stránky.

Ve třídě stránky i v každé třídě ovládacího prvku jsou také definovány různé události. Umožňují pokročilou práci se stránkou na straně serveru. Kde je však tento kód umístěn? Je-li na stránce třeba obsloužit jednoduchou událost, je přijatelné vkládat kód pomocí „závorek“ `<? ?>` nebo lépe pomocí tagu `<script runat="server"></script>`. Je-li však potřeba komplexnější kód, umísťuje se do separátních souborů, které jsou ke stránce připojeny v jejím záhlaví, kde je deklarován seznam připojených souborů (podobně jako se v Pascalu deklaruje seznam použitých jednotek klíčovým slovem `uses`, nikoliv však způsobem direktivy `#include` jazyka C, která soubory spojí do jednoho). Takový kód v pozadí stránky se označuje jako *Code Behind*.

ASP.NET obsahuje implicitně 45 předdefinovaných ovládacích prvků (*Web Controls*). Jsou to potomky třídy `System.Web.UI.WebControls.WebControl` a jsou umístěny v namespace `asp`. Uveďme několik jednoduchých příkladů:

```
<asp:Button id="Button1" runat="server" Text="Button"></asp:Button>
<asp:Label id="Label1" runat="server">Label</asp:Label>
```

Uvedený kód ASP.NET se po zpracování na serveru nahradí kódem HTML:

```
<input type="submit" name="Button1" value="Button" id="Button1">
<span id="Label1">Label</span>
```

Kromě předdefinovaných ovládacích prvků lze definovat i prvky uživatelské (*User Controls*). Ty se definují opět zcela odděleně od stránky, kde jsou použity. První možností, jak definovat uživatelský prvek, je napsat *Code Behind* a v něm implementovat potomka třídy určené k vytváření uživatelských prvků (vhodné především pro prvky, které se nenahrazují kódem HTML nebo se nahrazují jen kódem krátkým). Druhou možností je vzít část `.aspx` stránky, přesunout ji do zvláštního souboru (označeného příponou `.ascx`) a prohlásit ji za uživatelský prvek. Na takovýto soubor pak stačí přidat odkaz do záhlaví stránky, kde chceme

vytvořený prvek použít, a v tomto odkazu definovat jméno a předponu tagu (namespace). K souboru .ascx lze opět připojit Code Behind a tím prvek „oživit“. Při zpracování takto definovaného prvku je nejprve zpracován soubor .ascx (může totiž obsahovat další server-side prvky) a výsledek je vložen na místo zpracovávaného prvku.

Jaké máme možnosti řízení stavu stránek? ASP.NET také uchovává session proměnné resp. objekty a to přes URL nebo přes cookie (podle nastavení). Session objekty lze navíc oproti PHP/ASP uchovávat také ve zvláštním procesu běžícím na serveru (ASP.NET State Server Process) nebo i v databázi na SQL serveru. Uchovávání objektů v databázi je pomalejší, umožňuje však vysokou spolehlivost.

Zachovávání stavu stránek již není třeba do stránek dodělavat. Je totiž na stránky přidáváno automaticky, pokud není programátorem v objektu stránky nastaveno jinak. Stav stránky je dán daty, která nesou ovládací prvky. Každá stránka zachovávající si svůj stav obsahuje formulář (tag `<form runat="server" method="post">`), ve kterém jsou umístěny všechny ovládací prvky. Třídy všech ovládacích prvků mají vlastnost `ViewState` typu asociativní pole, ve které ovládací prvek může mít uloženy proměnné, jež chce zachovávat. Při generaci stránky se pak automaticky projdou všechny ovládací prvky na stránce, obsahy jejich vlastností `ViewState` se zakódují do řetězce a ten je pak vložen do formuláře pomocí skrytého tagu `<input type="hidden" name="__VIEWSTATE">`. Data formuláře jsou vždy odesílána na stejnou stránku, jakou byla vygenerována (parametr `action` tagu `<form>` je prázdný). Před spuštěním Code Behind reagujícím na doručení dat jsou serverem obnoveny hodnoty vlastností `ViewState` stránky a všech ovládacích prvků. Tento způsob je pro programátora stránky naprosto transparentní a k jeho uplatnění nemusí napsat ani jedinou řádku kódu.

Jak by vypadaly stránky příkladu z úvodu v ASP.NET? Nejprve vytvoříme stránku sloužící pro výběr školy a uložíme ji jako uživatelský prvek (soubor .ascx). Poté k ní napíšeme Code Behind definující událost vyvolanou při zvolení školy a vracející název vybrané školy. Tuto událost obslouží kód stránky, kde bude prvek použit. Tím je hotova komponenta, kterou lze v jakémkoliv stránce použít pro výběr školy.

Nyní vytvoříme stránku studenta. Bude obsahovat prvky s údaji o studentovi (např. `<asp:TextBox>` či `<asp:Label>`), tlačítko pro výběr školy (`<asp:Button>`) a prvek, jež jsme vytvořili pro výběr školy pojmenovaný např. `<Skoly:Vyber>`. Uživatelský prvek schováme nastavením vlastnosti `Visible` na `false`. Nyní stačí doplnit pár řádek Code Behind. Stisknutím tlačítka pro výběr školy je formulář odeslán na server metodou POST a tím je na serveru v objektu stránky vyvolána událost. Tu obsloužíme tak, aby zobrazila prvek `<Skoly:Vyber>`. Klient dostane jako odpověď stránku, kde vybere školu. Tím vyvolá námi napsanou událost, jež obsloužíme kódem, který schová prvek `<Skoly:Vyber>` a doplní jméno školy do formuláře. Jak je patrné, kód v pozadí stránek se zabývá jen aplikační logikou (jaké stránky se mají kdy zobrazit a kam se mají údaje doplňovat), nikoliv správou stavu prvků. Navíc jsme jednoduše vytvořili komponentu, kterou můžeme znovu využít jinde.

Poznamenejme ještě, že Code Behind i kód vložený do stránky .aspx se kompiluje a výsledný kód MSIL je ukládán na serveru. Tím dochází ke podstatnému zrychlení běhu kódu na serveru. Celá stránka .aspx se za běhu chová jako dynamická knihovna, která poskytuje serveru rozhraní a přes něj také výstupy, jež mají být odesílány klientovi. Aby nebylo třeba stále kompilovat kód a zpracovávat stránky, je použito několika úrovní cache.

Filosofie práce se stránkami na straně serveru, jak je navržená v ASP.NET, umožňuje komponentální programování stránek, oddělení deklarativního kódu stránky od procedurálního kódu pracujícího se stránkou a všechny výhody z toho plynoucí. Jelikož ovládací prvky jsou třídy, lze využít i dědičnosti a dalších výhod OOP. Na druhou stranu je zpracování stránek o něco pomalejší, což ale kompenzuje propracovaný caching. Jistou nevýhodou je také novota technologie. ASP.NET není ještě natolik zaběhlá, aby byly odstraněny všechny zásadní chyby. Sám jsem také na nějaké narazil (např. možnost shodit server pomocí umístění velmi dlouhého řetězce do proměnné `__VIEWSTATE`).

#### **Poznámka**

V tomto relativně omezeném prostoru nelze zmínit všechny možnosti ASP.NET, které lze využít při vytváření web stránek. Nezbylo zde místo na popsání dalších možností uchovávání stavu, vazby ovládacích prvků na databáze pomocí ADO.NET, validačních ovládacích prvků, koncepce zajištění bezpečnosti publikovaných dat, možnosti konfigurace webových aplikací a dalších součástí ASP.NET usnadňujících programování stránek. Více lze samozřejmě najít v příslušné dokumentaci na MSDN [3].

### **3.4.3 Web Services**

Vytváření stránek na Internetu je možnost, která je k dispozici již dlouhou dobu. Narozdíl od web služeb, jejichž koncepce vznikla poměrně nedávno. Je výsledkem společného úsilí velkých společností a je také z velké části standardizována konsorciem W3C. I zde má firma Microsoft své rozšíření (např. BizTalk), ale těmi se zabývat nebudeme. V této oblasti je dodržování daných standardů nutným předpokladem k dosažení cílů, ke kterým byly web služby navrženy. Jde zejména o možnost jednotného protokolu komunikace služeb a jejich jednotný způsob popisu, aby je bylo možné používat aniž by programátor klientské aplikace byl nucen znát jejich rozhraní předem.

Služba (*Web Service*) je objekt na serveru na Internetu, který poskytuje přes své rozhraní funkcionalitu jiným službám. V podstatě jde o distribuované komponentální programování. Aby bylo možné poskytovat službu programátorům klientských aplikací, musí existovat možnost, jak zjistit její umístění na Internetu. V našem druhém příkladu z úvodu programátor klienta věděl, že chce zjišťovat kurzy podle nějaké banky. Jak se ale služba jmenuje a na jakém serveru banky se nachází nevíme.

K vyhledávání služeb na Internetu slouží *UDDI (Universal Description, Discovery and Integration)*, což je registr služeb umístěný na Internetu s možností vyhledávání. Tento registr je vlastně také služba, ke které lze přistupovat programově. Její umístění je však známé. Výstupem z UDDI je seznam serverů, na kterých najdeme služby vyhovující zadaným kritériím. Pokud chceme zjistit seznam služeb dostupných na konkrétním serveru, použijeme protokol *SOAP Discovery* (zkrácované jako *Disco*). Tím tedy zjistíme, kde je služba umístěna a jak se přesně jmenuje.

Dále bychom potřebovali zjistit, jak se službou komunikovat. Popis metod dostaneme od služby, zavoláme-li její metodu, která musí být v každé službě přítomna a která vrací definici svého rozhraní v jazyce *WSDL (Web Service Description Language)*, jenž je podmnožinou XML. Ten přesně popisuje, jaké metody můžeme volat (včetně typů jejich parametrů), k jakým properties můžeme přistupovat a případně poskytuje nějaké komentáře. Každá služba je díky použití atributů v jejím MSIL kódu sebe-popisující a není třeba ji doprovázet dalšími popisy jako např. IDL.

Když už víme, jaké metody můžeme volat, potřebujeme také komunikaci nějakým protokolem realizovat. K tomu slouží přímo protokol HTTP nebo protokol *SOAP (Simple Object Access Protocol)* nad protokolem HTTP, jenž oproti HTTP umožňuje přenos komplikovanějších struktur (definuje jejich popis pomocí XML). Jeho prostřednictvím můžeme volat metody služby, předávat jim parametry a získávat návratové hodnoty.

V ASP.NET je zabudovaná výrazná podpora pro vytváření služeb. Každá služba je potomkem třídy `System.Web.Services.WebService`. Její vytvoření je velmi jednoduché, jelikož veškerá nízkoúrovňová funkcionalita pro komunikaci protokoly je již zabudována v .NET Frameworku. Kód služeb lze, jak je zvykem, psát v libovolném jazyce dostupném na .NET. K vytvoření služby stačí implementovat potomka třídy `WebService`. Potomek může mít libovolné metody, ale pro klienty jsou dostupné jen ty, které jsou označeny atributem `WebMethod`. Atribut může obsahovat i komentář, který je použit při generování WSDL popisu služby. Po kompilaci do MSIL a umístění na server je služba připravena reagovat na dotazy od klientů.

Dotazy na web služby lze posílat protokoly HTTP (metodou GET i POST) a to buď přímo (používáno nejčastěji při dotazu z HTML formuláře) nebo použitím protokolu SOAP. Příkladem může být volání metody `ZjistKurzy` webové služby `Kurzy` nějaké banky:

```
http://www.banka.cz/Kurzy.aspx/ZjistKurzy?mena=EUR&typ=nakup
```

Ze způsobu předávání parametrů při přímém volání přes protokol HTTP je vidět, že není možné předávat jiné typy parametrů než řetězce. Ačkoliv lze jistě každý jiný typ kódovat nějakým řetězcem, bylo by takové kódování u složitějších aplikací komplikované a jistě by každý programátor používal jiné. Proto byl vyvinut standardní protokol SOAP, jež právě takové kódování parametrů umožňuje.

Poté, co na server přijde žádost o volání metody web služby (přímo v HTTP nebo v SOAP), je serverem vyhodnocena, dále je spuštěna příslušná metoda služby s příslušnými parametry a její návratová hodnota je odeslána zpět klientovi ve formátu XML. V našem příkladě bychom dostali přibližně následující odpověď:

```
<?xml version="1.0" encoding="utf-8" ?><string>30,125</string>
```

Popis metod služby a jejich parametrů ve formátu WSDL lze získat zasláním dotazu:

```
http://www.banka.cz/Kurzy.aspx/?WSDL
```

Aby nebylo nutné při vytváření služeb psát WSDL dokument ručně, což by byla dosti vyčerpávající práce, při které by docházelo ke zbytečným chybám, umožňují vývojové nástroje jeho automatickou generaci na základě zdrojového textu třídy, která implementuje danou službu.

Kromě klientů dotazujících se na web služby pomocí formulářů HTML lze samozřejmě psát desktopové aplikace využívající web služby. Jelikož všechny protokoly použité pro komunikaci jsou standardní, nemusí být tyto klienti vytvářeni pod .NET. Stačí implementovat konstrukci dotazů SOAP a jejich posílání na server, kde se služba nachází. S výhodou lze však využít nástrojů, které z popisu služby ve WSDL vygenerují kód proxy třídy. Tato třída je v ASP.NET vždy potomkem třídy `SoapHttpClientProtocol` nacházející se v namespace `System.Web.Services.Protocols` a implementuje potřebné komunikační schopnosti. Proxy třídu přidáme ke zdroji programu klienta a vytvoříme v něm její instanci. Tato instance se bude chovat přesně jako web služba, kterou voláme. Následuje příklad volání metody služby v konzolové aplikaci v C#:

```
Kurzy kurzovniListek = new Kurzy();  
Console.WriteLine(kurzovniListek.ZjistKurz("EUR", "nakup"));
```

**Poznámka**

Takovéto volání služby je synchronní, tj. klientská aplikace je uspána do doby, než přijde odpověď. Služby je také možno volat asynchronně – při volání metody se specifikuje událost, která se má vyvolat, jsou-li již data k dispozici, a hned po volání se pokračuje v jiné práci.

## 4 Java 2, Enterprise Edition

Narozdíl od .NET Framework, J2EE není aplikace, ale pouze standard navržený firmou Sun. Je v něm popsáno, jakou základní funkcionalitu musí aplikace poskytovat, aby tento standard splňovaly. Vlastní implementace této funkcionality se ponechává na jednotlivých výrobcích software<sup>4</sup>. Implementací přímo od firmy Sun je J2EE SDK. Může být využita jen pro testování aplikací, které mají běžet pod J2EE, nikoliv pro komerční účely. Na standard není dáno žádné omezení, které by zakazovalo jeho rozšiřování.

Jediným programovacím jazykem platformy J2EE je jazyk Java. Java je překládána kompilátory do *byte-code* na počítačích, kde je aplikace vyvíjena, a ten je pak interpretován na cílových počítačích pomocí *Java Virtual Machine (JVM)*. Narozdíl od MSIL není kód většinou překládán do nativního kódu procesoru<sup>5</sup>. Jeho vykonávání je proto řádově pomalejší, na druhou stranu není třeba JIT kompilace. Aplikace v Javě běží v prostředí *Java Runtime Environment (JRE)*, jenž je analogií CLR. Knihovnou tříd je zde *Java Development Kit (JDK)*. Ten má kvůli různým potřebám aplikací několik edicí. Internetové multi-tier aplikace využívají plnou knihovnu (*Enterprise Edition*), řada aplikací si vystačí s její částí (*Standard Edition*) a mobilní telefony a podobná malá zařízení využívají *Micro Edition*. Pro seskupování souvisejících tříd, rozhraní a proměnných v knihovně tříd sloužily v .NET prostředí jmen. V Javě jsou jejich analogií *balíky (packages)*. Reference knihovny tříd je dostupná v libovolném vývojovém nástroji Javy (viz například [7]) nebo na stránkách firmy Sun [4].

### 4.1 Client-side aplikace na Internetu

Díky Javě máme již delší dobu možnost používat tzv. *applety*. Applety jsou malé programy, jejichž soubor .class s byte kódem je uložen na serveru, ale je vždy vykonáván na straně klienta v rámci jiného programu (obvykle Internetového prohlížeče). Analogií v .NET jsou WinForms Controls. Aby bylo možno používat applety, musí být na klientském počítači instalován Java Virtual Machine a prohlížeč je musí podporovat (jde o plug-in, který je ve většině prohlížečů obsažen implicitně nebo ho lze přidat).

Abychom v prohlížeči mohli spustit applet, vložíme do stránky HTML tag `<applet>` s atributem, který specifikuje umístění bytového kódu na serveru. Jakmile prohlížeč načte tento tag, spustí tzv. *Class Loader* (součástí JVM), jenž ze serveru stáhne kód appletu a kód všech potřebných tříd, na něž se applet odkazuje a které ještě na klientovi nejsou. Jsou-li všechny požadované soubory dostupné, je applet spuštěn. Aby byla zajištěna nezbytná bezpečnost při běhu appletu, omezuje dostupnou funkcionalitu tzv. *Security Manager*, který umožňuje uživateli zvolit, jaké skupiny funkcí mají být povoleny či zakázány. Obvykle nemají applety přístup do souborového systému a nemohou navazovat spojení s jinými servery, než odkud byly stáhnuty.

Applety se používají především k implementaci složitějšího grafického prostředí na straně klienta, než je schopen poskytnout jazyk DHTML. Za tím účelem je appletu na stránce HTML vymezen obdélníkový prostor, jehož rozměry jsou dány dalšími atributy tagu `<applet>`. V této oblasti je veškeré vykreslování ponecháno na appletu a prohlížeč ho neovlivňuje.

---

<sup>4</sup> Jednou z implementací J2EE je prostředí Borland JBuilder 5 Enterprise a server Apache Tomcat.

<sup>5</sup> Existují však i překladače bytového kódu do kódu nativního (např. Jove, BulletTrain, JET, a další).

## 4.2 Server-side aplikace na Internetu

K vytváření server-side aplikací je třeba Enterprise Edition. V této oblasti je J2EE je přibližně ekvivalentní ASP.NET. Některé rysy jsou implementovány téměř stejně, některé zcela jinak. Detailní popis J2EE nebude naším cílem, podíváme se jen stručně na techniky poskytující obdobnou funkcionalitu jako ASP.NET.

### 4.2.1 Servlets & Java Server Pages

*Servlety* jsou objekty napsané v Javě. Po umístění na server jsou adresovatelné protokolem HTTP. Pokud klient pošle požadavek na servlet, je servlet zaveden do paměti serveru, pokud tam ještě není. Poté je spuštěna metoda servletu odpovídající metodě HTTP (např. metodě GET odpovídá metoda `doGet`). Úkolem těchto metod je vytvoření odpovědi (např. stránky HTML). Ta je poté serverem odeslána klientovi. Servletu jsou k dispozici metody pracující s hlavičkou a parametry HTTP, session objekty, cookies, apod.

*Java Server Pages (JSP)* jsou podobné `.aspx` stránkám v ASP.NET. Jsou to textové soubory napsané v jazyce rozšiřujícím HTML o nové tagy. JSP mohou také obsahovat kód uzavřený v „závorkách“ `<% %>`, který je napsán v Javě. Některé tagy jsou nahrazovány HTML kódem a splňují tedy podobnou funkci jako ovládací prvky v ASP.NET. Jiné slouží k přístupu na komponenty *JavaBeans*<sup>6</sup> implementující aplikační logiku. Lze vytvářet i uživatelské tagy (*custom tags*). K tomu je třeba vytvořit tzv. *tag library*, což je XML soubor, v němž jsou tagy definovány. Po přidání reference na tento soubor do záhlaví stránky, můžeme nové tagy na této stránce používat.

### 4.2.2 Enterprise JavaBeans

*Enterprise JavaBeans (EJB)* jsou komponenty běžící na serveru naprogramované v Javě. Jsou dvojího typu: *session beans* a *entity beans*. Volání služeb EJB je vždy vzdálené, tj. pomocí RMI.

Entity bean je trvalý objekt uložený např. v databázi využívaný zpravidla více klienty. Reprezentuje nějaký skutečný objekt se složitějším rozhraním, se kterým aplikace pracuje a který je třeba poskytnout více klientům současně. Příkladem entity bean může být uživatelský účet.

Session bean je dočasný objekt vytvořený pro jednoho klienta existující po dobu existence jeho session (session se udržuje tak, jak bylo popsáno v kapitole 3.4.2). Session bean, které udržují svůj stav vzhledem ke klientovi (tj. jsou vázány na konkrétního klienta), se nazývají *stateful session beans*. Používají se k implementaci aplikační logiky (řízení jiných objektů) nebo reprezentují dočasné objekty (např. nákupní košík v elektronickém obchodu). Naopak funkcionalita poskytovaná *stateless session beans* nezávisí na klientovi, který se k nim připojí. Není třeba pro ně udržovat vazbu na klienta a proto také nekonzumují tolik

---

<sup>6</sup> JavaBean je komponenta (objekt s vlastnostmi – properties – a metodami řízený událostmi), která umožňuje komunikovat s ostatními komponentami a která splňuje další požadavky vyplývající ze specifikace [13]. Pokud komunikují dvě vzdálené komponenty napsané v Javě, je používáno *Remote Method Invocation (RMI)*. JavaBeans umožňují také komunikaci s objekty modelu CORBA (*Common Object Request Broker Architecture*) protokolem IIOP (*Internet Inter-ORB Protocol*) a své rozhraní popisují pomocí IDL (*Interface Definition Language*).

systemových prostředků serveru. Jejich ekvivalentem v ASP.NET jsou web services. Příkladem tedy může být v předchozích kapitolách zmiňovaný kurzovní lístek, katalog výrobků a další objekty.

Architektura EJB umožňuje pomocí Java XML metod komunikovat i se službami podporujícími protokol SOAP, jako jsou např. služby v ASP.NET. EJB lze použít i pomocí běžného formuláře na HTML stránce. Nelze to však přímo, ale přes servlet. Prohlížeč pošle na servlet dotaz metodou GET resp. POST a servlet zavolá příslušnou metodu EJB přes RMI a podle ní vytvoří odpověď, jež bude poslána zpět prohlížeči jako HTML stránka.

## 5 Další informace

Ke zpracovanému tématu existuje řada článků na Internetu i publikací. Mými zdroji byly především:

### Literatura:

- [1] Dalibor Kačmář: *Programujeme .NET aplikace ve Visual Studiu .NET*, Computer Press, 2001
- [2] Bruce Eckel: *Thinking in Java*, v českém překladu: *Myslíme v jazyku Java*, Grada, 2000

### Elektronické dokumenty:

- [3] MSDN: <http://msdn.microsoft.com>  
Class Library Reference: [/library/en-us/cpref/html/cpref\\_start.asp](http://msdn.microsoft.com/library/en-us/cpref/html/cpref_start.asp)
- [4] Sun J2EE:  
[http://java.sun.com/blueprints/guidelines/designing\\_enterprise\\_applications](http://java.sun.com/blueprints/guidelines/designing_enterprise_applications)  
<http://developer.java.sun.com/developer/infodocs>
- [5] Chad Vawter, Ed Roman: *J2EE vs. Microsoft.NET: A comparison of building XML-based web services*  
<http://www.theserverside.com/resources/articles/J2EE-vs-DOTNET/article.html>
- [6] Netscape DevEdge: <http://devedge.netscape.com>  
Netscape Plugin API: [/library/manuals/2002/plugin/1.0](http://devedge.netscape.com/library/manuals/2002/plugin/1.0)
- [7] Borland JBuilder 5 Documentation (dokumentace k vývojovému prostředí Borland JBuilder 5)  
<http://www.borland.com/jbuilder/index.html>

### Specifikace:

- [8] WSDL: <http://www.w3.org/TR/wsdl>
- [9] UDDI: <http://www.uddi.org>
- [10] XML 1.0: <http://www.w3.org/tr/1998/rec-xml-19980210>
- [11] DOM Level 1 Core: <http://www.w3.org/tr/rec-dom-level-1>
- [12] DOM Level 2 Core: <http://www.w3.org/tr/dom-level-2>
- [13] JavaBeans: <ftp://ftp.javasoft.com/docs/beans/beans.101.pdf>
- [14] ECMA-335: *Common Language Infrastructure*:  
<http://www.ecma.ch/ecma1/stand/ecma-335.htm>