Microsoft .net Framework

Příručka vývojáře pro přechod na platformu Microsoft .NET

Microsoft Visual Studio .net

http://msdn.microsoft.com/net/upgrading

Přecházíme na platformu Microsoft .NET

Microsoft Corporation

POZNÁMKA PRO NAŠI VÝVOJÁŘSKOU KOMUNITU

S vydáním softwarového produktu Microsoft® Visual Studio® .NET 2003 jsme přivedli na svět již druhou verzi mocných a vzrušujících nástrojů pro budování a rozmísťování distribuovaných aplikací pro operační systémy Windows, Web a mobilní zařízení. Nejnovější vydání tohoto softwarového produktu je postaveno na všech výtečných vlastnostech našich předcházejících produktů a zahrnuje nové a vskutku průkopnické schopnosti a atributy, přesně také, kterých jste se dožadovali. Abychom vám pomohli lépe a snáze nalézt cestu ke všem novým technologiím, které vývojová platforma Microsoft .NET zastřešuje, připravili jsme pro vás tuto příručku. Kromě toho vám ukážeme, jak provést migraci vašich dosavadních znalostí a aplikací tak, aby byl váš přechod na platformu Microsoft .NET co možná nejefektivnější.

Dokumenty a dva CD disky, které jsou součástí této příručky, obsahují informace, které budete potřebovat proto, abyste z produktu Visual Studio .NET a platformy .NET Framework získali co nejvíce:

- Příručku tvoří některé z našich nejpopulárnějších a vysoce hodnocených dokumentů ze série *Přecházíme na platformu Microsoft .NET.* Naleznete zde migrační průvodce pro mnohé programovací jazyky a technologie. Kromě jiných se dozvíte důležité informace o přechodu na technologii ASP.NET, či o programovacích jazycích jakými jsou Visual Basic .NET, řízené (managed) C++ a C#. Stranou ovšem nezůstanou ani programátoři pracující v jazyce Java, neboť pro ně je připravena samostatná sekce s názvem *Přenos Java aplikací na platformu .NET Framework.* V neposlední řadě se můžete těšit také na vyčerpávající informace o ADO.NET, COM Interop a společném běhovém prostředí (Common Language Runtime, CLR).

- První CD disk, označený jako *Upgrading to Microsoft .NET (Inovujeme na Microsoft .NET)*, obsahuje úplnou kolekci materiálů, které vás uvedou do problematiky platformy .NET Framework, technologie ASP.NET a produktu Visual Studio .NET. Najdete zde také studie o vytváření tříd, práci s frontami zpráv a používáním XML webových služeb namísto DCOM. Většina těchto článků se zaměřuje především na vývojáře používající jazyk Visual Basic, ovšem vykládanou problematiku lze uplatnit také v dalších programovacích jazycích Visual Studia .NET. Pokud si chcete prohlédnout seznam všech položek, které jsou na CD disku umístěny, nalistujte stranu 194.

  Na prvním CD disku můžete dále najít sadu cvičení a postupů, které vysvětlují proces provádění inovací aplikací z Visual Basicu 6.0 do Visual Basicu .NET a z Javy/JSP do C#/ASP.NET. Tyto materiály pro vás připravila společnost ArtinSoft, která je kromě jiného známá také tím, že její pracovníci vytvořili nástroje **Visual Basic Upgrade Wizard** a **Java Language Conversion Assistant**. Cvičení představují skvělou pomůcku pro získání základních znalostí o uvedených nástrojích, procesu konverze programového kódu různých programovacích jazyků a charakteristice nejběžnějších migračních scénářů. Ukázkové postupy vám pomohou urychlit proces vzdělávání a také vám ukážou, jak může být proces portace kódu aplikací efektivnější a produktivnější. Jakmile budete vybaveni patřičnými znalostmi .NET platformy a postupy pro inovaci vašich aplikací, budete moci získané dovednosti aplikovat i v mnoha jiných aplikacích. Navíc, vaše aplikace budou s podporou .NET platformy robustnější než kdykoliv předtím.

- Na druhém CD disku se nachází nový *Resource Kit* pro jazyk Visual Basic .NET, jenž představuje hodnotnou a vývojáři oblíbenou sadu cenných zdrojů pro programátory ve Visual Basicu. Visual Basic .NET Resource Kit je naplněn skutečně obrovským množstvím kvalitních .NET komponent třetích stran, které jsou vám poskytovány zcela zdarma. S Resource Kitem obdržíte také kompletní řešení ukázkových aplikací určených pro Windows i Web. Zdrojový kód ukázkových aplikačních řešení můžete použít i v jiných aplikacích, můžete jej modifikovat podle svých potřeb, nebo jej můžete použít pro studijní účely. Kdybychom chtěli popisovat všechny elementy, které tvoří Visual Basic .NET Resource Kit, jistě bychom zaplnili ještě mnoho stránek, proto se k dalším součástím této sady vyjádříme jenom stručně: Čekají na vás stovky ukázek zdrojových kódů aplikací, tréninková videa, zajímavé články a studie, ukázkové kapitoly ze znamenitých odborných publikací a mnohem více.

  Našim cílem je podpořit a povzbudit vývojáře, programátory a softwarové architekty, aby pomocí softwarového produktu Visual Studio .NET začali vytvářet novou generaci počítačových aplikací. Již nyní se nemůžeme dočkat všech těch nových a vzrušujících řešení, které dokážete vyvinout prostřednictvím platformy .NET Framework. Doufáme, že tato příručka je dobrým startovním bodem. Budete-li chtít získat podrobnější informace o procesu inovace na novou vývojovou platformu, navštivte prosím *http://msdn.microsoft.com/net/upgrading*.

S úctou

Marie K. Huwe
General Manager
Microsoft .NET Platform a Evangelism Group

Kapitola 6: OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ V JAZYCE VISUAL BASIC .NET

Kapitola 7: JAK NA LADĚNÍ APLIKACÍ V JAZYCE VISUAL BASIC .NET

Kapitola 8: DATOVÉ VAZBY A OBJEKTY PRO PŘÍSTUP K DATŮM V KNIHOVNĚ WINDOWS FORMS

Kapitola 9: C++ → C#: CO POTŘEBUJETE VĚDĚT PRO PŘECHOD Z C++ K C#

Kapitola 10: NATIVNÍ A ŘÍZENÉ C++: JAZYKOVÁ INTEROPERABILITA

KAPITOLA 11: QUAKE II .NET

KAPITOLA 12: PŘENOS JAVA APLIKACÍ NA PLATFORMU .NET FRAMEWORK

KAPITOLA 13: PŘENOS NATIVNÍHO KÓDU DO ŘÍZENÉHO PROSTŘEDÍ PLATFORMY .NET FRAMEWORK

DODATEK: OBSAH CD „PŘECHÁZÍME NA .NET“ 194

1 Programovací jazyky společnosti Microsoft

Prashant Sridharan
Senior Product Manager
Microsoft Corporation

PŘÍNOSY PLATFORMY MICROSOFT .NET

Vývojová platforma Microsoft® .NET Framework je integrální součástí podporovaných operačních systémů Microsoft Windows® pro vytváření a běh nové generace aplikací a XML webových služeb. Zajišťuje vysoce produktivní, na prověřených standardech založené prostředí, v němž může společně koexistovat více programovacích jazyků pro snadnější vývoj různorodých počítačových aplikací. .NET Framework umožňuje vývojářům využívat již nabyté znalosti, protože výrazně usnadňuje proces programové integrace s již existujícími softwarovými aplikacemi, moduly či komponenty. Programátoři budou zcela jistě nadšeni také nepřehlédnutelnými inovacemi při vývoji a udržování internetových aplikací či při vytváření distribučních aplikačních jednotek. Vývojová platforma .NET Framework je postavena na dvou základních pilířích, jimiž jsou: společné běhové prostředí (Common Language Runtime, CLR) a jednotná a hierarchicky uspořádaná knihovna tříd, která zapouzdřuje třídy pro vývoj webových aplikací pomocí ASP.NET, inteligentních klientských aplikací pro Windows (pomocí Windows Forms) a databázových aplikací prostřednictvím subsystému ADO.NET.

Programátoři mohou psát své aplikace pro platformu .NET Framework v mnoha programovacích jazycích. Ať již použijete kterýkoliv z podporovaných jazyků, finálním výstupem kompilátoru bude vždy jazyková mezivrstva, která je tvořena kódem jazyka Microsoft Intermediate Language (MSIL). Kód jazyka MSIL je spravován společným běhovým prostředím, které vydá v příhodné chvíli pokyn na jeho překlad do nativního kódu, jemuž rozumí instrukční sada procesoru počítače. Jelikož výsledním produktem kompilátoru každého programovacího jazyka platformy .NET Framework je kód jazyka MSIL, aplikace napsaná v kterémkoliv jazyce může bez potíží spolupracovat s aplikací, která byla vytvořená v jiném programovacím jazyce. Poté, co bylo uvedeno společné běhové prostředí (Common Language Runtime), je jenom na programátorech, který programovací jazyk si z bohaté palety dostupných jazyků vyberou. Jednoduše řečeno, programátoři si teď mohou vybrat „ten svůj“ jazyk, a to plně podle svých preferencí a charakteru řešených úkolů.

Bohatá paleta programovacích jazyků

Podobně jako jsou umělci jedineční v používání materiálů a nástrojů, které vyhovují jejich zkušenostem a osobním preferencím, tak i softwaroví vývojáři uplatňují programovací jazyky v závislosti od svých znalostí a řešené problematiky. Žádný jazyk dosud nevyhověl potřebám všech vývojářů a programátorů. Po pravdě řečeno, programátoři jsou již od narození podivná stvoření: Zčásti jsou to umělci, zčásti vědci, pokaždé tvrdohlaví a věčně nespokojní. Přestože neustále hledají cesty k dosažení dokonalosti, jsou nuceni akceptovat jistou formu nedostatečnosti moderních programovacích jazyků. Pro některé z nich není problémem změnit vývojový nástroj jenom z rozmaru, zatímco jiní se snaží získat větší zkušenosti v rámci jednoho programovacího jazyka. Koneckonců, také Michelangelo byl talentovaným sochařem a brilantním malířem, kdežto Monet exceloval „pouze“ s olejovými barvami a plátnem. I tak jsou to oba géniové, kterých si vážíme.

„Vždycky plánujeme, že budeme mít půl milionu nebo 50 milionů oddaných programátorů ve Visual Basicu. Ale víte co? Máme Visual Basic v .NET. A teď máme i Javu v .NET. Dokonce máme i COBOL v .NET!“

– Tim Huckaby, President a CEO, Interknowlody

Výběr programovacího jazyka je v podstatě osobní volbou, která je ovlivněna větším počtem faktorů. Místo toho, abychom programátory nutili k osvojení jednoho jazyka před jiným, Microsoft nabízí platformu, na níž mohou různé programovací jazyky jako C++, Fortran, COBOL, Visual Basic® a Perl, společně prosperovat a umožňovat přístup k síle a flexibilitě platformy .NET Framework.

Co se programovacích jazyků týče, Microsoft nabízí čtyři zástupce s odpovídajícími vývojovými prostředími. Každý jazyk byl navržen tak, aby mohl oslovit vybranou školu programátorů. Jedná se o následující programovací jazyky:

- **Visual Basic .NET** - nejnovější verze světově nejpopulárnějšího vývojového nástroje a jazyka. Visual Basic .NET doručuje nepřekonatelnou produktivitu a jedinečné jazykové vlastnosti pro úkolově-orientované programátory, kteří budují svá řešení pomocí platformy .NET Framework.

- **Visual C++® .NET** - programovací nástroj, pro který je typická maximální síla a ovladatelnost. Pomocí jazyka C++ mohou programátoři, orientující se zejména na sílu jazyka, překlenout technologie různých platforem a sestavovat tak nativní aplikace pro Windows, jako i moderní řešení pro platformu .NET Framework ruku v ruce s maximálním výkonem a rozšířenou funkčností.

- **Visual C#® .NET** - moderní a průkopnický programovací jazyk a nástroj. C# byl představen v roce 2001 a již od první chvíle poskytoval známou jazykovou syntaxi pro programátory v C++ a Javě, společně s unikátními jazykovými konstrukcemi, jež nabídly vývojářům pohled na elegantnější stavbu aplikací pro platformu .NET Framework.

- **Visual J#® .NET** - programovací jazyk, jenž je zaměřen na programátory v Javě, kteří by rádi rozšířili své pole působnosti a vyvíjeli řešení pro platformu Microsoft .NET. Visual J# .NET zprostředkovává jazyku Java a současným vývojářům ve Visual J++ úplný přístup ke zdrojům platformy .NET Framework a pokročilým XML webovým službám, zatímco pořád zachovává známé jazykové konstrukce a syntaxi.

Tato kapitola se zaměřuje na bohatou paletu programovacích jazyků, které vytvořila společnost Microsoft a které jsou dostupné pro vývojáře a programátory. Společně s platformou Microsoft .NET mohou programátoři využívat již získané schopnosti a dovednosti, zatímco jim nic nebrání v ovládnutí nové vývojové platformy, nástrojů a programovacích jazyků. Cílem je pomoci všem vývojářům vyvíjet podmanivá řešení rychleji než kdykoliv předtím.

VISUAL BASIC .NET

http://msdn.microsoft.com/vbasic

Visual Basic 1.0 způsobil zásadní převrat ve vývoji aplikací pro operační systém Windows odstraněním vstupních bariér a demonstrací, která názorně předvedla, že široké masy programátorů mohou být ve své práci daleko produktivnější. Také nejnovější reinkarnace s názvem Visual Basic .NET staví na kladných vlastnostech všech svých předchůdců. Visual Basic .NET vsází na intuitivní syntaxi a uživatelské rozhraní, stejně jako na množství podpůrných nástrojů a migračních průvodců, kteří urychlují sestrojování aplikací napojených na platformu Microsoft .NET. Visual Basic .NET s výhodou využívá mimořádnou efektivnost vývojového procesu, což je vlastnost, jež byla již mnohokrát obhájena také stylem práce předcházejících verzí tohoto programovacího nástroje. Samozřejmě, nová edice Visual Basicu přidává ještě spoustu nových a ještě hodnotnějších programových funkcí, které dávají programátorům všech úrovní, od začátečníků až po zkušené vývojáře, možnost budovat aplikace pro Windows, Web a mobilní zařízení.

Vývoj orientovaný na úkoly

Závazné termíny nejsou ve světě softwarového průmyslu ničím neznámým. Ve skutečnosti jsou závazné termíny pro mnoho programátorů na denním pořádku. Tito programátoři bývají často pověřováni vývojem oportunistických aplikací, které řeší určité obchodní požadavky. Aplikace přitom vyžadují jenom malé množství plánování a rychlou cestu k tvorbě distribučních jednotek. V některých případech bývají takováto řešení pečlivě testována a odlaďována, zatímco jindy je primárním cílem pouhé sestavení aplikace a vytvoření příslušných aplikačních souborů určených k distribuci, čímž je úkol programátora splněn a ten se pak může vrhnout na další z nastávajících úkolů. Tato skupina vývojářů se označuje jako vývojáři orientovaní na úkoly, jejichž cílem je rychlé doručení vyvinutého řešení. Z tohoto důvodu je nutné, aby úkolově orientované vývojové nástroje kladly důraz na vysokou produktivitu práce, a to prostřednictvím vytvoření abstraktní vrstvy mezi programátorem a níže umístěnou vývojovou platformou. Programátor se poté může soustředit především na návrh aplikace, její vývoj a flexibilní rozmístění aplikace mezi cílovými skupinami uživatelů.

Programátorova volba

Visual Basic .NET představuje ideální variantu pro následující typy programátorů, kteří chtějí používat vývojovou platformu .NET Framework pro vytváření nové generace aplikací a softwarových služeb:

- **Programátoři hledající rychlý a produktivní vývojový nástroj pro platformu .NET Framework.** Microsoft Visual Basic .NET nabízí snadno pochopitelnou jazykovou syntaxi a intuitivní vývojové prostředí, které v notné míře pomáhá programátorům při návrhu, programování a sestavování jejich aplikací. Navíc, kolem Visual Basicu se vytvořila již velice početná vývojářská komunita s takřka nevyčerpatelnými zdroji informací, které jsou vývojářům ve Visual Basicu .NET plně k dispozici. Pomocí těchto informačních zdrojů mohou vývojáři citelně urychlit svůj start do tajů nové platformy Microsoft .NET.

- **Programátoři, kteří jsou znalí dřívějších verzí jazyka Visual Basic.** Visual Basic .NET staví na klíčových slovech, syntaxi a dalších známých elementech jazyka Visual Basic. Kupříkladu implicitní nerozlišování malých a velkých abecedních písmen či snadno pochopitelná jazyková syntaxe bude pro tradiční programátory ve Visual Basicu dobře známá již od prvních okamžiků v novém prostředí. Pokud jste investovali mnoho úsilí do vývoje řešení v předcházejících verzích jazyka Visual Basic, můžete přenést kód těchto projektů i do nové verze, a to pomocí vestavěného průvodce pro inovaci. Nicméně, většina existujících ActiveX® ovládacích prvků může být i ve Visual Basicu .NET nadále používána.

- **Vývojáři požadující známá paradigma, která se týkají vizuálního vývoje aplikací a inteligentní práce s editorem pro zápis programového kódu.** Mnohé principy vytváření vizuálního rozhraní aplikací v režimu jejich návrhu a psaní zdrojového kódu byly převzaty z předchůdců Visual Basicu .NET a dále rozšířeny. To znamená, že i v novém Visual Basicu se střetnete s vizuálním návrhem grafického rozhraní aplikací a s technologií IntelliSense®, která vám bude asistovat při psaní kódu. Pochopitelně, vývojáři jazyka nezapomněli ani na implementaci automatického formátování programového kódu, což je vskutku užitečná vlastnost, která zabezpečuje mnohem snazší čitelnost napsaného zdrojového kódu aplikací.

- **Programátoři, kteří by rádi budovali své aplikace pomocí přívětivějšího a přístupnějšího programovacího jazyka.** Visual Basic .NET byl zkonstruován tak, aby byl otevřený široké škále vývojářů, od nováčků až po experty. Začátečníci zanedlouho objeví mnoho vynikajících programových vlastností a charakteristik, které jsou stejně užitečné jako klíčová vylepšení prostředí Visual Basicu.

Jedinečné jazykové dovednosti

Programovací jazyk Visual Basic .NET obsahuje několik zcela výjimečných jazykových rysů a vlastností, které nelze najít v jiných programovacích jazycích společnosti Microsoft pro vývojovou platformu .NET. Zde je jejich výčet:

- **Implicitní inicializace proměnných.** Visual Basic .NET nepožaduje, aby byly proměnné a datové členy inicializovány před svým použitím. Začínající programátoři proto nebudou zbytečně uváděni do rozpaků zdánlivě tajemně vyhlížejícími požadavky, které můžete najít v jiných .NET programovacích jazycích.

- **Implicitní přiřazování datových typů proměnným a pozdní vázání objektů (late-binding).** Ve zdrojovém kódu jazyka Visual Basic .NET nemusí být explicitně specifikován datový typ proměnné před jejím použitím, což pomáhá programátorům psát užitečný kód již po krátkém tréninku.

- **Způsob práce enumerací.** Visual Basic .NET nabízí programátorům intuitivnější chování kódu při práci s enumeračními (výčtovými) typy.

- **Implicitně veřejný přístup.** Implicitně jsou vlastnosti a metody tříd napsaných v jazyce Visual Basic .NET veřejně přístupné. Tato skutečnost přispívá k přátelštější a intuitivnější programové syntaxi.

- **Použití sdílených členů.** Přístup ke sdíleným členům lze ve Visual Basicu .NET provést buď pomocí názvu třídy, nebo názvu instanční proměnné příslušného typu, jemuž sdílené členy patří. Zápis programového kódu je tím pádem snazší a názornější. Kupříkladu:

```
Dim x As New MyClass
x.SharedMethod() ' Pracuje stejně dobře jako kód na dalším řádku.
MyClass.SharedMethod()
```

- **Volitelné parametry.** Visual Basic .NET podporuje volitelné parametry, pomocí nichž mohou návrháři tříd pracovat flexibilněji při sestavování svých knihoven tříd. Volitelné parametry rovněž poskytují programátorům schopnost psát prospěšný kód, aniž by bylo nutné učit se všechny prvky objektově orientovaného programování.

- **Filtrované Catch bloky.** Visual Basic .NET zavádí novou a nutno přiznat, že značně efektivní strukturovanou správu chybových výjimek prostřednictvím filtrovaných **Catch** bloků. Filtrované **Catch** bloky umožňují vývojářům zachycovat a filtrovat chyby na základě třídy výjimky, jakéhokoliv podmíněného výrazu nebo explicitního identifikačního čísla chyby.

- **Parametrické vlastnosti.** Vlastnosti ve Visual Basicu .NET mohou obsahovat parametry, a tudíž jsou flexibilnější než jejich protějšky z C#.

- **Deklarativní zpracovatele událostí.** Zpracovatele událostí ve Visual Basicu .NET mohou deklarovat události, na které mohou reagovat pomocí klíčového slova **Handles**.

- **Redeklarace členu rozhraní.** Visual Basic .NET dovoluje programátorům přejmenovat člen rozhraní při jeho implementaci ve třídě.

Unikátní dovednosti prostředí Visual Basicu .NET

Visual Basic .NET kromě početních jazykových rozšíření přichází také s klíčovými inovacemi svého prostředí, které nyní vychází programátorům a vývojářům ještě více vstříc a umožňuje jim navrhovat a psát okouzlující aplikace a softwarové služby. Tyto inovační prvky jsou dostupné výhradně ve Visual Basicu .NET:

- **Kompilace v pozadí.** Kompilace v pozadí pracuje permanentně za scénou a tiše kompiluje všechen programový kód, který zadáte do editoru. Programátoři ve Visual Basicu .NET tak mohou okamžitě vědět, zdali je zapsaný kód bez chyb, nebo se v něm nacházejí nějaké nesrovnalosti.

- **Pokročilé formátování kódu.** Editor pro zápis programového kódu může (volitelně) automaticky formátovat zapsaný kód, čímž výrazně šetří váš čas. Pokročilé formátování si dovede poradit s automatickým zarovnáváním fragmentů zdrojového kódu, se změnou velkých či malých abecedních znaků v názvech klíčových slov a proměnných či dokonce s přidáním chybějícího návěstí Then v příkazu If a mnoha dalšími operacemi.

Výkon

Finální oblastí s velkou mírou důležitosti je výkon. Kompilátor Visual Basicu .NET generuje výslední kód v podobě kódu jazyka Intermediate Language (IL), který je, co se týče výkonu, plně srovnatelný s kódem, jenž je generován kompilátorem jazyka C#.

VISUAL C++ .NET

http://msdn.microsoft.com/visualc

Programátoři, pro které je podstatný především výkon jejich aplikací, budou zcela určitě potřebovat vytěžit z vývojové platformy všechny její schopnosti, a to při maximálním možném výkonu. Navzdory tomu, že platforma .NET Framework a společné běhové prostředí přinášejí mnoho výhod, někteří programátoři budou i nadále vytvářet aplikace, jejichž činnost bude využívat veškeré pokročilé programové funkce operačního systému Windows. Tento segment vývojářů pracuje již tradičně s jazykem Visual C++, pomocí něhož mohou dosáhnout preciznější kontroly nad během aplikací. Programátoři mohou aplikovat různé metody, jako je například hloubková optimalizace programového kódu a velice efektivní přístup k systémovým službám, jako je třeba přístup do operační paměti či do souborové struktury pevného disku. Visual C++ .NET v tomto směru následuje svého předchůdce a také nabízí softwarovým odborníkům přímý přístup k aplikačnímu programovému rozhraní Win32® API. Visual C++ ovšem obsahuje také přidanou hodnotu v podobě zcela neomezeného přístupu k bohaté nabídce dovedností vývojové platformy .NET Framework a řízení aplikací prostřednictvím společného běhového prostředí.

Vývoj orientovaný na výkon

Existuje opravdu nepřeberný počet situací, kdy musí programátoři sáhnout po vybraných fundamentálních programových funkcích operačního systému. Z historického hlediska uváděla společnost Microsoft na trh spektrum programovacích nástrojů, z nichž některé prováděly abstrakci od funkcí operačního systému v plném rozsahu, zatímco jiné doručovaly ničím nespoutaný přístup ke všem programovým zákoutím operačního systému. V současné době je situace taková, že vývojová platforma .NET Framework nabízí řadu aplikačních programových rozhraní pro programování robustních podnikových řešení, ovšem přesto nezajišťuje přístup k veškerým funkcím, které můžete najít v operačních systémech řady Microsoft Windows. Výkonově orientované vývojářské nástroje berou v potaz požadavek vývojářů na maximální výkon aplikací, a proto přicházejí s pokročilými jazykovými konstrukcemi, knihovnami tříd a prostředími navrženými tak, aby byl zaručen komfortní vývoj plné škály řešení, která odpovídají požadavkům zákazníků.

Programátorova volba

Visual C++ .NET je vhodným vývojovým nástrojem pro mnoho programátorů:

- **Programátoři, kteří si přejí budovat aplikace a komponenty založené na Win32 API.** I v dnešních dobách moderních metod a technik vývoje softwaru žijí programátoři, kteří stále potřebují vyvíjet nativní aplikace pro systém Windows. Tato skupina programátorů používá při své práci aplikační programové rozhraní Win32 API a nativní knihovny tříd jazyka C++ pro dosažení mimořádného výkonu a funkčnosti. Visual C++ .NET 2003 uvádí některé optimalizace vestavěného kompilátoru, které mohou podpořit programátory při dosahování ještě větších výkonnostních efektů v kýžených typech aplikací.

- **Programátoři, kteří by rádi překlenuli mezeru mezi nativními aplikacemi a aplikacemi vyhovujícími standardům platformy .NET Framework.** Existující aplikace mají často složitou programovou strukturu, vytvoření které stálo mnoho času, energie a finančních prostředků. Z mnoha důvodů není možné, aby byly tyto aplikace okamžitě přepsány pro běh na platformě .NET Framework. Pomocí nástroje Visual C++ mohou vývojáři rozšířit existující aplikace nebo přidat nový programový kód, jehož exekuci bude mít na starosti .NET Framework. Přitom však stále mohou přistupovat i k těm nejpokročilejším funkcím Windows API. Aktivovat programové funkce Win32 API mohou programátoři také z jiných jazyků jako je C# či Visual Basic, ovšem ani jeden z uvedených konkurentů nedisponuje vrozenou schopností C++ pro interoperabilitu s existujícím kódem systému Windows.

- **Vývojáři zaměřující se v první řadě na výkonnostní charakteristiky svých aplikací.** Jazyk C++ poskytuje vývojářům nejvyšší stupeň kontroly při návrhu a běhu programů. Zkušení programátoři mohou použitím jazyka C++ navrhovat a implementovat aplikace, jejichž kód je prováděn rychleji a efektivněji ve srovnání s tím, kdyby byly tyto aplikace napsány v jiných programovacích jazycích. To se týká jednak nativních aplikací určených pro operační systém Windows, jako i řízených .NET aplikací.

- **Programátoři, kteří chtějí vyvíjet skutečná meziplatformní řešení.** Jedině programovací jazyk C++ zahrnuje ISO standardizovanou a skutečně přenositelnou programovou syntaxi, kterou lze portovat na prakticky každý operační systém. Visual C++ .NET 2003 se daleko více přibližuje k stanoveným syntaktickým standardům, než tomu bylo kdykoliv předtím. Programátoři mohou těžit z pokročilých jazykových prvků a výhod, které nabízejí knihovny tříd pro širokou paletu rozmanitých operačních systémů.

Jedinečné jazykové dovednosti

Visual C++ .NET zahrnuje množství jedinečných jazykových rysů a dovedností, které byly do jazyka zařazeny pro uspokojení vysokých nároků náročných programátorů a softwarových expertů. Tyto jazykové schopnosti přispívají ke skutečnosti, že C++ je nejmocnější ze všech programovacích jazyků softwarového produktu Visual Studio společnosti Microsoft. Na následujících řádcích je uvedena charakteristika elementů, které dělají jazyk C++ tak výjimečným:

- **Šablony**. Šablony jsou charakteristické několika jazykovými rysy v době kompilace programu, které jsou dostupné jenom v C++. Používání šablon představuje dobrý krok pro zabezpečení znovupoužitelnosti a lepší výkonnosti programového kódu.

- **Ukazatele**. Pomocí ukazatelů mohou programátoři získat přímý přístup do různých částí operační paměti počítače. Ukazatele jsou nepostradatelnou programovací pomůckou, prostřednictvím které lze maximalizovat rychlost exekuce programových instrukcí aplikací.

- **Vícenásobná dědičnost.** Jazyk C++ uvádí programátory do vzrušujícího světa objektově orientovaného programování (OOP) tím, že implementuje všechny důmyslné prvky a konstrukce této filozofie programování.

- **Rozšiřující sada vnitřních API (intrinsics).** Vývojáři mohou nyní pracovat s klíčovými prvky platformy, které nejsou dostupné přes standardní programovací techniky. Příkladem je aktivace vnitřních instrukcí sady MMX nebo AMD 3DNow!.

- **Atributy v době kompilace programu.** Atributy v C++ reprezentují snadno dosažitelné prostředky pro vytváření vysoce odladěných opakujících se segmentů programového kódu využitím jednodušší a robustnější jazykové syntaxe.

Unikátní vlastnosti vývojového prostředí

Visual C++ .NET 2003 je obdařený spoustou inteligentních rysů vývojového prostředí, které asistují programátorovi při vytváření flexibilních a výkonných aplikací:

- **Optimalizovaný kompilátor.** Kompilátor ve Visual C++ dovede vyladit aplikace podle různých scénářů, mezi nimiž nechybí optimalizace pro cílová běhová prostředí, optimalizace provádění kalkulací s čísly s pohyblivou řádovou čárkou či optimalizace generování kódu jazyka MSIL.

- **Kontrola bezpečnosti kódu za běhu programu.** Programátoři mohou psát bezpečnější nativní aplikace pro operační systémy Windows za pomoci pokročilých vlastností kompilátoru, které pomáhají ochraňovat aplikace před škodlivými útoky.

- **Podpora 32 a 64 bitů.** Kompilátory pro Visual C++ jsou dostupné pro velké množství hardwarových platforem, mezi nimiž nechybí 32 a 64bitové mikroprocesory společností Intel a AMD jako i další mikroprocesorová zařízení. Psaní skutečně škálovatelných aplikací je tedy docela snadné.

- **Pokročilé hlášení programových chyb.** Není žádnou novinkou, že počítačové aplikace jsou náchylné na výskyt programových chyb. Visual C++ dovoluje vývojářům snadnější identifikaci a korekci chyb, a to dokonce i v distribučních jednotkách aplikací pomocí technologie Minidump.

- **Vyspělé techniky odlaďování aplikací.** Odlaďovací nástroje Visual Studia a Visual C++ jsou garancí současného odlaďování nativního i řízeného zdrojového kódu.

V budoucích verzích programovacího nástroje Visual C++ .NET se vývojáři mohou těšit na další nepostradatelná vylepšení a nové inovativní techniky, které posunou pomyslnou laťku pokroku ještě o kousek výše:

- **Generics**. Nová technologie, která nabídne opětovné použití parametrizovaných algoritmů za běhu programu.

- **Šablony řízených typů.** Nové šablony s sebou přinesou schopnost používat šablonovou syntaxi jazyka C++ s řízenými typy.

- **Bohatší podpora pro vytváření assembly, která budou těsněji provázaná s pravidly společné jazykové specifikace (Common Language Specification, CLS).** Kompilátor jazyka C++ bude umožňovat programátorům označovat CLS-nekompatibilní typy a neověřitelný programový kód.

VISUAL C# .NET

http://msdn.microsoft.com/vcsharp

Programovací jazyky Visual Basic a Visual C++ stály odjakživa na rozdílných stranách spektra vývojářů. Visual Basic kladl důraz především na vysokou produktivitu práce a nabízel programátorům co možná nejlehčí a nejpřátelštější vývoj aplikací, a to i za cenu poněkud omezeného přístupu k systémovým zdrojům. Na druhé straně, Visual C++ šel takřka na hranice možností, protože dovoloval programátorům převzít kontrolu nad systémem plně do svých rukou, i když někdy bylo nutné obětovat notnou dávkou pracovní produktivity.

Společnost Microsoft se rozhodla jednou a provždy překlenout pomyslný most mezi oběma uvedeným vývojovými prostředky, a proto navrhla a vyvinula moderní, flexibilní a inovativní programovací jazyk s názvem C#. C# je vhodným nástrojem pro vývojáře, kteří se soustřeďují zejména na eleganci a efektivnost napsaného zdrojového kódu. Ve skutečnosti budou konstrukce a syntaxe jazyka C# okamžitě známé všem vývojářům v C++. Přestože je C# dalším vývojovým stupínkem v evoluci jazyků C/C++, produktivitou své práce se může rovnat i s Visual Basicem.

Vývoj orientovaný na programový kód

Zcela jistě se shodneme na tom, že všichni programátoři musí při plnění svých projektů psát mnoho řádek programového kódu. Psaní kódu ovšem není jedinou činností, kterou je programátor živ. Kromě zadávání příkazů a klíčových slov tráví programátoři spoustu času také používáním průvodců, ovládacích prvků a návrhářů s cílem zvýšit produktivitu a efektivnost své práce v rámci návrhu a vývoje softwarových aplikací. Negativním efektem programátorů specializujících se na kód je skutečnost, že mohou postupně ztrácet kontrolu nad vším tím kódem, jenž vyplňuje jejich programy. Tito programátoři dokonce tak lpí na kódu, jenž vytvoří, že po jistém čase začnou věřit „svému“ kódu více než kódu, který byl připraven průvodci vývojového prostředí. A i když v některých okamžicích přece jenom sáhnou po průvodci či návrháři, pořád se budou snažit modifikovat vygenerovaný kód podle svých představ.

Kromě toho, programátoři zaměřující se na kód mají tendenci k psaní kódu, jenž bude opětovně použitelný jinými vývojáři, dokonce i takovými, kteří nemají úplné znalosti jejich vývojových praktik. Vývojová řešení a knihovny tříd, které jsou výsledkem práce těchto programátorů, vznikají především v editorech zdrojových kódů a ve většině případů postrádají implementaci pečlivější navrhovací fáze.

Programátorova volba

Pokud hledáte programovací jazyk vývojové platformy .NET Framework pro vytváření nové generace aplikací a služeb, C# je pro vás tou správnou volbou. Po pravdě řečeno, jazyk C# je vhodný pro všechny následující segmenty softwarových odborníků:

- **Programátoři hledající produktivní programovací jazyk z rodiny jazyků C/C++.** C# přichází s programovou syntaxí, která bude již na první pohled známá všem vývojářům, kteří jsou znalí jazyků C a C++. C# má však ve své nabídce daleko více skvělých prvků, mezi které patří podpora přetěžování operátorů, práce s enumeračními typy, rozlišování malých a velkých abecedních znaků, práce s vlastnostmi, událostmi, atributy, delegáty a další. Ruku v ruce s platformou .NET Framework jsou známé, ale i nově zavedené prvky jazyka rychle pochopitelné pro programátory v C++. Ti tak mohou okamžitě okusit vyšší efektivnost práce, lepší ovladatelnost a v neposlední řade také signifikantní přínosy v oblasti bezpečné exekuce zdrojového kódu.

- **Návrháři systémů a softwaroví architekti.** Portfolio programových schopností jazyka C# je doslova přeplněno úžasnými rysy, mezi něž bezpochyby patří přetěžování operátorů či dokonce práce s nezabezpečeným kódem, pomocí něhož je možné komunikovat s aplikacemi staršího data. Všechny tyto výtečné vlastnosti dělají z jazyka C# kandidáta pro vývoj komplexních podnikových systémů a knihoven tříd. Inovace lze však očekávat i do budoucna: Nové prvky jako Generics a iterátory budou i nadále přispívat k tomu, abyste mohli pomocí jazyka C# vyvíjet mohutná podniková řešení na platformě .NET Framework.

- **Programátoři, kteří již investovali mnoho prostředků do vývoje softwarových produktů pomocí jazyka Java.** Patříte k programátorům s mnoha aplikacemi naprogramovanými v Javě, kterým jste věnovali mnoha času, energie a finančních prostředků? Chtěli byste svá stávající řešení portovat na platformu .NET Framework? Jestliže ano, programovací nástroj Visual C# .NET 2003 vám může podat pomocnou ruku. Součástí nástroje je totiž průvodce Java Language Conversion Assistant (JLCA), jenž je schopný provést rychlou konverzi předmětných fragmentů programového kódu z Javy do C#. JLCA provádí pečlivou analýzu zdrojového kódu jazyka Java, který vzápětí převádí do kódu jazyka C#. Pokud dojde při konverzním procesu k potížím, jsou problémové partie kódu zřetelně označeny, takže programátoři je mohou rychle najít a dokončit konverzní proces.

C#: Programovací jazyk s mnoha prvky jazyka C++

Je to skutečně tak. Programovací jazyk C# přináší mnoho skvělých vlastností z dnes již tradičního jazyka C++ i do prostředí moderní vývojové platformy .NET Framework. Ano, spousta programových elementů a konstrukcí se podobá těm, které znáte z C++. V prostředí jazyka C# však můžete k těmto prvkům přistupovat s daleko větší efektivností a produktivitou práce. Na následujících řádcích se dozvíte více:

- **Podpora všech typů společného běhového prostředí.** Programovací jazyk C# dokáže pracovat se všemi typy, které může společné běhové prostředí použít.

- **Předávání argumentů odkazem a výstupní parametry.** Jestliže se rozhodnete pro C#, budete moci předávat proměnné funkcím odkazem anebo dokonce vytvářet výstupní parametry, které je nutno inicializovat před ukončením práce funkce, v níž jsou definovány.

- **Přetěžování operátorů.** Vývojáři řešení mohou vytvářet robustnější knihovny tříd pomocí techniky zvané přetěžování operátorů.

- **Příkaz using.** Aplikací příkazu using mohou programátoři lépe kontrolovat jak jejich aplikace hospodaří se zdroji.

- **Nezabezpečený kód.** Jazyk C# dovoluje vývojářům pracovat s ukazateli a manipulovat tak s vybranými segmenty operační paměti. Ačkoliv nezabezpečený kód je pořád realizován v rámci společného běhového prostředí, jeho použití umožňuje programátorům preciznější ovladatelnost využití systémové paměti. Navzdory tomu, pokud si přejete získat maximální možnou kontrolu nad operační pamětí, lépe vám bude pravděpodobně vyhovovat Visual C++.

- **XML dokumentace.** Programátoři v C# mohou opatřit vybrané partie zdrojového kódu poznámkami jazyka XML.

Je zcela zřejmé, že programovací jazyk C# je vskutku životaschopnou alternativou ve světě programování profesionálních aplikací. Dokonce jde o tak rychle se rozvíjející technologii, že již začíná přesahovat meze svého původního návrhu. Vývojáři a designéři jazyka C# plánují do budoucích verzí zařadit množství dalších zajímavých prvků a programových konstrukcí, které budou dále rozvíjet moderní a inovativní prostředek pro vývoj všemožných typů aplikací. A co se očekává?

- **Generics**. Obdoba šablon z C++, které výrazně přispějí k lepšímu opětovnému použití již jednou napsaného zdrojového kódu.

- **Iterátory**. Programová konstrukce, prostřednictvím které budete moci rychleji a snadněji pracovat s datovými kolekcemi.

- **Anonymní metody.** Zapojíte-li do kódu anonymní metody, můžete vykonávat jednoduché úkoly pomocí delegátů.

- **Parciální typy.** Implementace parciálních typů nabídne programátorům možnost rozdělit programový kód do více souborů.

VISUAL J# .NET

http://msdn.microsoft.com/vjsharp

Společnost Microsoft přivedla na svět také programovací jazyk Visual J# .NET, jehož cílem je nabídnout syntaxi známého programovací jazyka Java i pro vývojovou platformu .NET Framework. Visual J# .NET přijde vhod programátorům, studentům a profesorům, kteří mohou budovat svá řešení pro platformu .NET Framework, aniž by byli nuceni vzdávat se již zažitého programovacího stylu, jenž znají z Javy. Visual J# .NET ovšem oslovuje také mnohočetnou skupinu programátorů, kterým přírostl k srdci programovací nástroj Visual J++ 6.0. I tito vývojáři mohou využít svých dosavadních znalostí a začít s tvorbou nové generace aplikací a softwarových služeb pro moderní vývojovou platformu.

Vývoj v jazyce Java

Programovací jazyk Java vyřešil více problémů, kterým byli vývojáři v C++ nuceni často čelit, a to při zachování jednodušší syntaxe a známých prvků objektově orientovaného programování. Pro všechny vývojáře, kteří jsou znalí jazyka Java, je přechod na Visual J# .NET naprosto plynulý a po krátkém seznámení se s prostředím se zde budou cítit jako doma. Výběr jazyka Visual J# .NET navíc znamená, že programátoři mohou využívat všechny vzrušující vlastnosti, které jim .NET Framework nabízí.

Programátorova volba

Programovací nástroj Visual J# .NET je nepostradatelný pro následující typy počítačových specialistů:

- **Stávající programátoři pracující v jazyce Java.** Je zcela samozřejmé, že programátoři, kteří jsou velmi dobře obeznámení s jazykem Java, se nebudou chtít pouštět do studia nového programovacího jazyka. Visual J# .NET staví na dosavadních zkušenostech vývojářů v Javě a výrazně tak minimalizuje investice nutné pro přechod na platformu .NET Framework.

- **Programátoři, kteří vynaložili prostředky na vývoj aplikací ve Visual J++.** Visual J# .NET dokáže konvertovat projekty vytvořené v prostředí Visual J++. Takto upravené projekty je okamžitě možné dále rozšiřovat a obohacovat o novou funkcionalitu dostupnou na platformě .NET Framework.

- **Studenti, učitelé a profesoři.** Programovací nástroj Visual J# .NET mohou využívat studenti, učitelé a profesoři pro výučbu základů počítačových věd. Visual J# .NET vyhovuje požadavkům zkoušky Advanced Placement Computer Science.

Jedinečné jazykové vlastnosti

Pro vývojáře pracující s jazykem Java bude přechod na J# velice snadný díky zařazení mnoha známých prvků a programových konstrukcí, které napomáhají tomu, aby se práce v jazyce J# stala synonymem pro komfortní a příjemný vývoj aplikací běžících pod křídly platformy .NET Framework. J# přichází s několika želízky v ohni:

- **Syntaxe jazyka Java.** Pro programátory v Javě je nejdůležitějším zjištění, že v prostředí J# naleznou jazykovou syntaxi, která je jim dobře známá. Dalším pozitivem je neomezený přístup ke zdrojům platformy .NET Framework.

- **Podpora knihoven tříd.** V jazyce J# mohou vývojáři pracovat s nezávisle vyvinutými knihovnami tříd, které poskytují takřka veškerou funkcionalitu knihoven tříd z Java Development Kit (JDK) verze 1.1.4 a také mnoho tříd z balíčků JDK 1.2 java.util.

- **Vlastnosti, delegáti a události.** Do programovacího jazyka J# byla začleněna podpora stěžejných prvků platformy .NET Framework jako jsou vlastnosti, delegáti či události. Kromě toho byla zachována soudržnost s tradiční syntaxí jazyka Java.

- **Javadoc komentáře.** J# si dokáže poradit s komentáři fragmentů zdrojového kódu podle stylu Javadoc. Visual J# .NET obsahuje nástroj, jenž umožňuje uživatelům vyvolat HTML API, které dokáže generovat dokumentaci z určených Javadoc komentářů.

Unikátní vlastnosti vývojového prostředí

Programovací jazyk Visual J# .NET je plně integrován do vývojového prostředí Visual Studia .NET, což umožňuje vývojářům v J# využívat vestavěné návrháře, odlaďovací nástroje či editory s pokročilými vlastnostmi. Rovněž byly vytvořeny také nové pomůcky, jejichž cílem je podat vývojářům pomocnou ruku při portaci jejich řešení na platformu Microsoft .NET:

- **Průvodce inovací pro Visual J++.** Programátoři používající Visual J++ mohou inovovat své projekty směrem k Visual J# .NET. Průvodce inovací provádí konverzi projektových souborů a nabízí uživatelům kontextuální nápovědu v případě potíží.

- **Binární konvertor.** Tento nástroj realizuje konverzi souborů s bajtovým kódem Javy do podoby assembly, kterou lze použít v .NET aplikacích.

SHRNUTÍ

Programovací jazyky mohou být využívány pro budování široké škály programových řešení. Každý jazyk je vybaven svými jedinečnými vlastnostmi, které z něj dělají výtečně navržený celek pro přípravu rozmanitých typů aplikací. Společnost Microsoft nabízí programátorům, vývojářům a softwarovým architektům paletu čtyř programovacích jazyků, které společně představují přístup k vývojové platformě .NET Framework pro mnoho milionů uživatelů.

2 Proč ASP.NET?

Microsoft Corporation

Shrnutí

Vytváření dynamických a vysoce výkonných webových aplikací nebylo ještě nikdy jednodušší. ASP.NET kombinuje doposud nevídanou vývojářskou produktivitu s výkonem, spolehlivostí a efektivností instalace.

Vývojářská produktivita

ASP.NET umožňuje dodávat webové aplikace pro skutečné nasazení v rekordním čase.

- **Snadný programovací model.** S pomocí technologie ASP.NET lze vytvářet opravdové webové aplikace snadněji než kdykoliv předtím. Serverové ovládací prvky ASP.NET dovolují uplatňovat styl programování podobný HTML, který vás přesvědčí o tom, že budování podmanivých stránek je možné nyní dosáhnout s mnohem menší porcí kódu, než jak tomu bylo u tradičního ASP. Zobrazování dat, kontrolování uživatelských vstupů a přenášení souborů na server, to vše jsou činnosti, které dokážete provést s grácií a lehkostí. Ovšem ze všeho nejlepší je to, že ASP.NET stránky se správně zobrazují ve všech prohlížečích včetně Netscapu, Opery, AOL a Internet Exploreru.

- **Nové možnosti ve výběru a použití programovacích jazyků.** Když vsadíte na technologii ASP.NET, můžete opětovně využít investice, které jste vložili do studia programovacího jazyku vaší volby. Na rozdíl od ASP, které podporuje jenom interpretovaný VBScript a JScript, ASP.NET je otevřeno více než pětadvaceti programovacím jazykům (včetně vestavěné podpory pro Visual Basic .NET, C# a JScript .NET), což přináší dosud nepoznanou flexibilitu při výběru kýženého programovacího jazyka.

- **Výtečná podpora nástrojů.** ASP.NET můžete ovládnout pomocí jakéhokoliv textového editoru, dokonce i Poznámkového bloku! Když ale zapojíte Visual Studio .NET, vývoj webových aplikací bude velice podobný práci ve Visual Basicu. Nyní můžete vizuálně navrhovat ASP.NET webové formuláře prostřednictvím intuitivních technik. Rovněž se můžete těšit na plně kvalifikovanou podporu při psaní zdrojového kódu včetně automatického dokončování příkazů a barevného odlišování specifických partií kódu. Visual Studio .NET kromě jiného nabízí také vestavěnou podporu pro odlaďování a rozmísťování ASP.NET webových aplikací.

  Verze Enterprise sady Visual Studio .NET je dodávána s dodatečnými nástroji, které vám pomohou se správou životních cyklů vyvíjených aplikací. V praxi to znamená, že můžete vytvářet organizační plány, provádět analýzy, návrhy a testy a koordinovat vývojové týmy, které se zabývají vytvářením ASP.NET webových aplikací. K dispozici je vám modelování vztahů mezi třídami pomocí jazyka UML a vytváření databázových modelů (konceptuální, logické a fyzické modely). Využít můžete také testovací nástroje pro testování funkčnosti, výkonu a škálovatelnosti. Kromě toho jsou pro vás připraveny i podniková řešení a šablony, a to vše v jednotném vývojovém prostředí Visual Studia .NET.

- **Široký rámec tříd.** Techniky, které byly v dřívějších časech obtížně proveditelné, nebo vyžadovaly komponenty třetích stran, mohou být v prostředí platformy .NET Framework realizovány pomocí několika řádků programového kódu. .NET Framework je kolekcí více než 4500 tříd, které zapouzdřují funkcionalitu potřebnou pro práci s XML či databázemi. Možností potenciálního využití je však daleko více. Třídy mohou být využity třeba pro přenášení souborů směrem k serveru, provádění operací s regulárními výrazy, ke generování obrazových map, pro sledování a zaznamenávání výkonnostních charakteristik systémů, pro zpracovávání fronty zpráv či zasílaní elektronických zpráv.

Vylepšená výkonnost a škálovatelnost

Prostřednictvím technologie ASP.NET můžete obsloužit více uživatelů, a to při stejných hardwarových nárocích.

- **Kompilovaná exekuce.** ASP.NET je mnohem rychlejší než tradiční ASP, ovšem pořád pracuje s efektivním aktualizačním modelem, jenž je známý z ASP. Rozdíl je v tom, že nyní není vyžadována žádná explicitní kompilace. ASP.NET zcela automaticky zjistí přítomnost jakýchkoliv změn, provede dynamickou kompilaci potřebných souborů a uchová kompilované výsledky pro pozdější opětovné použití. Dynamická kompilace zaručuje, že vaše aplikace je vždy aktuální a díky kompilované exekuci i rychlá. Mnohé aplikace, které byly přeneseny z ASP, dokázaly obsloužit tři až pětkrát tolik požadavků jako jejich starší protějšky.

- **Podpora výstupní vyrovnávací paměti.** Výstupní vyrovnávací paměť technologie ASP.NET může dramaticky zvýšit výkonnost a škálovatelnost vaší aplikace. Když je aktivní výstupní vyrovnávací paměť při práci s webovou stránkou, ASP.NET tuto stránku zpracovává pouze jednou, přičemž uloží výsledky své činnosti v paměti a pošle je klientovi. Jestliže jiný uživatel zašle požadavek pro získání stejné stránky, ASP.NET vezme uloženou podobu stránky z paměti, aniž by bylo nutné opětovné zpracovávání webové stránky. Výstupní vyrovnávací paměť je konfigurovatelná a může být použita pro ukládání individuálních regionů na stránce, nebo přímo celé stránky. Používání výstupní vyrovnávací paměti může radikálně přispět k navýšení výkonu při práci s daty řízenými webovými stránkami. Není totiž nutné, aby byla databáze dotazována při zpracování každého požadavku od uživatele.

- **Řízení stavu aplikací ve webových farmách.** ASP.NET dovoluje sdílet uživatelská data napříč všemi počítači, které vytvářejí webovou farmu. Nyní může uživatel získat přístup k potřebným datům pomocí zaslání požadavku na kteroukoliv počítačovou stanici ve webové farmě. Použití komponentů s obchodní logikou, které byly vytvořeny na platformě .NET Framework je nyní, co se týče správy vláken, bezproblémové, takže se již není nutno obávat potenciálních konfliktů.

- **Microsoft .NET překonává J2EE.** Při přímém porovnání výkonnosti a škálovatelnosti mezi dvěma implementacemi aplikace s názvem Pet Store, které byly připraveny pomocí J2EE firmy Sun a ASP.NET společnosti Microsoft, bylo zjištěno, že platforma Microsoft .NET výrazně překonává řešení J2EE. Několik rozhodujících faktorů: ASP.NET webová aplikace vyžadovala jenom čtvrtinu zdrojového kódu, byla 28krát rychlejší (což činí 2700%) a byla schopna vyřídit více než sedmkrát tolik uživatelských požadavků než její protějšek v J2EE. A to vše při šestkrát menší zátěži procesoru.

Chcete-li si prohlédnout výsledky měření, stáhnout kód nebo spustit .NET verzi aplikace Pet Store, navštivte http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/bdasamppet.asp?frame=true

Vylepšená spolehlivost

Technologie ASP.NET garantuje, že vaše aplikace bude vždy pro vaše zákazníky dosažitelná.

Ochrana proti vzniku paměťových děr, mrtvých bodů a zhroutění aplikací. ASP.NET automaticky odhaluje a zotavuje se z chyb, jakými jsou paměťové díry a mrtvé body, takže vaše aplikace bude vždy schopna odpovídat na dotazy uživatelů.

Povězme, že vaše aplikace vytváří malou paměťovou díru, přičemž tato díra se týden co týden zvětšuje, což výustí až do alokace značné části virtuální paměti serveru. Technologie ASP.NET odhalí tuto nebezpečnou situaci a automaticky spustí další instanci pracovního procesu ASP.NET aplikace a přesměruje všechny nové požadavky na tuto nově vytvořenou instanci aplikace. Když stará instance aplikace ukončí zpracovávání čekajících požadavků, je elegantně uvolněna z paměti a paměťová díra je zlikvidována. ASP.NET webová aplikace se takto zcela samostatně a plně automaticky dokázala zotavit ze vzniklé chyby, a to bez zásahů ze strany administrátora či přerušení činnosti aplikace.

Snadné rozmísťování aplikací

ASP.NET odstraňuje všechna bolestivá místa, která způsobovala potíže při procesu rozmísťování webových aplikací.

- **Rozmísťování aplikací je nyní rychlé a efektivní.** ASP.NET dělá instalaci aplikací mnohem jednodušší, než tomu bylo dříve. Skutečně, pomocí ASP.NET můžete provést rozmístění aplikace tak snadno, jako kdybyste pracovali pouze s jedinou HTML stránkou: jednoduše ji překopírujete na server. Již nemusíte spouštět program regsvr32 pro registraci potřebných komponent, protože všechna konfigurační nastavení jsou uložena v XML souboru, jenž je součástí souborové struktury vaší aplikace.

- **Dynamická aktualizace spuštěných aplikací.** Plusem technologie ASP.NET je, že vám dovoluje aktualizovat kompilované komponenty, aniž byste byli nuceni restartovat webový server. V dřívějších dobách tradičních COM komponent museli vývojáři restartovat webový server pokaždé, když bylo zapotřebí provést aktualizaci aplikace. Ve společnosti ASP.NET se budete cítit mnohem lépe: Vše, co musíte udělat, je nahradit knihovnu DLL stávající komponenty - ASP.NET automaticky rozezná změnu a začne používat nový aplikační kód.

- **Snadná inovace.** Abychom si rozuměli, nikdo vás nenutí, abyste převáděli vaše existující aplikace, když chcete používat ASP.NET. ASP.NET běží na internetovém informační serveru (IIS) vedle klasické ASP, a to na operačních systémech Windows 2000 a Windows XP. Vaše stávající ASP aplikace budou i nadále využívat dynamicky linkovanou knihovnu ASP.DLL, zatímco exekuci nových ASP.NET aplikací si vezme na starost nová mašinérie technologie ASP.NET. Samozřejmě, můžete realizovat migraci celých aplikací, nebo jen několika webových stránek. ASP.NET vám dokonce umožňuje pokračovat v používání tradičních obchodních COM komponent.

Nové aplikační modely

Pokud se rozhodnete pro technologii ASP.NET, můžete si být jisti, že vaše aplikace osloví širší spektrum vašich partnerů a zákazníků.

- **XML webové služby.** Prostřednictvím XML webových služeb mohou vaše aplikace komunikovat a sdílet data přes Internet, bez ohledu na použitý operační systém nebo programovací jazyk. Pomocí ASP.NET je vystavování a volání XML webových služeb docela jednoduché.

  Jakákoliv třída může být převedena do podoby XML webové služby zapsáním pouhých několika řádků programového kódu. Služba pak může být aktivována libovolným SOAP klientem.

  Zasílaní požadavků XML webové službě z vaší aplikace je až neuvěřitelně snadné. Přitom nemusíte disponovat žádnými znalostmi fungování sítí, XML nebo SOAP.

- **Podpora mobilních zařízení.** ASP.NET Mobile Controls nabízejí efektivní programovatelnost mobilních telefonů, personálních digitálních asistentů (PDA) a dalších více než 80 mobilních zařízení. Stačí, když aplikaci napíšete pouze jednou a vestavěné podpůrné nástroje a komponenty se postarají o automatické vygenerování WAP/WML, HTML nebo iMode podle požadavků cílového zařízení.

3 Přechod z ASP na ASP.NET

Autor: Scott Michell, 4GuysFromRolla.com
Technický editor: Paul Litwin, Litwin Consulting

Shrnutí

Tato kapitola začíná podrobným přehledem typické daty řízené ASP aplikace, a poté se zaměřuje na proces migrace této aplikace do prostředí ASP.NET.

Plánované úkoly

- Spuštění ASP a Microsoft ASP.NET na stejném webovém serveru
- Charakterizování běžné ASP aplikace
- Migrace ASP aplikace do prostředí ASP.NET

Znalostní předpoklady

Aby vám informace obsažené v této kapitole přinesly co možná největší užitek, měli byste splňovat následující vědomostní požadavky:

- Měli byste rozumět programovacím technikám, terminologii a použití jazyka Microsoft Visual Basic.
- Měli byste znát technologii ASP.

Spuštění ASP a ASP.NET na stejném webovém serveru

Jednou z prvních věcí, které si při práci s ASP.NET zcela jistě všimnete, je nová souborová přípona: Webové stránky využívající možností technologie ASP.NET jsou identifikovány extenzí .aspx, kdežto ASP stránky používaly příponu .asp. Rozdíl je také ve vnitřním zpracování dotazů: Když obdrží ASP.NET webová stránka požadavek, IIS tlumočí tento požadavek procesu s názvem aspnet_wp.exe. Na druhé straně, ASP využívá služeb knihovny DLL s názvem asp.dll.

Obě technologie (ASP i ASP.NET) mohou být používány současně na jednom webovém serveru. To znamená, že webové úložiště nebo webová aplikace mohou obsahovat jednak ASP stránky a také novější ASP.NET stránky. Protože k ASP i ASP.NET stránkám lze přistupovat přes jeden a tentýž webový server, není nutné, abyste již vytvořené ASP stránky převáděli do ASP.NET. Na druhé straně ovšem existuje spousta pádných důvodů, proč byste tak mohli udělat. Některé největší výhody jsou shrnuty v následujícím seznamu:

- **Vyšší výkonnost.** Testy realizované společností Microsoft prokázaly, že ASP.NET aplikace mohou vyřídit dva až třikrát tolik požadavků za vteřinu, než jejich ASP protějšky.

- **Vyšší stabilita.** ASP.NET pečlivě monitoruje a řídí všechny nezbytné procesy, takže když se některý z nich začne chovat podivně (způsobí paměťovou díru, nebo přestane reagovat na mrtvém bodě), bude na jeho místě vytvořen nový proces. Tato jednoznačně přínosná vlastnost pomáhá vašim aplikacím reagovat na požadavky uživatelů i v případě možných potíží.

- **Vyšší vývojářská produktivita.** Nové rysy, jako jsou třeba ovládací prvky na straně serveru či zpracovávání událostí v ASP.NET, představují přínosy, které mohou vývojáři využít při rychlejším sestrojování aplikací s menším objemem programového kódu. Oddělování HTML obsahu od zdrojového kódu je nyní snazší než kdykoliv předtím.

Bohužel, převádění ASP stránek na ASP.NET stránky není tak jednoduché, jako pouhé přejmenování souborových přípon z .asp na .aspx, protože, nehledě na jiné faktory, existují velké rozdíly mezi skriptovacím jazykem Microsoft Visual Basic Scripting Edition (VBScript) a jazykem Visual Basic .NET. Dobrou zprávou je, že mnohé z potřebných změn se týkají syntaxe a lze je provést automatizovaně. Programový kód jazyka Visual Basic .NET, jenž využívá komponenty COM (jako třeba ADO nebo vaše vlastní COM komponenty), může být ponechán ve stávající podobě. Naproti tomu, aby kód jazyka C# pracoval spolehlivě s COM komponentami, je potřebná pečlivější analýza a změna kódu, což je úkol, který přesahuje rámcové možnosti této kapitoly.

Tato kapitola je rozdělena do dvou částí: V první uvidíte bližší pohled na tradiční ASP aplikaci, která využívá přístup k datům databáze. Poté se zaměříme na faktory, které byste měli znát, rozhodnete-li se provést migraci ASP aplikace do prostředí moderní technologie ASP.NET.

- **Poznámka**: Tato kapitola se soustřeďuje na portaci ASP aplikace do ASP.NET, přičemž je kladen důraz na to, aby byl tento proces tak snadný, jak to jen jde. Není zde vysvětleno, jak přebudovat ASP aplikaci od startovní čáry pomocí nových rysů ASP.NET.

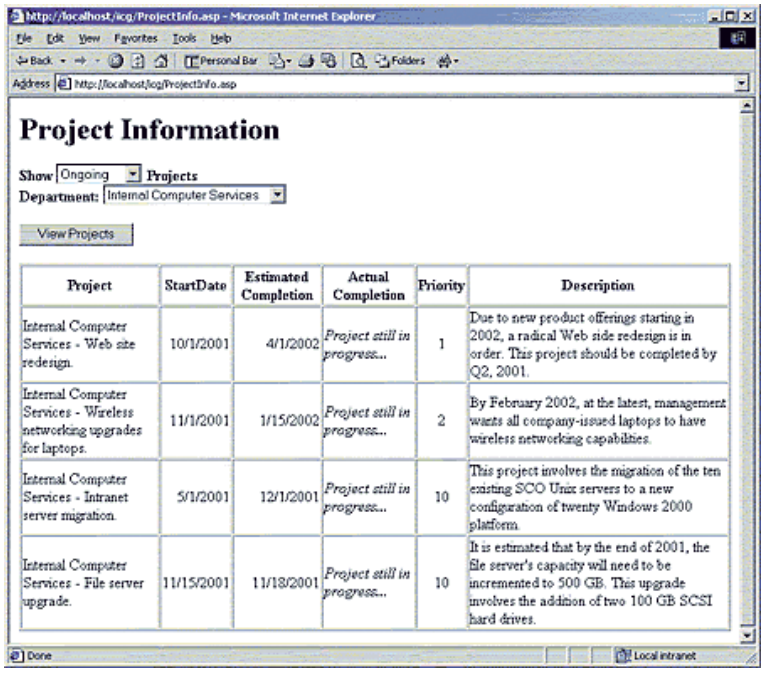
BLIŽŠÍ POHLED NA UKÁZKOVOU ASP APLIKACI

Ukázková ASP aplikace, s níž se seznámíte, a kterou budete posléze převádět do ASP.NET, pracuje jako projektový diář fiktivní společnosti. Aplikace je napsaná v skriptovacím jazyce VBScript. Jde o typickou aplikaci, která zobrazuje informace o nastávajících a uplynulých projektech a umožňuje uživateli provádět jistá konfigurační nastavení.

Tato ASP aplikace je řízená daty, což znamená, že informace o všech projektech jsou uloženy v databázi. Přesněji řečeno, zde jsou použity dvě tabulky dat: Project a Department. Tabulka s názvem Department sdružuje údaje o každém oddělení společnosti, kdežto tabulka s názvem Project obsahuje informace o projektech, jako je třeba jméno projektu, datum zahájení prací na projektu, odhadované datum dokončení projektu, aktuální pokrok v pracích na projektu, priorita projektu, název oddělení, které je zodpovědné za plnění projektu a popis projektu.

Na webové stránce Project Information (Informace o projektu) jsou zobrazeny projekty, které byly zahájeny v uplynulém roce. Uživatel může upravovat pohled na vybraný projekt pomocí dvou seznamů. První seznam umožňuje nastavit zobrazení nadcházejících, dokončených, nebo všech projektů. (Nastávající projekt poznáte tak, že jeho datum dokončení je rovno konstantě NULL. Dokončený projekt disponuje aktuálním datem a datem svého ukončení). Druhý seznam dovoluje uživateli prohlížet projekty podle oddělení, které na nich pracují.

Na obr. 3.1 můžete vidět uživatelské rozhraní ukázkové ASP aplikace. V tomto případě si uživatel zvolil zobrazení všech nadcházejících projektů, za jejichž správu je zodpovědné interní oddělení počítačových služeb.



Obr. 3.1: Přehled nadcházejících projektů oddělení počítačových služeb

Zkoumání programového kódu ukázkové aplikace

Zdrojový kód celé ukázkové aplikace je obsažen v rámci jedné ASP webové stránky, která používá formulář pro zobrazení a realizaci konfiguračních nastavení. Protože naším cílem je demonstrace převodu ASP aplikace do ASP.NET, vysvětlení práce kódu ASP stránky bylo vynecháno. Předpokládá se, že víte, jak ASP stránka pracuje a jak lze získat přístup k datům pomocí technologie ADO.

Následující výpis zdrojového kódu ukazuje vytvoření objektu **ADO Connection** a otevření databáze Microsoft Access.

```
Set objConn = Server.CreateObject("ADODB.Connection")
objConn.ConnectionString = _
   "PROVIDER=Microsoft.Jet.OLEDB.4.0;DATA SOURCE=" & _ Server.MapPath("Projects.mdb") & ";"
objConn.Open
```

Sada záznamů (recordset) je zaplněna položkami **Name** a **DepartmentID** pro každý řádek v tabulce Department. Obsah sady záznamů je zobrazen v seznamu.

```
Set objDeptListRS = Server.CreateObject("ADODB.Recordset")
objDepartmentListingRS.Open "Department", objConn, _
         adOpenForwardOnly, adLockReadOnly, adCmdTable

'Iterate through the Recordset
Response.Write "<b>Department:</b> " & _
      "<select size=""1"" name=""lstDepartmentID"">"
Response.Write "<option value=""-1"">" & _
      "-- Show All Departments --</option>" & vbCrLf

Do While Not objDeptListRS.EOF
   Response.Write "<option value=""" & _
      objDepartmentListingRS("DepartmentID") & """"

   'Do we need to make the item SELECTED?
   If CInt(objDepartmentListingRS("DepartmentID")) = _
                              CInt(iDepartmentID) then
      Response.Write " selected"
   End If

   Response.Write ">" & objDeptListRS("DepartmentName") &_
               "</option>"
   objDeptListRS.MoveNext
Loop
Response.Write "</select>" & vbCrLf & vbCrLf
```

Dále je vytvořen dynamický dotaz jazyka SQL, který je zkonstruován na základě voleb, jejichž výběr provedl uživatel. Všimněte si, že v následujícím fragmentu zdrojového kódu je vytvořená klauzule **WHERE** dotazu SQL, což znamená, že vždy budou vyhledány jenom ty projekty, které byly zahájeny v předcházejícím roce. Vzhledem k uživatelovým preferencím může být použití klauzule **WHERE** dostatečné. Proměnné **iDepartmentID** a **strProjectView** jsou proměnnými, které byly deklarovány dříve a jimž byly přiřazeny hodnoty ze seznamu, podle specifikace uživatele.

```
strSQL = "SELECT D.DepartmentName, ProjectName, " & _
         "StartDate, EstimatedEndDate, ActualEndDate, " & _
         "Priority, ProjectDescription " & _
"FROM Project P " & _
      "INNER JOIN Department D ON " & _
            "D.DepartmentID = P.DepartmentID " & _
"WHERE StartDate >= #" & _
 DateAdd("yyyy", -1, Date()) & "# "

'Now, construct addition WHERE clauses if needed
If CInt(iDepartmentID) <> -1 then
   'Add a clause for the department ID
   strSQL = strSQL & " AND P.DepartmentID = " & _
                              iDepartmentID
End If

'What types of projects do we want to view?
Select Case strProjectView
   Case "ongoing":
      strSQL = strSQL & " AND ActualEndDate IS NULL"
   Case "completed":
      strSQL = strSQL & " AND ActualEndDate IS NOT NULL"
End Select
```

Pokračujeme vytvořením dalšího objektu typu **Recordset** s názvem **objProjectsRS** a jeho naplněním výsledky předcházejícího dynamického dotazování pomocí SQL. Poté je tento objekt použit k iteraci a zobrazení HTML tabulky.

```
'Output the HTML table tag and th tags
Response.Write "<p><table align=""center"" " & _
            "border=""1"" cellspacing=""1"">" & vbCrLf
Response.Write "<tr><th>Project</th><th>StartDate</th>" &_
      "<th>Estimated Completion</th><th>Actual " & _ "Completion</th><th>Priority</th>" & _
      "<th>Description</th></tr>" & vbCrLf
'Loop through the entire Recordset
Do While Not objProjectsRS.EOF
   'Display the Recordset information

   objProjectsRS.MoveNext 'move to the next record
Loop
Response.Write "</table>" & vbCrLf & vbCrLf
```

PŘEVÁDĚNÍ ASP APLIKACE DO PROSTŘEDÍ ASP.NET

Když se rozhodnete pro převod ASP aplikace do prostředí ASP.NET, budete muset zvážit, kolik času jste ochotní strávit implementací nových možností ASP.NET do stávající ASP aplikace. Pro provedení inovace zpravidla stačí, když pozměníte staré souborové přípony z .asp na .aspx a provedete některé syntaktické změny. To by mělo stačit, aby ASP stránka pracovala jako nová ASP.NET stránka. Tento postup, přestože může být realizován velice rychle, nevyužívá žádnou z mnoha nových vlastností vývojové platformy .NET, včetně ASP.NET Web Controls, Microsoft ADO.NET a tříd Microsoft .NET Frameworku. Je zřejmé, že zařazení nových vlastností prodlouží čas vyhrazený pro migraci aplikace, ovšem na druhé straně, po ukončení převodních prací získáte ASP.NET stránky, jejichž kód bude lépe čitelný a které se budou snadněji spravovat.

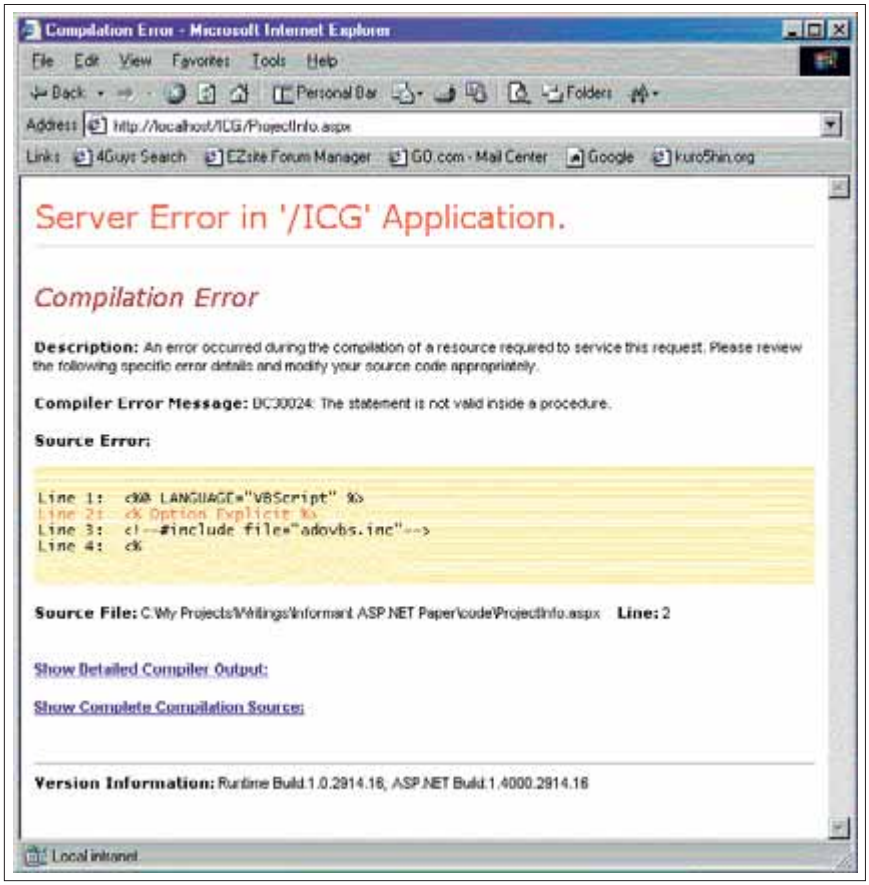
Jako vývojář budete muset pečlivě posoudit, kterou alternativu inovace si vyberete. Když vás tlačí čas, rychlá migrace bude pro vás patrně vhodnější. Jestliže je situace příznivější, možná bude stát za to, abyste si vyhradili dostatek času pro návrh a sestavení bohaté ASP.NET aplikace. Také můžete uplatnit zlatou střední cestu. Spravujete-li rozsáhlý web, jenž využívá služeb většího množství COM komponent pro implementaci obchodní logiky, můžete se rozhodnout uskutečnit pouze převod uživatelského rozhraní vaší webové aplikace a nadále spoléhat na tradiční COM komponenty.

V následující sekci si předvedeme, jak uskutečnit migraci kódu ASP aplikace do ASP.NET s co nejmenším počtem nevyhnutelných zásahů.

Převádění ASP aplikací do ASP.NET

Prvním krokem při převodu ASP aplikací do prostředí ASP.NET je přejmenování souborových přípon stránek z .asp na .aspx. Protože naše ukázková aplikace pozůstává jenom z jediné ASP webové stránky, změna souborové extenze bude velice rychlá. Když jste hotovi, zkuste zobrazit novou .aspx stránku pomocí internetového prohlížeče. Načetla se stránka bez potíží? Pravděpodobně nikoliv. V kódu jazyka VBScript je potřebné opravit určité syntaktické potíže.

Když se budete pokoušet převádět ASP aplikaci, kterou jsme rozebírali dříve, první chybové hlášení, které obdržíte, si bude stěžovat na volbu **Option Explicit** (obr. 3.2).

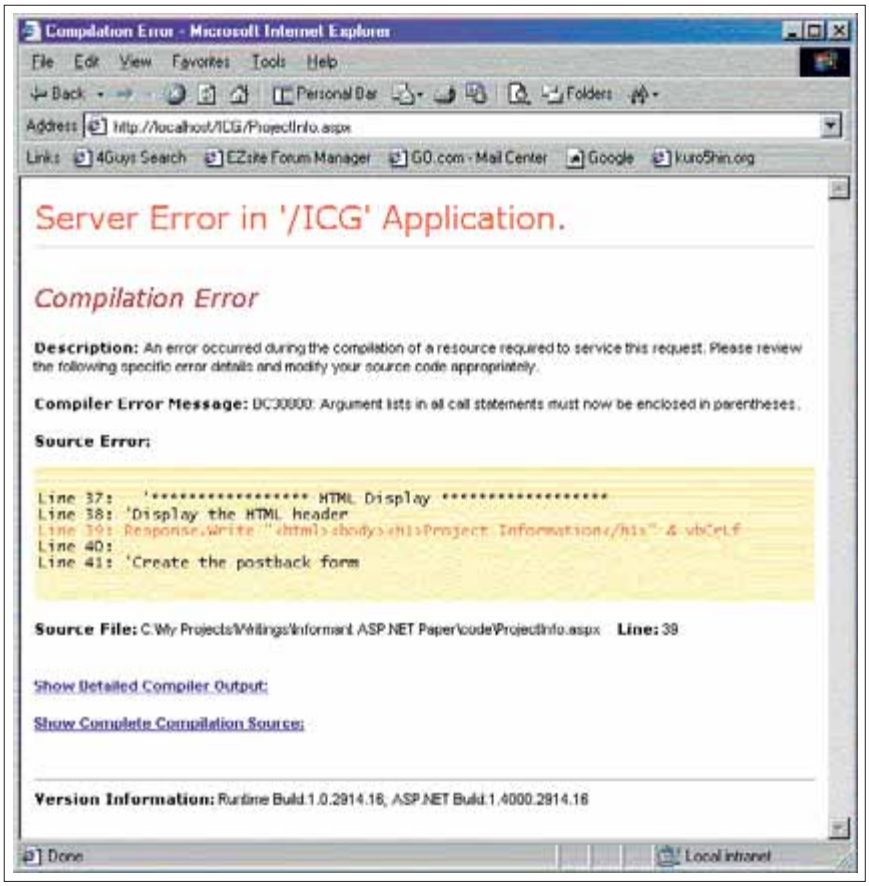


Obr. 3.2: Chybové hlášení

Použití volby **Option Explicit** by mělo být v prostředí ASP.NET webové stránky nahrazeno direktivou **@Page**. Odstraňte proto ze souboru .aspx řádky 1 a 2 a nahraďte je řádkem s direktivou **@Page**:

```
<% @Page Language="VB" Explicit="True" %>
```

Po provedení změny zkuste opětovně načíst webovou stránku do prohlížeče. Výsledkem bude zase chyba - tentokrát si bude prohlížeč stěžovat na chybějící závorky v příkazu **Response.Write** (obr. 3.3).



Obr. 3.3: Problémy s chybějícími závorkami v příkazu ***Response.Write***

Chybějící závorky způsobí chybu, protože Visual Basic .NET vyžaduje, aby byly všechny parametry procedur Sub a funkcí uzavřeny v kulatých závorkách. Projděte celý dokument a umístěte závorky všude tam, kde u příkazů **Response.Write** chybějí. Poté ještě jednou načtěte ASP.NET stránku prostřednictvím vašeho prohlížeče.

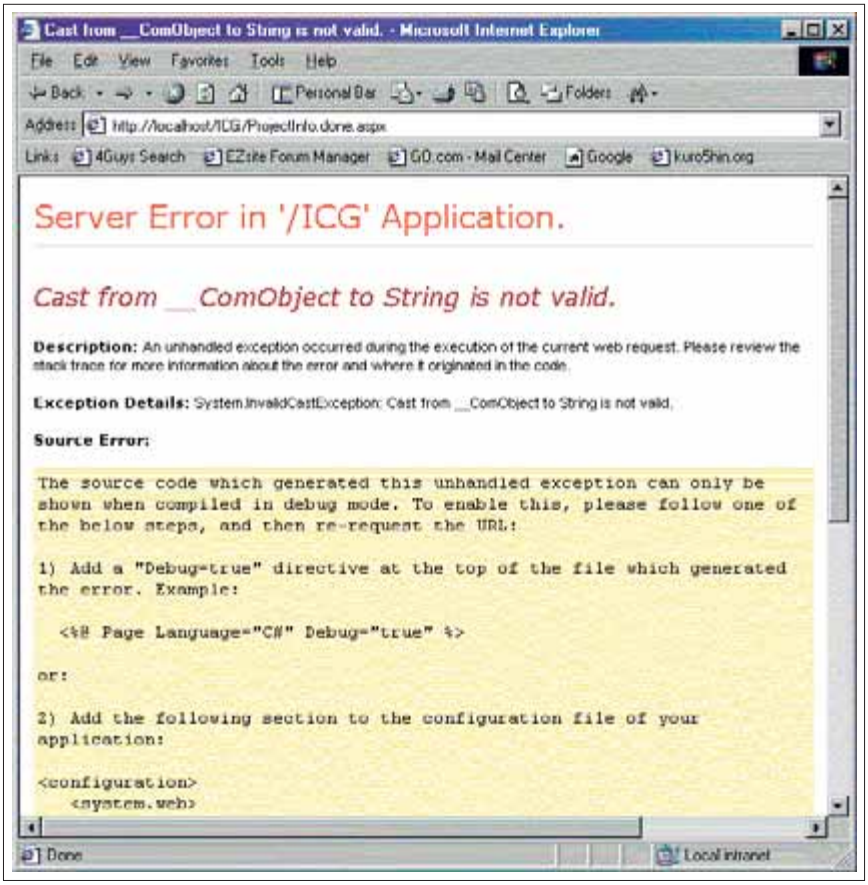
Programovací jazyk Visual Basic .NET, na rozdíl od Visual Basicu 6.0 a VBScriptu, nepodporuje výchozí vlastnosti. Výchozí vlastnosti dovolovaly vývojářům pracovat méně obezřetně - jestliže při aktivaci tradičních COM komponent z Visual Basicu 6.0 nebo VBScriptu programátor špatně určil vlastnost, byla použita výchozí vlastnost. Kupříkladu, výchozí vlastnost objektu **ADO Recordset** je kolekce **Fields** a výchozí vlastností kolekce **Fields** je vlastnost **Value**. Tudíž, když použijete následující kód:

```
Response.Write objRecordset("columnName")
```

v klasickém ASP říkáte v podstatě toto:

```
Response.Write objRecordset.Fields("columnName").Value
```

Při inovaci vaší ASP aplikace na ASP.NET aplikaci můžete obdržet následující tajemnou chybovou zprávu: *Cast from __ComObject to String is not valid* (obr. 3.4).



Obr. 3.4: Chybová zpráva při inovaci na ASP.NET

Zobrazení této chybové zprávy je zapříčiněno právě skutečností, že Visual Basic .NET nepodporuje výchozí vlastnosti. Když tedy použijeme tento kód:

```
Response.Write(objRecordset("columnName"))
```

Visual Basic .NET se snaží přetypovat objekt **Fields** do podoby textového řetězce (typ **String**), což samozřejmě není možné. Řešením je explicitní určení vlastnosti **Value**:

```
Response.Write(objRecordset.Fields("columnName").Value)
```

Když budete pokračovat v procesu převádění vaší ASP.NET aplikace, zcela jistě se setkáte tváří v tvář s dalšími syntaktickými nesrovnalostmi. V tab. 3.1 můžete nalézt přehled syntaktických chyb, s nimiž přijdete do styku při realizování migrace ukázkové aplikace z ASP do ASP.NET.

- **Poznámka**: Ve verzi .NET programovací jazyk Visual Basic dospěl a stal se z něj skutečně plnohodnotný nástroj pro tvorbu programových řešení. Nyní podporuje strukturovanou správu chyb pomocí konstrukce **Try-Catch-Finally**, opravdový objektově orientovaný vývoj a spoustu dalších rozšíření. Bohužel, modernizace jazyka Visual Basic si vyžádala uskutečnění jistých syntaktických modifikací. Z tohoto důvodu není Visual Basic .NET stoprocentně zpětně kompatibilní s jazyky Visual Basic 6.0 a VBScript. Chcete-li získat více informací o jazykových změnách, s nimiž Visual Basic .NET přichází, určitě si přečtěte informační materiál s názvem *Preparing Your Visual Basic 6.0 Applications for the Upgrade to Visual Basic .NET (Připravujeme aplikace Visual Basicu 6.0 pro migraci do prostředí jazyka Visual Basic .NET).*

  Tento materiál můžete najít na adrese *http://msdn.microsoft.com/asp.net/using/migrating/default.aspx?pull=/library/en-us/dnvb600/html/vb6tovbdotnet.asp.*

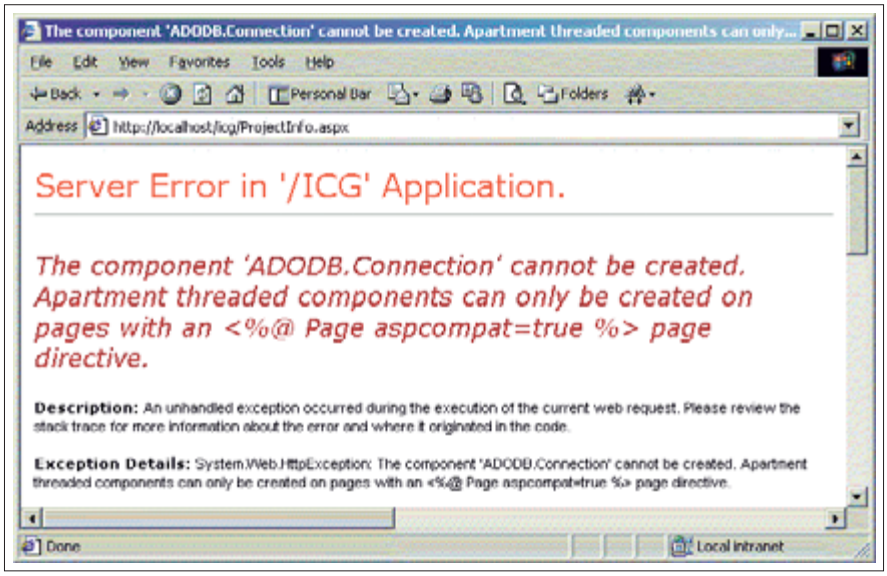
Tab. 3.1: Přehled syntaktických chyb, se kterými se můžete střetnout při migraci ukázkové ASP webové aplikace

CHYBA	DŮVOD	ŘEŠENÍ
Neplatné umístění příkazu Option Explicit v těle procedury.	Volba **Option Explicit** musí být definována v direktivě **Page** pomocí atributu **Explicit**.	Přidejte řádek **<%@Page Explicit="True"%>** na začátek kódu ASP.NET stránky.
Je možné použít jenom jednu direktivu **Page**.	Když přidáme direktivu **Page**, direktiva **@LANGUAGE="VBSCRIPT"** v kódu stránky ASP se stane nadbytečná.	Odstraňte řádek, jenž obsahuje **<%@LANGUAGE="True"%>** a přidejte atribut **Language="VB"** do direktivy **Page**.
Seznamy argumentů při volání procedur Sub nebo funkcí musí být umístěny do kulatých závorek.	Visual Basic .NET vyžaduje, aby byly při volání procedur všechny argumenty uzavřeny v kulatých závorkách. Značné množství příkazů **Response.Write** se vyskytuje v nevhodné podobě.	Přidejte chybějící závorky: Kupříkladu příkaz **Response.Write str** změňte na **Response.Write(str)**.
Příkazy **Let** a **Set** již nejsou nadále podporovány jako přiřazovací příkazy	Jelikož Visual Basic .NET již nepodporuje výchozí vlastnosti, klíčová slova **Let** a **Set** byla z jazyka odstraněna.	Vymažte všechna klíčová slova **Let** a **Set**. V ukázkové aplikaci je použito klíčové slovo **Set** pro přiřazení odkazů na objekty **Connection** a **Recordset** do příslušných objektových proměnných.
Klíčové slovo **Date** představuje datový typ a ne platný výraz. Je očekávána proměnná, konstanta nebo procedura.	V ukázkové ASP aplikaci je volána funkce **Date()** pro získání aktuálního data. Tato programová konstrukce již není nadále podporována.	Nahraďte volání funkce **Date()** příkazem **DateTime.Now**.
Jméno **IsNull** není deklarováno.	Visual Basic .NET nepodporuje funkci **IsNull**. (V ukázkové aplikaci není projekt dokončen, jestliže aktuální datum dokončení je rovno konstantě NULL. V ASP verzi se funkce **IsNull** používá pro zjištění, zdali je hodnotou proměnné konstanta NULL.	Abyste vyřešili tento problém, nahraďte funkci **IsNull** funkcí **IsDBNull**.
Cast from _ComObject to String is not valid (Přetypování z **_ComObject** do typu **String** není platné).	Visual Basic .NET nepodporuje výchozí vlastnosti, takže když používáte tradiční COM komponenty, vždy se ujistěte, že jste explicitně specifikovali název vlastnosti, kterou chcete použít.	Změňte instance příkazu **objRS("colName")** na **objRS.Fields("colName").Value**.

POTÍŽE S COM KOMPONENTY

Některé potíže se mohou vynořit v okamžiku, kdy začnete používat COM komponenty uvnitř ASP.NET webové stránky. Jedná se zejména o COM komponenty, jejichž vlákna pracují v komnatách (Apartment-Threaded, AT), nebo COM objekty, které přistupují k interním objektům platformy ASP (jako je **Request**, **Response**, **Server**, **Application** a **Session**) pomocí objektu **ObjectContext**.

Například, objekty ADO jsou v registrech operačního systému deklarovány jako AT. Když se budete pokoušet používat AT komponenty prostřednictvím ASP.NET webové stránky, obdržíte chybové hlášení se sdělením, že AT komponenta, kterou jste chtěli vytvořit, nemohla být sestrojena. Obr. 3.5 zobrazuje ASP.NET verzi projektového diáře, tedy naší ukázkové aplikace. Všimněte si, že se zde vyskytuje chyba, protože objekty ADO jsou implicitně označovány jako AT objekty.



Obr. 3.5: Při pokusu o získání přístupu k AT COM komponentám pomocí ASP.NET webové stránky, se objeví hlášení o chybě

Naštěstí, ASP.NET nabízí ASP kompatibilní mód, jak ostatně praví také chybové hlášení. Budete-li chtít aktivovat tento kompatibilní mód, přidejte atribut **aspcompat=true** do direktivy **Page**. Přidáním uvedeného atributu dosáhnete dvojitého účinku:

- ASP.NET použije při přístupu ke COM komponentě vlákno jednovláknové komnaty (Single-Threaded Apartment, STA). Výchozím nastavením je používání modelu vícevláknové komnaty (Multi-Threaded Apartment, MTA).

- ASP.NET poskytne přístup k interním objektům ASP pomocí modelu zpětné kompatibility.

Aktivace ASP kompatibilního módu dovoluje také ASP.NET webovým stránkám používat AT COM komponenty, nebo COM komponenty, které jsou schopny přistupovat k interním objektům ASP.

■ **Poznámka**: Hlubší diskuse na téma vláknových modelů COM komponent přesahuje možnosti této kapitoly. Budete-li chtít získat podrobnější informace o uvedeném tématu, neměli byste minout výtečné čtení s názvem Understanding and Using COM Threading Models (Porozumění a používání vláknových modelů COM), které můžete najít na adrese *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomg/html/comthreading.asp*.

Závěr

Právě jste se obeznámili s tím, jak realizovat migraci ASP webové aplikace do prostředí ASP.NET. Jak jste mohli vidět, inovace na novou technologii nezahrnuje mnoho práce navíc, než jen změnu souborových přípon webových stránek. Převod naší ukázkové aplikace, která obsahovala asi 200 řádků kódu, z ASP do ASP.NET trval méně než pět minut. Když se rozhodnete pro inovaci na ASP.NET, mějte na paměti následující:

- Věnujte čas studiu syntaktických změn, s nimiž přichází nová verze jazyka Visual Basic s přídomkem .NET. Budete-li obeznámeni se změnami v tomto programovacím jazyce, budete schopni realizovat proces migrace aplikací o poznání rychleji a bez větších problémů.

- Pokud ASP stránka využívá služeb COM komponent, jejichž vlákna jsou spravována v komnatách, nebo jestli COM komponenty přistupují k interním objektům ASP, nezapomeňte přidat atribut **aspcompat="true"** do direktivy **Page**.

O autorovi

Scott Mitchell je zakladatel a editor webového sídla *www.4GuysFromRolla.com*, jednoho z největších webů, které se věnují problematice vývoje ASP stránek. Scott nadšeně používá a píše o aktivních serverových stránkách již od ledna roku 1998. Napsal stovky článků, které se zabývají tématikou ASP a ASP.NET. Rovněž uvedl na trh několik knih o ASP a ASP.NET, které byly určeny široké škále zájemců, začátečníky počínaje a profesionály konče.

O Informant Communications Group

Informant Communications Group, Inc. (*www.informant.com*) je různorodá mediální společnost, která se soustřeďuje na bouřlivě se rozvíjející sektor informačních technologií. ICG byla založena v roce 1990 a od té doby se specializuje na vydávání publikací o vývoji softwaru, pořádání konferencí, přípravu katalogů a webů. Kanceláře společnosti se nacházejí v USA a Velké Británii. ICG si vydobyla postavení respektovaného mediálního a marketingového integrátora, jenž uspokojuje poptávku IT specialistů po kvalitních technických informacích.

4 Směrem k ASP.NET: Klíčové faktory

Autor: Jim Kieley, Microsoft Corporation

SHRNUTÍ

Čtvrtá kapitola prozkoumává základní faktory, které byste měli vzít v potaz při inovování stávajících ASP aplikací na platformu ASP.NET nejrychlejší možnou cestou.

ÚVOD

Přestože návrháři společnosti Microsoft odvedli při vývoji technologické platformy ASP.NET z hlediska zpětné kompatibility s ASP opravdu skvělou práci, existuje pár klíčových faktorů, kterým byste měli porozumět ještě předtím, než se vrhnete na portování webových aplikací. Solidní znalost technologií, které se změnily, nebo byly uvedeny s .NET platformou a ASP.NET, vám umožní lépe pochopit všechny nezbytné souvislosti.

Tento materiál s sebou přináší množství informací o kritických oblastech, které byly v nové verzi platformy ASP změněny. Cílem je poskytnout vám jasné a srozumitelné poznatky, z nichž můžete čerpat během procesu pozvedání úrovně existujících ASP aplikací. Dozvíte se také, jaké nové schopnosti uvádí ASP.NET a jak lze tato rozšíření začlenit do již fungujících aplikací. Buďte si prosím vědomi, že tato kapitola nepředstavuje vyčerpávající pohled na všechna vylepšení a nové prvky, které byly do platformy ASP.NET implementovány. Středem zájmu se místo toho stávají jisté problémové okruhy, o nichž byste měli být informovaní, a znalost kterých vám může v notné míře pomoci při uskutečnění hladkého přechodu mezi ASP a ASP.NET.

Jelikož vycházím z premisy, že drtivá většina ASP aplikací je napsána pomocí skriptovacího jazyka Microsoft Visual Basic Scripting Edition (VBScript), předpokládám, že vaší volbou se při migraci směrem k ASP.NET stane programovací jazyk Visual Basic .NET. Ačkoliv tohle není nevyhnutelný požadavek, může změna hlavního programovacího jazyka způsobit vynaložení dodatečného úsilí pro obeznámení se s modifikovanou architekturou nového vývojového nástroje, jímž Visual Basic .NET beze sporu je.

Koexistence

Ještě předtím, než si začneme povídat o specifických kompatibilních a migračních okruzích, je důležité, abyste porozuměli tomu, jak mohou obě vývojové platformy (ASP a ASP.NET) společně existovat. ASP i ASP.NET webové aplikace mohou běžet vedle sebe na jednom webovém serveru, aniž by se jakýmkoliv nepříznivým způsobem vzájemně ovlivňovaly. Tato příznivá skutečnost je způsobena tím, že obě technologie používají separátní souborové přípony (.asp oproti .aspx) a konfigurační modely (metabázy/registry oproti konfiguračním souborům, jejichž základy spočívají na jazyku XML). Oba systémy disponují také zcela samostatnými exekučními prostředími.

Je dokonce možné, aby jedna část vaší aplikace běžela pod křídly ASP, zatímco o běh jiné části se bude starat ASP.NET. Tato vlastnost je velice užitečná, a to zejména v situaci, kdy potřebujete provést rychlou inovaci jednotlivých modulů vaší aplikace do ASP.NET. Možná, že si teď řeknete, že zcela nejlepší by bylo kompletní přepracování webové aplikace jednou a provždy. Ačkoliv tenhle scénář může být vhodný pro jistou skupinu webových aplikací, domnívám se, že existuje opravdu velký počet webů, u kterých uvedený přístup nebude možné realizovat, a to z mnoha různých důvodů: komplikovaná hierarchická struktura webového sídla, kapacitní náročnost či rychlá obměna obsahu a náplně webu. Koneckonců, samotní uživatelé budou již zanedlouho od vašeho webu vyžadovat mnoho moderních rysů, což bude patrně vést k tomu, abyste dali šanci novým technologiím. Navíc, když se jednou rozhodnete pro přechod na ASP.NET, budete určitě chtít využít tuto příležitost pro provedení tolika návrhových vylepšení, kolik budete schopni zvládnout. A jelikož proces inovace je vskutku dlouhotrvající investicí, jistě oceníte schopnost paralelní exekuce obou technologií vedle sebe.

POTÍŽE S KOMPATIBILITOU

Povznesení vaší aplikace na úroveň ASP.NET nemusí být snadné, ovšem na druhé straně by se nemělo jednat ani o nijak zvlášť komplikovanou operaci. ASP.NET si velmi dobře rozumí se svou předchůdkyní, což je určitě impozantní skutečnost, když si uvědomíme, že ASP.NET představuje kompletně novou softwarovou platformu. Úloha návrhářů ASP.NET nebyla proto vůbec lehká: Přestože bylo potřebné zachovat 100 %-ní zpětnou kompatibilitu s ASP, finálním cílem bylo vyvinout technologii, jejíž dovednosti by daleko přesahovaly meze ASP. Vykonané změny slouží k dosahování lepších výsledků, což je potěšitelné o to víc, že nároky, které jsou kladené na vývojáře, nejsou přehnaně veliké. Pokud bychom chtěli být přesní, mohli bychom dostupné změny rozdělit do následujících kategorií:

- Změny uvnitř základního aplikačního programového rozhraní (API)
- Strukturální změny
- Změny v syntaktické skladbě programovacího jazyka Visual Basic
- Změny související s implementací technologie COM
- Změny v konfiguraci aplikací
- Změny v řízení stavu aplikací
- Změny související se zabezpečením aplikací
- Změny při přístupu k datům

Na následujících řádcích se na všechny uvedené změny podíváme blíže.

ZMĚNY UVNITŘ ZÁKLADNÍHO APLIKAČNÍHO PROGRAMOVÉHO ROZHRANÍ (API)

Základní API technologie ASP je tvořeno souhrnem interních objektů (**Request**, **Response**, **Server** a další) a jejich metodami. Pokud odhlédneme od několika málo změn, lze součásti API používat i v ASP.NET, aniž by byla jakkoliv narušena správná funkčnost aplikace. Všechny změny, které se týkají objektu **Request** jsou názorně vysvětleny v tab. 4.1.

Tab. 4.1: Změny provedené v základním aplikačním programovém rozhraní (API)

METODA	ZMĚNA
Request(argument)	V prostředí ASP je tato vlastnost kolekcí. V ASP.NET představuje **Request** vlastnost **NameValueCollection**, která vrací textový řetězec v závislosti od předaného argumentu.
Request.QueryString(argument)	V prostředí ASP je tato vlastnost kolekcí. V ASP.NET představuje **QueryString** vlastnost **NameValueCollection**, která vrací textový řetězec v závislosti od předaného argumentu.
Request.Form(argument)	V prostředí ASP je tato vlastnost kolekcí. V ASP.NET představuje **Form** vlastnost **NameValueCollection**, která vrací textový řetězec v závislosti od předaného argumentu.

Jak si můžete všimnout, provedené změny jsou v podstatě stejné pro všechny vestavěné vlastnosti.

Jestliže položka, k níž přistupujete, obsahuje právě jednu hodnotu pro specifikovaný klíč, není nutná modifikace vašeho programového kódu. Na druhé straně, jestli existuje několik hodnot pro daný klíč, budete muset pro obdržení kolekce hodnot použít jinou metodu. Pamatujte také na to, že ve Visual Basicu .NET jsou kolekce číslovány od nuly, kdežto základem pro práci s kolekcemi v skriptovacím jazyce VBScript je jednička.

Kupříkladu, pokud bychom chtěli v ASP zaslat požadavek pro získání textových řetězců z adresy *http://localhost/myweb/valuetest.asp?values=10&values=20*, použili bychom následující fragment kódu:

```
<%
 'This will output "10"
 Response.Write Request.QueryString("values")(1)

 'This will output "20"
 Response.Write Request.QueryString("values")(2)
%>
```

V prostředí ASP.NET existuje vlastnost **QueryString**, pomocí níž lze získat objekt **NameValueCollection**, z něhož je možné získat přístup ke kolekci **Values** a posléze obdržet tu položku, o kterou máte zájem. Ještě jednou si připomeňme, že první položku kolekce získáme pomocí nulového indexu a ne indexu 1. Prostudujte si další výpis kódu:

```
<%
 'This will output "10"
 Response.Write (Request.QueryString.GetValues("values")(0))

 'This will output "20"
 Response.Write (Request.QueryString.GetValues("values")(1))
%>
```

Následující fragment zdrojového kódu se bude chovat stejně v ASP i v ASP.NET:

```
<%
 'This will output "10", "20"
 Response.Write (Request.QueryString("values"))
%>
```

STRUKTURÁLNÍ ZMĚNY

Pod pojmem „strukturální změny“ se rozumí změny, které ovlivňují návrh a stavbu kódu při vývoji aktivních serverových stránek. Abyste mohli garantovat spolehlivost práce vašeho kódu v ASP.NET, měli byste věnovat několik okamžiků pro obeznámení se se změnami v této oblasti.

Segmenty kódu: Deklarování proměnných a funkcí

V ASP můžete deklarovat veřejné proměnné a procedury Sub mezi oddělovači programového kódu:

```
<
%
 Dim X
 Dim str
 Sub MySub()
   Response.Write "This is a string."
 End Sub
%>
```

V ASP.NET není uvedené rozložení kódu přípustné. Nyní musíte deklarovat všechny proměnné a funkce uvnitř bloku, jenž je uveden mezi značkami **<script>** a **</script>**.

```
<script language = "vb" runat = "server">
   Dim str As String
   Dim x, y As Integer

   Function Add(I As Integer, J As Integer) As Integer
      Return (I + J)
   End Function
</script>
```

Míchání programovacích jazyků

V prostředí ASP máte při výběru programovacího jazyka v zásadě pouze dvě možnosti: VBScript nebo Microsoft JScript®. Dobrou vlastností je, že v kódu jedné stránky se mohou nacházet skripty napsané v obou výše uvedených skriptovacích jazycích.

ASP.NET je na tom, co se týče použití programovacích jazyků, o poznání lépe. Ve skutečnosti totiž můžete pracovat s kterýmkoliv jazykem, jenž splňuje pravidla daná společným běhovým prostředím platformy .NET Framework. C#, Visual Basic .NET a JScript, to vše jsou programovací jazyky, které tvoří portfolio vývojových nástrojů společnosti Microsoft. Všimněte si, že mluvím o Visual Basicu .NET a ne o VBScriptu. Je to proto, že na platformě .NET Framework neexistuje žádný jazyk s názvem VBScript. VBScript byl plně nahrazen právě jazykem Visual Basic .NET. Je sice pravdou, že si můžete vybrat kterýkoliv z dostupných .NET programovacích jazyků, ovšem použití samotných jazyků již nemůžete vzájemně kombinovat na jedné webové stránce tak, jak jste to mohli dělat v ASP. Je samozřejmě možné vytvořit dvě webové stránky, z nichž každá bude obsahovat kód dvou různých jazyků, jakými jsou třeba C# a Visual Basic .NET. Souběžné použití segmentů kódů různých jazyků na jedné stránce již však není dovoleno.

Nové Page direktivy

V ASP musíte umístit všechny direktivy na první řádek s kódem stránky, například takto:

```
<%LANGUAGE="VBSCRIPT" CODEPAGE="932"%>
```

V prostředí ASP.NET je nutné použít direktivu **Language** společně s direktivou **Page** následovně:

```
<%@Page Language="VB" CodePage="932"%>
<%@QutputCache Duration="60" VaryByParam="none" %>
```

Direktivy se mohou nyní rozprostírat na tolika řádcích, kolik jich budete potřebovat. Direktivy mohou být umístěny kdekoliv uvnitř vašeho .aspx souboru, ovšem pokud chcete následovat zažité konvence, měli byste je vkládat na začátek kódu souboru.

Platforma ASP.NET přinesla několik nových direktiv. Jelikož vám tyto nově zařazené direktivy mohou přinést usnadnění či zefektivnění práce, doporučuji vám, abyste se s jejich významem a syntaxí obeznámili v dokumentaci věnované ASP.NET.

Překládané procedury Sub nejsou nadále podporovány

Máte-li důkladnější znalosti vývoje ASP webových stránek, patrně víte, že zde můžete používat tzv. překládané (render) procedury Sub. Překládané procedury Sub jsou subrutiny, v jejichž tělech se nacházejí fragmenty jazyka HTML. Příklad překládané procedury Sub předvádí následující ukázka programového kódu:

```
<%Sub RenderMe()
%>
<H3> This is HTML text being rendered. </H3>
<%End Sub
RenderMe
%>
```

Prostřednictvím překládaných procedur a funkcí můžete vytvořit mnoho líbivých efektů, ovšem tento styl programování není v ASP.NET podporován. A možná je to i lepší. Jsem si totiž jist, že jste již viděli funkce, jejichž kód se brzy stal nečitelný a jejichž správa byla opravdovou hrůzou. Nejschůdnější cestou, jak uvést dosavadní překládané procedury a funkce do chodu i v ASP.NET, je nahradit HTML kód voláními metody **Response.Write**:

```
<script language="vb" runat="server">
Sub RenderMe()
Response.Write("<H3> This is HTML text being rendered. </H3>")
End Sub
</script>

<%
 Call RenderMe()
%>
```

Nejsnazší cesta ale nemusí vždy znamenat také cestu nejlepší. V závislosti od rozsahu a spletitosti programového kódu může být někdy vhodnější poohlížet se po webových ovládacích prvcích, pomocí nichž můžete programově upravovat HTML atributy a pečlivě oddělit zdrojový kód od dalšího obsahu webu. Tuto proceduru byste měli podstoupit i v případě, když chcete, aby byl váš kód lépe čitelný.

ZMĚNY V SYNTAKTICKÉ SKLADBĚ PROGRAMOVACÍHO JAZYKA VISUAL BASIC

Jak jsem se zmiňoval již dříve, skriptovací jazyk VBScript musel ustoupit svému mohutnějšímu a lépe vybavenému kolegovi, jehož jméno je Visual Basic .NET. V této sekci poukáži na některé problémové oblasti, které souvisejí se změnou syntaktické struktury programovacího jazyka Visual Basic .NET. Předem vám dávám na vědomí, že nabízené informace nepředstavují ani zdaleka vyčerpávající pojednávání o všech změnách, modifikacích a vylepšeních, které byly do nové verze Visual Basicu implementovány. Místo toho jsem se zaměřil na témata, s nimiž jako vývojáři v ASP/VBScript přijdete zcela jistě do styku, jakmile učiníte první krůčky směrem k ASP.NET a Visual Basicu .NET. Budete-li přesto cítit, že potřebujete znát také další informace, které se s jazykem Visual Basic .NET pojí, doporučuji vám nalistovat příslušné stránky v on-line dokumentaci.

Rozlučte se s datovým typem Variant

Známe ho, milujeme ho, dokonce ho milujeme až k nenávidění. Ano, mluvím o datovém typu **Variant**. Datový typ **Variant** již není součástí platformy .NET a tudíž jej nenajdete ani ve Visual Basicu .NET. To tedy znamená, že pokud jste v ASP používali proměnné typu **Variant**, měli byste tento datový typ nahradit novým typem **Object**. Datové typy proměnných, které jste deklarovali ve vaši ASP aplikaci, by měly být nahrazeny příslušnými .NET primitivními datovými typy. Jestliže je vaše proměnná z hlediska Visual Basicu objektovou proměnnou, jednoduše ji v prostředí ASP.NET deklarujte pomocí datového typu **Object**.

Visual Basic a datové typy

Jedním variantním typem, jenž si zaslouží pečlivější pozornost je typ **VT_DATE**. Tento datový typ byl ve Visual Basicu převtělen do typu s názvem **Date**, jenž byl ukládán v podobě desetinného čísla s dvojitou přesností (typ **Double**). Proměnné tohoto typu spotřebovávaly čtyři bajty. V jazyce Visual Basic .NET jsou data uchovávána prostřednictvím datového typu společného běhového prostředí s názvem **DateTime**, jehož proměnné mají 8bajtovou celočíselnou reprezentaci.

Vzhledem k tomu, že jediným datovým typem je v ASP typ **Variant**, vaše proměnné pracující s daty (typ **Date**) by měly být i nadále provozuschopné v závislosti od způsobů, podle nichž jsou používány. Je však možné, že při realizaci jistých programových operací se mohou vyskytnout neočekávané potíže, které jsou způsobené změnou primitivních datových typů. Věnujte proto pozornost těm segmentům kódu, v nichž dochází k předávaní dat COM objektům, nebo které provádějí konverzní operace hodnot typu **Date** pomocí konverzní funkce **CLng**.

Volba Option Explicit je nyní implicitně aktivní

Také v ASP bylo možné používat volbu **Option Explicit**, ovšem tato volba nebyla implicitně aktivní. Tato situace se v jazyce Visual Basic .NET změnila. Volba **Option Explicit** je nyní standardně aktivní, což znamená, že všechny proměnné musejí být správně deklarovány předtím, než budou používány. Dokonce je dobré ještě více přitvrdit a nahradit volbu **Option Explicit** volbou **Option Strict**. Poté bude Visual Basic .NET vyžadovat nejenom to, abyste všechny proměnné řádně deklarovali, ale abyste také specifikovali datový typ každé deklarované proměnné. Možná, že na první pohled to vypadá jako extra práce navíc, ovšem tato technika byla s pozitivním výsledkem prověřena mnoha programátory. Pokud se rozhodnete nepoužívat volbu **Option Strict**, váš kód bude méně výkonný, protože když výslovně neurčíte datové typy deklarovaných proměnných, bude všem přiřazen typ **Object**. Je sice možné, že většina implicitních konverzí bude i nadále fungovat, nicméně kontrola explicitní deklarace datových typů proměnných je přesto nejlepším řešením.

Klíčová slova LET a SET nejsou nadále podporována

Objektové reference mohou být nyní kopírovány přímo mezi referenčními proměnnými, třeba takto: **MyObj1 = MyObj2**. Známá klíčová slova **Let** a **Set** již nebudete v těchto situacích potřebovat. Disponuje-li kód příkazy **Let** a **Set**, budete je muset odstranit.

Použití kulatých závorek při volání metod

Při volání metod objektů v ASP jste mohli specifikovat argumenty, aniž byste je umístili do závorek (viz příklad níže):

```
Sub WriteData()
Response.Write "This is data"
End Sub
WriteData
```

V ASP.NET musíte při volání funkcí nebo procedur Sub vždy používat kulaté závorky, a to i v případě, kdy cílová metoda nepracuje s žádnými parametry. Když budete váš kód psát podle následujícího modelu, procedury budou pracovat spolehlivě v ASP i ASP.NET.

```
Sub WriteData()
Response.Write("This is data")
End Sub
Call WriteData()
```

Předávání argumentů hodnotou (ByVal) je nyní implicitní

Ve Visual Basicu byly implicitně všechny argumenty předávány odkazem pomocí klíčového slova **ByRef**. Jazyk Visual Basic .NET standardně předává argumenty hodnotou (klíčové slovo **ByVal**). Budete-li potřebovat předat argument odkazem, budete muset před název formálního parametru funkce nebo procedury Sub explicitně umístit klíčové slovo **ByRef**. Přesně tak, jak demonstruje následující programová ukázka:

```
Sub MyByRefSub (ByRef Value)
Value = 53
End Sub
```

Problematika předávání argumentů vyžaduje více pozornosti. Když se rozhodnete inovovat směrem k ASP.NET, měli byste dva až třikrát kontrolovat vztah mezi argumenty a formálními parametry, abyste se ujistili, že vše probíhá tak, jak jste to původně plánovali. Domnívám se, že jisté partie vašeho kódu budou vyžadovat pečlivější analýzu.

Výchozí vlastnosti patří minulosti

Kdybyste ve Visual Basicu .NET hledali výchozí vlastnosti, vaše hledání by bylo marné. Spoléhal-li se kód vaší ASP aplikace na použití výchozích vlastností objektů, budete muset provést změny, pomocí nichž explicitně determinujete cílovou vlastnost, kterou si přejete použít. Následující výpis zdrojového kódu vám poví víc.

```
'ASP Syntax (Implicit retrieval of Column Value property)
Set Conn = Server.CreateObject("ADODB.Connection")
Conn.Open("TestDB")
Set RS = Conn.Execute("Select * from Products")
Response.Write RS("Name")

'ASP.NET Syntax (Explicit retrieval of Column Value property)
Conn = Server.CreateObject("ADODB.Connection")
Conn.Open("TestDB")
RS = Conn.Execute("Select * from Products")
Response.Write (RS("Name").Value)
```

Změny v datových typech

Programovací jazyk Visual Basic .NET vnáší nové světlo také do sféry datových typů a jejich použití. Tak například, proměnné typu **Integer** jsou nyní dlouhé 32 bitů. Novinkou je také rozsah proměnných datového typu **Long**, jenž činí 64 bitů.

Potenciální potíže s datovými typy mohou vzniknout při aktivaci metod COM objektů z ASP.NET, nebo při volání funkcí Microsoft Win32® API z uživatelských komponentů připravených ve Visual Basicu.

Strukturovaná správa chyb

Jste-li znalí jazyka Visual Basic, jistě víte, že správa chyb byla realizována prostřednictvím příkazů **On Error Resume Next** a **On Error GoTo**. Přestože tyto příkazy jsou použitelné i v jazyce Visual Basic .NET, nepředstavují již nyní nejlepší způsob ošetřování vzniklých chybových stavů aplikací. Visual Basic .NET disponuje rozvinutou strukturovanou správou chyb, která je prováděna pomocí klíčových slov **Try**, **Catch** a **Finally**. Je-li to možné, měli byste vždy používat novou strukturovanou správu chyb, jelikož nabízí robustnější a výkonnější mechanizmus pro zachytávání chybových výjimek.

ZMĚNY SOUVISEJÍCÍ S IMPLEMENTACÍ TECHNOLOGIE COM

Uvedení platformy .NET Framework a ASP.NET neovlivnilo žádným způsobem technologii COM. Bohužel to neznamená, že byste měli být klidní, co se týče používání objektů COM uvnitř ASP.NET webových aplikací. Přinejmenším o dvou věcech byste měli být informovaní.

Změny v modelech správy vláken

ASP.NET využívá vícevláknové komnaty (Multiple Threaded Apartment, MTA), což znamená, že COM komponenty, které byly vytvořeny pro běh v jednovláknových komnatách (Single Threaded Apartment, STA) nebudou v ASP.NET pracovat správně a budete jim tedy muset věnovat jistou pozornost. Model správy jednovláknových komnat implicitně používají všechny COM komponenty, které byly připraveny pomocí jazyka Visual Basic 6.0 a dřívějších verzí.

Atribut ASPCOMPAT

Určitě rádi uslyšíte, že můžete vaše STA COM komponenty používat nadále i v ASP.NET webových aplikacích, aniž byste museli nějak významněji zasahovat do struktury programového kódu. Vše, co musíte udělat, je přidat atribut **aspcompat=true** do značky **<%@Page>** vaší ASP.NET webové stránky. Upravená verze této značky by tedy mohla vypadat asi následovně: <%@Page aspcompat=true Language=VB%>. Aplikujete-li tento atribut, přinutíte váš kód běžet v módu jednovláknové komnaty (STA) a tím zabezpečíte, že vaše COM komponenty budou řádně pracovat i v novém prostředí. Na druhé straně, pokud byste se pokusili používat jakoukoliv STA komponentu bez uvedení vzpomínaného atributu, obdrželi byste chybovou výjimku za běhu aplikace. Vhodná konfigurace atributu **aspcompat** má vliv také na správné fungování komponent COM+ 1.0, které přistupují k neřízeným vestavěných objektům ASP. Tyto interní objekty jsou dostupné před objekt **ObjectContext**.

Provedete-li aktivaci tohoto atributu, výkon vaší aplikace nepatrně poklesne. Proto je vhodné zařazovat atribut **aspcompat** jenom v okamžicích, kdy jeho služby doopravdy potřebujete.

Časné vázání versus pozdní vázání

V ASP probíhají všechna volání COM objektů prostřednictvím rozhraní **IDispatch**. Tato technika je známá jako pozdní vázání (late-binding), protože volání aktuálních objektů jsou za běhu aplikace řízena nepřímo přes rozhraní **IDispatch**. Vyhovuje-li vám takovýto způsob používání objektů, můžete jej nadále uplatňovat i v ASP.NET aplikacích.

```
Dim Obj As Object
Obj = Server.CreateObject("ProgID")
Obj.MyMethodCall
```

Ačkoliv pozdní vázání pracuje dobře, existuje lepší řešení, pomocí něhož můžete přistupovat k objektům vašich komponentů. Ano, tušíte správně, toto řešení má název časné vázání (early-binding) a je dostupné přímo z prostředí ASP.NET. Při časném vázání určíte již při psaní programového kódu typ objektu, který se má vytvořit.

```
Dim Obj As New MyObject
MyObject.MyMethodCall()
```

Technika časného vázání vám dovoluje realizovat typově bezpečnou interakci s objekty vašich komponent. Abyste mohli provádět časné vázání objektů COM komponent, musíte do vašeho projektu přidat odkaz na cílovou komponentu, jejíž objekty hodláte vytvářet. Zařazení odkazu na COM komponentu je podobné způsobu, jakým jste tuto činnost uskutečňovali ve vývojovém prostředí jazyka Visual Basic 6.0. Jestliže pracujete ve Visual Studiu .NET, bude ihned po přidání odkazu na COM komponentu vytvořen řízený proxy objekt, který bude zabezpečovat komunikaci mezi řízeným kódem a komponentou. Vy však budete mít dojem, že pracujete přímo s COM komponentou. Po pravdě řečeno, provádění operací s COM komponentou ve Visual Studiu .NET je velice podobné programování jakékoliv jiné .NET komponenty.

Nastal čas, abychom si pověděli pár slov o výkonnostních charakteristikách časného vázání objektů. Jelikož je při vzájemné spolupráci s COM komponentou vytvářen speciální proxy objekt, můžeme očekávat jisté výkonové penalizace. Tyto, přestože jsou přítomné, nehrají v naprosté většině případů zásadní roli. Důvodem je skutečnost, že ačkoliv musí CPU spočítat více instrukcí, pořád jde o menší sadu příkazů, než kdybyste prováděli nepřímá volání přes rozhraní **IDispatch**. Řečeno jinými slovy, použitím časného vázání objektů více získáte, než ztratíte. Zcela nejlepším řešením je oprostit se od používání COM komponent a pracovat jenom s moderními řízenými objekty. Tento algoritmus není bohužel vždycky realizovatelný, protože mnoho vývojářů a společností investovalo do vývoje COM komponent mnoho času, úsilí a prostředků, takže je pochopitelné, že chtějí, aby jim tyto investice přinášely své plody i nadále.

Metody OnStartPage a OnEndPage

Další z potenciálně problémových oblastí je použití metod **OnStartPage** a **OnEndPage**. Jestliže jste při programování interních objektů ASP pracovali s těmito metodami, budete muset použít direktivu **ASPCOMPAT** a zavolat metodu **Server.CreateObject** pro vytvoření objektu komponenty pomocí techniky časného vázání.

```
Dim Obj As MyObj
Obj = Server.CreateObject(MyObj)
Obj.MyMethodCall()
```

Všimněte si, že namísto „ProgID“ jsme použili název konkrétního typu přesně podle specifikací techniky časného vázání objektů. Ještě jednou připomínám, že pro správný běh kódu je nutné začlenit do aplikačního projektu referenci na cílovou COM komponentu. Uvedený příklad byl ukázkou jediného případu, v němž je nutné i nadále používat metodu **Server.CreateObject**.

COM: Shrnutí

Tab. 4.2 přináší všechny relevantní informace, s nimiž byste se měli obeznámit, jestliže plánujete i do budoucnosti využívat vaše COM komponenty tak efektivně, jak to jenom jde.

Tab. 4.2: ASP.NET a COM komponenty

TYP/METODA COM KOMPONENTY	POSTUP V ASP.NET
Uživatelské STA komponenty (Komponenty připravené ve Visual Basicu nebo jiné komponenty, které pracují v prostředí jednovláknových komnat)	Použijte **ASPCOMPAT** a časné vázání objektů.
Uživatelské MTA komponenty (ATL komponenty nebo uživatelské COM komponenty, které pracují v prostředí vícevláknových komnat)	Nepoužívejte **ASPCOMPAT**, použijte časné vázání objektů.
Interní objekty přístupné přes objekt **ObjectContext**	Použijte **ASPCOMPAT** a časné vázání objektů.
Metody OnStartPage a OnEndPage	Použijte **ASPCOMPAT** a metodu **Server.CreateObject(SpecifikaceTypu)**

Tato nastavení jsou platná i v případě, kdy jsou vaše komponenty rozmísťovány prostřednictvím COM+.

ZMĚNY V KONFIGURACI APLIKACÍ

V prostředí ASP jsou informace týkající se konfiguračních nastavení všech webových aplikací uloženy v systémovém registru a v metabázi IIS. Prohlížení a upravování takovýchto informací není nic jednoduchého, zvlášť když nejsou na webovém serveru nainstalovány ty správné administrátorské nástroje. Technologie ASP.NET přichází se zbrusu novým konfiguračním modelem, jenž je založen na srozumitelných a dobře čitelných XML souborech. Každá ASP.NET webová aplikace disponuje svým vlastním souborem Web.Config, který je uložen v hlavní aplikační složce. Modifikací obsahu souboru můžete realizovat konfiguraci uživatelských a bezpečnostních voleb aplikace a také můžete ovlivňovat způsob, jakým aplikace pracuje.

Jestliže jste jako já, budete mít chuť spustit program pro správu IIS a provést modifikaci nastavení vaší ASP.NET aplikace. Důležité je si uvědomit, že nyní máme k dispozici dva zcela odlišné konfigurační modely. Pokud budeme abstrahovat od některých bezpečnostních nastavení, můžeme prohlásit, že drtivá většina dalších konfiguračních voleb ASP bude v ASP.NET ignorována. Veškerá konfigurační data bude potřebné vložit do XML souboru s názvem Web.Config.

Popis konfigurace aplikací v .NET prostředí by vydal na samostatnou kapitolu, ne-li knížku, a proto se jím nebudeme podrobněji zabývat. Místo toho vám ukážu některá zajímavá nastavení, která můžete ve vašem konfiguračním souboru provést. Mějte prosím na paměti skutečnost, že existují také mnohé další konfigurační volby, o kterých zde nemluvím.

Tab. 4.3: Ukázka konfiguračních voleb, které jsou dostupné v XML souboru Web.Config

KONFIGURAČNÍ VOLBA	CHARAKTERISTIKA
<appSettings>	Upravuje uživatelská konfigurační nastavení aplikace.
<authentication>	Upravuje nastavení autentikace ASP.NET aplikace.
<pages>	Identifikuje konfigurační nastavení, která se vážou k specifickým volbám stránky.
<processModel>	Provádí konfiguraci nastavení modelu procesu na systémech vybavených IIS.
<sessionState>	Specifikuje možnosti nastavení stavu relace.

Bázová knihovna tříd s názvem .NET Framework Class Library obsahuje třídy, které vám umožní přistupovat k uvedeným konfiguračním nastavením pomocí programového kódu.

ZMĚNY V ŘÍZENÍ STAVU APLIKACÍ

Jestli vaše aplikace používá interní objekty **Session** a **Application** pro uchovávání informací o svém stavu, budete určitě potěšeni, když se dozvíte, že tyto objekty lze aktivovat i v ASP.NET bez jakýchkoliv potíží. Navíc, ASP.NET vám nyní nabízí rovněž další možnosti jak ukládat data, která souvisejí se stavem webových aplikací.

Možnosti řízení stavu aplikace

Technologie ASP.NET poskytuje dodatečné konfigurační volby, pomocí nichž můžete uchovávat informace o stavu aplikace, a to nejenom na jednom webovém serveru. Řídit stav aplikace totiž můžete také napříč celou webovou farmou.

Volby týkající se řízení stavu aplikace můžete naprogramovat prostřednictvím značky <**sessionState**>, kterou umístíte do souboru **Web.Config**.

```
<sessionState
      mode="Inproc"
      stateConnectionString="tcpip=127.0.0.1:42424"
      sqlConnectionString="data source=127.0.0.1;user id=sa;password=" cookieless="false"
      timeout="20"
/>
```

Atribut **mode** určuje model správy ukládání informací o stavu aplikace. Na výběr máte z následujících možností: **Inproc**, **StateServer**, **SqlServer** a **Off** (tab. 4.4).

Tab. 4.4: Ukládání informací o stavu relace aplikace

VOLBA	CHARAKTERISTIKA
Inproc	Stav relace je uložen lokálně na serveru, podobně jak tomu bylo v ASP.
StateServer	Stav relace je uložen pomocí procesu služby, jenž se nachází na lokálním nebo vzdáleném serveru.
SqlServer	Stav relace je uložen v databáze nástroje SQL Server™.
Off	Sledování stavu relace je deaktivováno.

Atributy StateConnectionString a sqlConnectionString přicházejí do úvahy, jestli používáte jednu z těchto voleb. Pamatujte, že v rámci jedné aplikace můžete determinovat pouze jednu volbu pro konfiguraci správy informací o jejím stavu.

Skladování COM komponent

Když se rozhodnete uchovávat reference na COM komponenty pomocí objektu **Session** nebo **Application**, ztratíte tím možnost využít sil nových mechanizmů interakce s daty (prostřednictvím voleb **StateServer** nebo **SqlServer**). Místo toho budete muset použít volbu **Inproc**. Tato skutečnost je zčásti způsobena požadavkem, podle něhož musí být řízený objekt schopen serializace, což u objektů COM komponenty není jaksi možné. Moderní řízené komponenty však tuto podmínku splňují, a mohou být tedy použity pro implementaci nových modelů správy stavu webových aplikací.

Výkon

Je léty prověřenou pravdou, že za vyšší výkon je vždy nutné něčím zaplatit. Bohužel, i v dnešních dobách je otázka výkonu velice ošemetnou záležitostí. Pro některé programátory bude nejlepším řešením použití konfigurační volby **Inproc**, zatímco pro jiné **StateServer** či **SqlServer**. Vyberte si takovou volbu, která bude nejlépe vyhovovat vašim požadavkům a nezapomeňte provést výkonnostní testy, abyste se ujistili, že váš výběr byl opodstatněný.

Sdílení stavu aplikace mezi ASP a ASP.NET

I když je možné, aby vaše webová aplikace zahrnovala tak ASP jako i ASP.NET stránky, není možné mezi těmito platformami sdílet hodnoty datových členů objektů **Session** a **Application**. Můžete provést buďto duplikaci informací, nebo navrhnout a sestavit provizorní řešení, dokud nebude vaše aplikace zcela převedena do ASP.NET. Pokud jste objekty **Session** a **Application** využívali pouze zběžně, budete na tom ještě dobře. V opačném případě budete muset postupovat se vší opatrností a patrně bude zapotřebí vyvinout techniky, pomocí nichž budete sdílet předmětná data o stavu aplikace.

ZMĚNY SOUVISEJÍCÍ SE ZABEZPEČENÍM APLIKACÍ

Bezpečnost aplikací představuje velice důležitou oblast, kterou byste rozhodně neměli přehlížet. V této části vám ve stručnosti představím bezpečnostní systém platformy ASP.NET. Budete-li vyžadovat více podrobnějších informací, doporučuji vám přečíst si témata v dokumentaci Visual Studia .NET.

Celkové zabezpečení ASP.NET webové aplikace se odvíjí zejména od bezpečnostních konfiguračních nastavení, která jsou specifikována v XML souboru **Web.Config**. ASP.NET přitom spolupracuje také s Internetovými informačními službami (IIS) s cílem vytvoření jednotného a kompletního bezpečnostního modelu vaší webové aplikace. Bezpečnostní nařízení IIS budou aplikována na ASP.NET aplikaci stejně tak jako na tradiční ASP aplikaci. Nová technologie však pochopitelně přináší inovaci také v oblasti bezpečné správy aplikací.

Autentikace

ASP.NET podporuje čtyři typy autentikačních voleb (tab. 4.5).

Tab. 4.5: Techniky autentikace v ASP.NET

VOLBA	CHARAKTERISTIKA
Windows	ASP.NET používá autentikaci Windows.
Forms	Autentikace založená na cookies, pracující na báze klientských formulářů.
Passport	Autentikace je prováděná externí službou společnosti Microsoft s názvem Passport.
None	Není realizována žádná forma autentikace.

Při bližším pohledu na výše uvedenou tabulku zjistíte, že s jednotlivými volbami, kromě nové autentikace pomocí služby Passport, jste se mohli setkat již v ASP. Následující ukázka kódu předvádí, jak je možné aktivovat autentikaci Windows:

```
<configuration>
<system.web>
<authentication mode="Windows"/>
</system.web>
</configuration>
```

Autorizace

Když uživatelé vaší aplikace úspěšně prošli procesem autentikace, můžete se soustředit na jejich autorizaci, v rámci které budete ověřovat, zdali ten-který uživatel disponuje povolením pro přístup k zabezpečeným zdrojům. Výpis kódu, jenž je umístěn na nadcházejících řádcích, ukazuje, jak umožnit přístup pouze uživatelům s jmény „jkieley“ a „jstegman“.

```
<authorization>
   <allow users="NORTHAMERICA\jkieley, REDMOND\jstegman"/>
   <deny users="*"/>
</authorization>
```

Impersonace

Impersonace představuje proces, pomocí něhož objekt provádí exekuci kódu ve jménu jisté uživatelské entity. Impersonace v ASP umožňovala běh programového kódu ve jménu uživatele, jenž úspěšně prošel procesem autentikace. Alternativně mohou vaši aplikaci používat i uživatelé, kteří pracují anonymně pod speciální identitou. V prostředí ASP.NET není implicitně prováděna impersonace na základě příchozích požadavků. V tomto směru se ASP.NET od ASP liší. Budete-li trvat na realizaci impersonace, budete ji muset aktivovat v souboru **Web.Config** asi takto:

```
<identity>
   <impersonation enable = "true"/>
</identity>
```

ZMĚNY PŘI PŘÍSTUPU K DATŮM

Vývojová platforma .NET Framework s sebou přinesla nový pohled na správu dat, kterou má od nynějška na starosti technologie zvaná ADO.NET. ADO.NET je bezesporu žádanou inovací stávající technologie pro přístup k datům (ADO), ovšem bližší představení této technologie přesahuje možnosti této kapitoly. Ačkoliv můžete ve většině případů i nadále používat ADO tak, jak jste to dělali před příchodem platformy .NET Framework, bližší seznámení s ADO.NET vám vřele doporučuji. ADO.NET totiž zahrnuje všechny prostředky pro správu a uskutečňování operací s daty, které jste mohli kdy potřebovat.

PŘÍPRAVA PRO PŘECHOD SMĚREM K ASP.NET

V předcházejících kapitolách jsme společně odhalili širokou škálu kritických oblastí, s nimiž pravděpodobně přijdete do kontaktu v případě, že se rozhodnete pro přechod na platformu ASP.NET. Možná si lámete hlavu nad tím, co byste mohli pro lepší připravenost na ASP.NET udělat ještě dnes. Proto jsem pro vás připravil seznam námětů, které když budete brát v potaz, bude proces migrace vaší webové aplikace do ASP.NET mnohem plynulejší a rychlejší. Některé z uváděních doporučení budou přínosná i pro stávající ASP aplikace, o jejichž inovaci směrem k ASP.NET zatím neuvažujete.

Použití volby Option Explicit

Aktivace volby **Option Explicit** byla vždy dobrým nápadem, ovšem ani v dnešních časech ji nevyužívají všichni programátoři. Zařadíte-li do vašeho kódu řádek s volbou **Option Explicit**, přinejmenším získáte přehled o deklarování a použití vašich proměnných. Když se ovšem jednou rozhodnete pro ASP.NET, navrhuji vám, abyste raději přešli na volbu **Option Strict**. V programovacím jazyce Visual Basic .NET je volba **Option Explicit** implicitně aktivní, ovšem když zapnete také volbu **Option Strict**, bude prováděna rovněž kontrola, zdali jsou všechny proměnné explicitně deklarovány pomocí specifických datových typů. Přiřazování datových typů deklarovaným proměnným vyžaduje jistou práci navíc, ovšem z dlouhodobého hlediska se vám tato investice vrátí i s pomyslnými úroky.

Vyhněte se používání výchozích vlastností

Jak jsme si již řekli, výchozí vlastnosti nejsou v ASP.NET webových aplikacích podporovány. Na druhou stranu, explicitní přístup k cílovým vlastnostem není až tak namáhavý, jak se může na první pohled zdát. Kromě toho se váš kód stane lépe čitelným a přístupným pro další možné inovace v budoucnosti.

Používejte závorky a příkaz Call

Moje rada zní: Používejte závorky a příkaz **Call** kdykoliv je to možné, jak jsme si předvedli během naší exkurze. Koneckonců, v ASP.NET je používání závorek výslovně vyžadováno. Zařazení příkazu **Call** vám pomůže povznést disciplinu při psaní řádků zdrojového kódu a připraví vám dobrou výchozí pozici pro budoucí inovace.

Vyvarujte se odkazování na jiné soubory

Vím, že toto doporučení se lépe říká, než uskutečňuje v praxi. Nicméně, pokud je to jenom trochu možné, měli byste se raději vyhýbat vnořování odkazů na dodatečné soubory. Cílem takového počínání je eliminovat ty oblasti, v nichž dochází k vícenásobnému odkazování na data, která jsou uložena v rozdílných souborech. Pokud používáte techniku odkazování na jiné soubory, může se vám po jistém čase stát, že váš kód bude závislý na nějaké globální proměnné, která bude ovšem deklarována v jiném souboru. Přístup k této proměnné pak můžete získat jenom pomocí zahrnutí dodatečného souboru, jenž obsahuje odkaz na soubor s deklarací veřejné proměnné. Tyto vícenásobné reference jsou opravdu neužitečné a neefektivní, a proto se jim vyhýbejte velkým obloukem.

Jestliže přecházíte na ASP.NET, je velice pravděpodobné, že hodláte přenášet také vaše globální proměnné a procedury, které budete posléze ukládat do knihoven tříd. Věřte, že vaše práce bude mnohem snadnější, když budete vědět, kde se deklarace uvedených entit nacházejí.

Uspořádejte funkce do samostatných souborů

Jednou ze strategií, které se používají v migračním procesu, je přenos všech funkcí a potřebného programového kódu, jenž je umístěn v souborech na straně serveru, do moderních knihoven tříd jazyků Visual Basic nebo C#. Tento přístup vám konečně umožní uložit kód tam, kam patří, a sice do objektů (což je mnohem lepší řešení, než používat vícenásobné interpretované ASP soubory). Když vhodně uspořádáte programový kód s výhledem do budoucnosti, ušetříte nemalé množství času. Nejlépe uděláte, když logicky uspořádáte skupiny procedur a funkcí do samostatných souborů, které můžete posléze využít při přípravě kolekce tříd ve Visual Basicu nebo v jazyce C#.

Disponujete-li množstvím globálních proměnných a konstant v souborech na straně serveru, zkuste popřemýšlet, zdali by nebylo efektivnější je umístit do jednoho samostatného souboru. V ASP.NET budete poté moci sestavit třídu, která bude domovem pro vaše globální proměnné a konstanty. Tak získáte čistší a lépe spravovatelný systém.

Oddělte programový kód webových stránek od jejich obsahu

Separování zdrojového kódu od obsahu webových stránek je dalším skvělým nápadem, který byste mohli zužitkovat. Jednoduše vezměte všechen kód a uložte jej do jednoho souboru, a poté přidejte odkaz na tento soubor do hlavního HTML dokumentu. Oddělení programového kódu od obsahové vrstvy skýtá prostor pro lepší správu a uspořádanější vzhled kódu. Navíc, tento model ideálně pracuje i pod křídly ASP.NET.

Nedeklarujte funkce uvnitř bloků <%%>

Deklarování funkcí, které se nacházejí uvnitř bloků <%%>, není v ASP.NET povoleno. Vaše funkce byste měli deklarovat uvnitř bloků, které jsou uvedeny značkou <**script**>. Ukázku této techniky můžete najít v kapitole *Strukturální změny*, která se nachází výše.

Nepoužívejte překládané procedury a funkce

Jak jsem se zmínil již dříve, použití překládaných funkcí není tou nejlepší volbou. Můžete-li změnit svůj kód už nyní, měli byste pamatovat na použití bloků **Response.Write** pokaždé, když se rozhodnete zkonstruovat tento typ funkcí či procedur.

Zdroje uvolňujte explicitně (aktivováním metody Close)

Ujistěte se, že pokaždé, když již nepotřebujete vytvořené objekty, uvolníte alokované zdroje aktivací metody **Close**, případně pomocí jiné ekvivalentní metody. Kromě toho, Visual Basic a VBScript se při dealokování zdrojů chovají jinak, než je tomu v .NET prostředí. Jak jistě víte, v „před .NET“ dobách byly objekty uvolňovány okamžitě, ovšem nyní je práce s nimi plně v kompetenci softwarové služby, které se říká **Garbage Collection**. Tato služba neprovádí deterministickou dealokaci objektů, což znamená, že si nemůžete být zcela jisti tím, kdy k uvolnění alokovaných zdrojů vůbec dojde. Můžete však **Garbage Collection** naznačit, že si přejete uvolnit jisté paměťové zdroje.

Na míchání programovacích jazyků raději zapomeňte

Poslední dobrá rada říká, že byste se měli vystříhat míchání jazyků VBScript a JScript při psaní kódu, jenž bude běžet na straně serveru, uvnitř jedné webové stránky. Tento programátorský přístup je ve všeobecnosti poněkud nestabilní. Navíc, kvůli novému modelu komplice, jenž přináší ASP.NET, je povoleno použití jenom jednoho programovacího jazyka (uvnitř bloku <%%>) na jednu webovou stránku. Skripty na straně klienta můžete emitovat tak, jak jste to dělali dosud.

SHRNUTÍ

Je pravdou, že přechod na platformu ASP.NET nemusí být vždy nejjednodušší. Jak jste se mohli dočíst, existuje několik nástrah, o nichž byste měli být informováni ještě předtím, než začnete připravovat inovaci vaší aplikace. Implementace většiny změn, na které jsem se snažil poukázat, by pro vás neměla být vůbec přehnaně obtížná.

Spravujete-li rozsáhlé webové sídlo, budete asi překvapeni, když po ukončení procesu inovace stávajících ASP stránek uvidíte, kolik nepoužitelného a neefektivního kódu je zapotřebí odstranit či upravit. Poté, co absolvujete tuto zatěžkávající zkoušku, můžete se vydat vstříc novým a vzrušujícím prvkům, které technologie ASP.NET a také .NET platforma nabízejí.

O AUTOROVI

Jim Kieley pracuje jako softwarový vývojář ve společnosti Microsoft. Již od založení .NET platformy úzce spolupracuje s členy týmu Visual Studia. Zaměřuje se především na vývoj ASP.NET webových aplikací a také jiných typů řešení, která běží na platformě .NET Framework. Jim je k zastižení na adrese *jkieley@microsoft.com*.

5

Vylepšování dokonalosti: Budování aplikací pro systém Windows s jazykem Visual Basic .NET

Microsoft Corporation

SHRNUTÍ

Programovací jazyk Visual Basic byl častou volbou programátorů, kteří se rozhodli budovat svá řešení prostřednictvím systému rychlého vývoje aplikací (RAD). Pomocí vizuálního programování bylo možné navrhnout kompletní projekty mnohem rychleji, než tomu bylo v jiných programovacích jazycích. Visual Basic .NET si přináší vytvořenou dokonalost, ovšem ještě dále ji vylepšuje. Vytváření aplikací pro systém Windows je tak ještě jednodušší než kdykoliv předtím.

Cíle

- Navrhnout a zkonstruovat jednoduchou aplikaci pro systém Windows za pomoci Visual Basicu .NET.
- Demonstrovat použití několika nových ovládacích prvků a vlastností, které ve významné míře minimalizují čas potřebný na vývoj.

ÚVOD

Vývojáři ve Visual Basicu měli po dlouhá léta tajemství. Mohli totiž vytvářet aplikace pro systém Windows s rychlostí, o níž si mnozí jiní mohli nechat jenom zdát. Zatímco někteří programátoři nahlíželi na Visual Basic jako na programovací jazyk na hraní a poukazovali na jisté syntaktické nedostatky, které se podle nich nemohli vůbec rovnat s „čistou“ sílou jazyků jako je třeba C++, jiní si užívali volných víkendů a večeří v rodinném kruhu s dobrým vědomím, že veškeré závazné mezníky vývoje vysoce kvalitních aplikací budou splněny řádně a načas. Ačkoliv nikdo přesně neví jak, tajemství Visual Basicu bylo odhaleno. Věrní uživatelé Visual Basicu tak byli vzápětí pověřováni čím dál náročnějšími a rozsáhlejšími projekty, jejichž požadavky se pravidelně měnily a termíny se zkracovaly. Vysvitlo, že ti, jejichž mínění o Visual Basicu nebylo příliš valné, vůbec nevěděli, jak je možné tento programovací jazyk použít při budování sofistikovaných aplikací. Veškeré očekávání se proto zaměřili na Visual Basic.

Visual Basic .NET je vývojovým nástrojem nové generace, který nejenom že splňuje všechna očekávání, ale stále si zachovává tradičnou vývojovou linii, kterou přinesl Visual Basic. To ovšem není všechno: Visual Basic .NET nabízí vyšší výkon a vylepšenou flexibilitu, čímž umlčuje všechny kritické hlasy a dovoluje jít vývojářům až na hranice programátorských možností. Přestože pomocí Visual Basicu .NET můžete vyvíjet nové typy projektů a využívat všech výhod integrace s platformou .NET Framework, pro velkou skupinu programátorů bude i nadále vývoj „okenních“ aplikací jejich oblíbeným šálkem čaje. Když si uvědomíme tuto skutečnost, můžeme se vydat na exkurzi do prostředí jazyka Visual Basic .NET. Uvidíte, co nového přináší nová verze Visual Basicu a jak lze s její pomocí urychlit vývoj aplikací pro systém Windows.

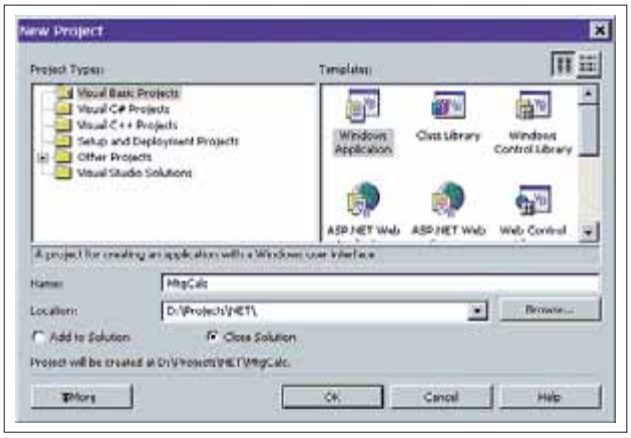
- **Poznámka**: Pokud jste v jazyce Visual Basic .NET nováčky, přečtěte si dokument Visual Basic .NET - *Better for Today, Ready for Tomorrow (Visual Basic .NET - lepší pro dnešek, připraven pro zítřek)*, jenž poskytuje přehled všech jazykových rozšíření na vysoké úrovni.

APLIKACE HYPOTEKÁRNÍ KALKULAČKA

Nejlépe si sílu a lehkost Visual Basicu .NET při vytváření aplikací pro systém Windows ukážeme jednoduše tak, že jednu skutečnou aplikaci opravdu sestrojíme. Pro naše potřeby si vystačíme s jednoduchou kalkulačkou půjček pro hypotekárního věřitele, jejíž pomocí může být potenciální vypůjčovatel informován o tom, jakými faktory je ovlivňována jeho měsíční splátka. Do úvahy budeme brát následující faktory: počáteční obnos, lhůta splatnosti, úroková míra a suma zálohy. Naše aplikace bude zaměřena na osoby, které si chtějí vypůjčit peníze prvně a které hledají nové bydliště. Tito vypůjčovatelé přitom potřebují jednoduchým způsobem zjistit rozsah cen, které budou vyhovovat jejich měsíčním příjmům a také přesný odhad hotovosti, který budou muset nashromáždit, aby mohli splatit zálohu. Z uvedeného zaměření cílového publika aplikace jasně plyne, že jednoduchost použití a uživatelská přívětivost budou tvořit nejvyšší priority. Náš zákazník disponuje existujícím zdrojovým kódem jazyka Visual Basic 6 (jenž provádí kalkulace) a hodlá přenést připravenou obchodní logiku do nového projektu.

ZAČÍNÁME

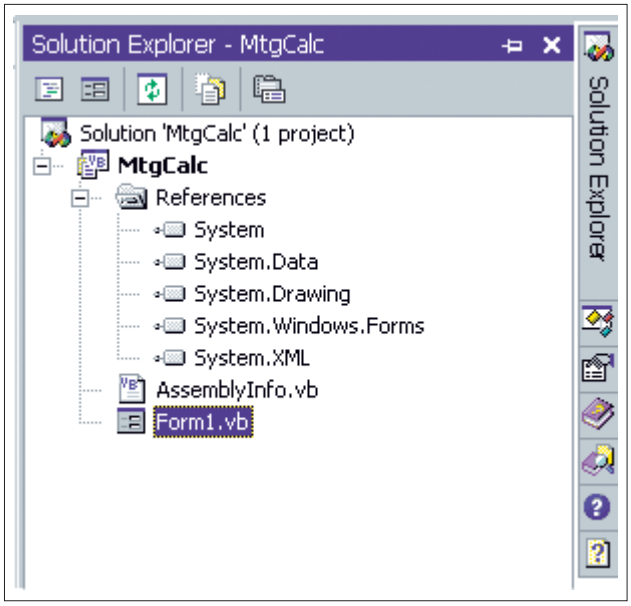
Od verze .NET je Visual Basic součástí integrovaného vývojového prostředí (Integrated Development Environment, IDE) Visual Studia .NET. Jednou z nejlepších vlastností IDE je jeho signifikantní podpora pro pokročilé úpravy podle přání programátora. Všechny kroky, které budou následovat, předpokládají, že na vašem počítači je přítomná standardní instalace Visual Studia .NET. Jestliže bude zmíněna reference na nějaké okno, které nevidíte, obecně platí, že aktivací příslušného příkazu z nabídky **View** můžete dané okno opětovně zobrazit. Výjimkou jsou rozličná okna nápovědy a okno úvodní stránky (**Start Page**), které můžete ovládat prostřednictvím příkazů nabídky **Help**. Práce na našem projektu zahájíme spuštěním Visual Studia .NET a klepnutím na tlačítko **New Project**, které je umístěno na stránce **Start Page**. Alternativně můžete také otevřít nabídku **File**, ukázat na příkaz New a kliknout na položku **Project**.



Obr. 5.1: Vytváření nového projektu

Máte-li zobrazeno dialogového okno **New Project**, klepněte na položku **Visual Basic Projects** (je to první položka ve stromové struktuře nalevo). Z pravého pole vyberte projektovou šablonu s názvem **Windows Application** (Standardní aplikace pro systém Windows). Jméno naší aplikace bude **MtgCalc** a jak si můžete všimnout, toto jméno je zapsáno v textovém poli **Name**. Chcete-li, můžete také vybrat cílovou složku, do níž budou uloženy projektové soubory (textové pole **Location**). Jste-li s provedenými změnami spokojeni, klepněte na tlačítko **OK**.

Dialogové okno **Solution Explorer** (Průzkumník řešení) slouží pro pohled na hierarchickou strukturu našeho projektu. Jestliže se podíváte na obsah tohoto okna, můžete vidět, že některé soubory byly vytvořeny zcela automaticky. Jeden rozdíl oproti předcházející verzi Visual Basicu spočívá v tom, že Visual Studio .NET automaticky vytváří řešení (položka **Solution**) a vytvořený projekt ukládá do tohoto řešení. Ve Visual Studiu .NET jsou řešení ekvivalenty projektových skupin, které znáte z Visual Basicu 6. Řešení dovolují vývojářům seskupovat příbuzné projekty a otevírat je současně v integrovaném vývojovém prostředí. Naše aplikace bude opravdu jednoduchá, a proto nám bohatě postačí pouze jeden jediný projekt.



*Obr. 5.2: Dialogové okno **Solution Explorer** (Průzkumník řešení)*

Složka projektu **MtgCalc** obsahuje položku **References**, která je, co do významu, identická s příkazem **References** nabídky **Project** ve Visual Basicu 6. Jak můžete vidět, jisté reference byly pro nás vytvořeny implicitně. Kolekce vytvořených referencí je závislá na typu projektové šablony, kterou použijete. V tomto případě byly do projektu přidány ty reference, které jsou důležité pro činnost standardních aplikací pro Windows. Nové reference mohou být přidány klepnutím na jméno projektu pravým tlačítkem myši a vybráním příkazu **Add Reference**.

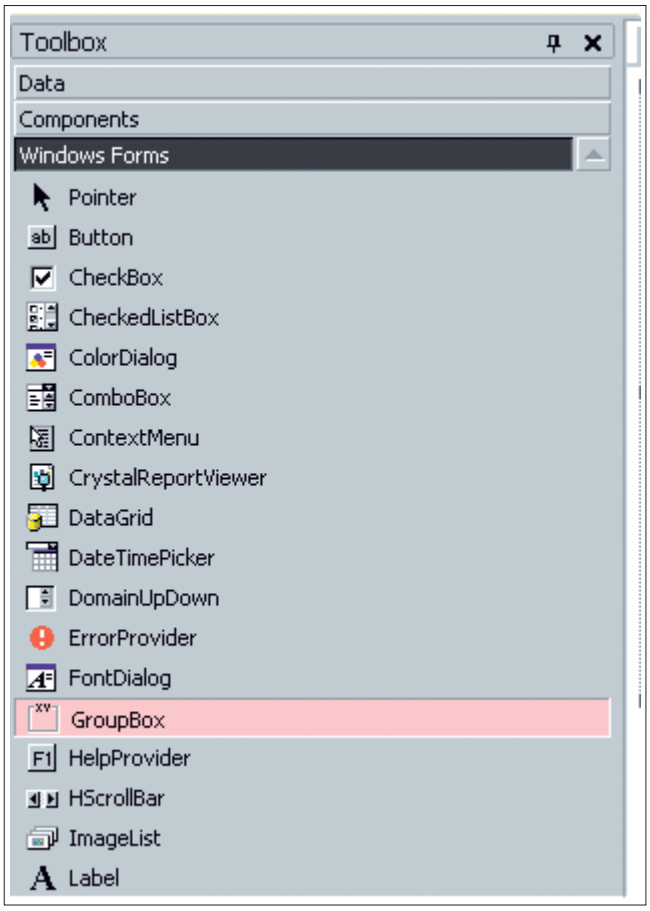
Odhlédneme-li od referencí, zjistíme, že do našeho projektu byly přidány také dva další soubory. Jeden z nich nese název **Form1** (jako ve Visual Basicu 6), přičemž obsahuje programový kód třídy formuláře. Jak upravit podobu formuláře si ukážeme v následující podkapitole. Druhým kandidátem je nový typ souboru, jenž je pojmenován jako **AssemblyInfo.vb**, ovšem tento si nebudeme pro tuto chvíli všímat.

ROZVRŽENÍ FORMULÁŘE

Máme-li vytvořený projekt standardní aplikace pro Windows, můžeme se soustředit na vytváření vzhledu hlavního formuláře. Když v okně **Solution Explorer** poklepete na položku **Form1**, Visual Basic .NET otevře formulář v okně vizuálního návrháře. Najeďte kurzorem myši na plochu formuláře a stiskněte pravé tlačítko myši. Objeví se kontextová nabídka, z níž vyberte příkaz **Properties**. Popisek **Form1** v titulkovém pruhu formuláře může jenom stěží vystihnout charakter vyvíjené aplikace, a proto jej prostřednictvím vlastnosti **Text** formuláře změníme na **Mortgage Calculator** (Hypotekární kalkulačka). Abychom získali na formuláři více prostoru, upravíme také složenou vlastnost **Size**, která představuje velikost formuláře. Šířku (**Width**) formuláře nastavíme na hodnotu 480 pixelů a výšku (**Height**) pak na 400 pixelů. Nyní můžeme začít s přidáváním instancí některých ovládacích prvků.

Návrh vizuální stránky formuláře je v zásadě stejný jako ve Visual Basicu 6, ovšem signifikantně se rozšířil počet ovládacích prvků, jejichž instance můžete na plochu formuláře umístit. V dialogovém okně **Toolbox** (Souprava nástrojů) můžete vidět „léty prověřené“ zástupce, mezi nimiž nechybí **Label** (Popisek), **Button** (Tlačítko), **TextBox** (Textové pole) a jiné základní ovládací prvky. Je ovšem nutno zdůraznit, že i tyto prvky byly vylepšeny a nabízejí tak vylepšenou funkcionalitu. Kromě již známých ovládacích prvků a komponent se můžete setkat také s novinkami, které jste museli v minulosti hledat v produktovém portfoliu jiných společností. Nyní jsou však nedílnou součásti Visual Basicu .NET. V práci s novými ovládacími prvky najdete zalíbení, a to zejména kvůli vyspělé konzistenci. Vlastnosti a metody, které provádějí podobné funkce jsou od nynějška pojmenované stejně u všech ovládacích prvků. Kupříkladu, jestliže ovládací prvek disponuje popiskem, v jeho výbavě se určitě nachází i vlastnost **Text**. Je přitom zcela nepodstatné, zda jde o prvek **Button**, **Label** či **TextBox**.

Budete-li chtít vkládat instance ovládacích prvků na formulář, určitě oceníte, když budete mít snadný přístup k oknu **Toolbox**. Je-li nastaveno automatické skrývání okna (což znamená, že okno zmizí, když na něj nebude spočívat kurzor myši), najeďte myší na ikonu **Toolbox**, která se nachází na levé straně obrazovky a okno se objeví. Klikněte na ikonu připínáčku, čímž okno přišpendlíte a zabezpečíte tak, aby bylo viditelné po celou dobu návrhu aplikace.



*Obr. 5.3: Dialogové okno **Toolbox** (Souprava nástrojů)*

Nyní je ta pravá chvíle pro umístění několika instancí ovládacích prvků na formulář. V okně **Toolbox** klepněte na záložku **Windows Forms** a postupujte následovně:

1. Klepněte na ikonu ovládacího prvku **GroupBox** (Rámeček), ovšem pořád držte stisknuté levé tlačítko myši. Nyní přetáhněte kurzor myši do levého horního rohu formuláře a uvolněte stisknuté tlačítko. Vlastnosti **Name** a **Text** vložené instance ovládacího prvku změňte na **gbMortgageInfo**, resp. **Mortgage Info**.

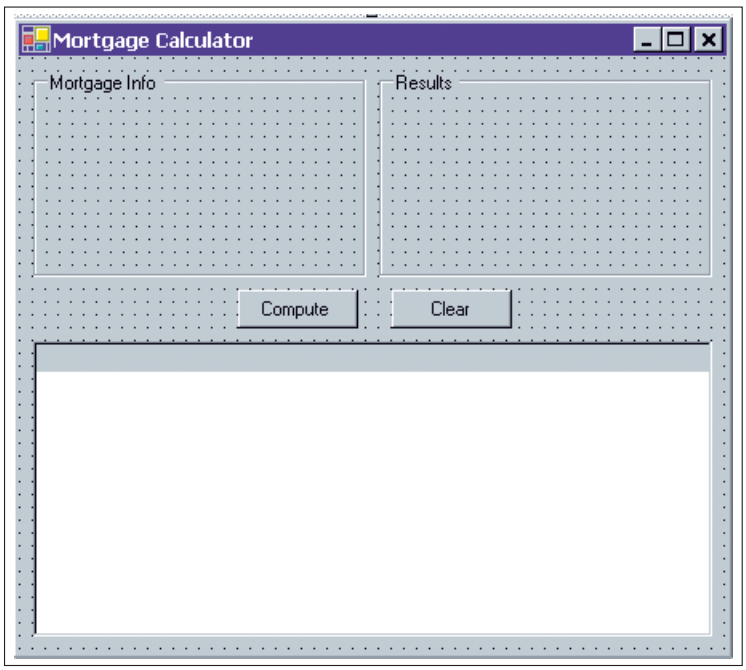
2. Napravo od stávající instance ovládacího prvku GroupBox umístěte další instanci tohoto prvku. Do vlastnosti **Name** instance vložte textový řetězec **gbResults**. Vlastnost **Text** naplňte textem **Results**.

3. Přetáhněte na formulář dvě instance ovládacího prvku **Button** a uspořádejte je jednu vedle druhé pod instance ovládacího prvku **GroupBox**. Vlastnosti tlačítek upravte následovně:

Button1	**Name** = btnCompute	**Text** = Compute
Button2	**Name** = btnClear	**Text** = Clear

4. Nyní přidejte do spodní poloviny formuláře jednu instanci ovládacího prvku **ListView**. Instanci pojmenujte (vlastnost **Name**) jako **lvAmortization** a její vlastnost **View** nastavte na hodnotu **Details**.

V tuto chvíli by měl váš formulář vypadat tak, jako jeho kolega z obr. 5.4.



Obr. 5.4: Vzhled formuláře po vložení instancí ovládacích prvků

Abychom mohli získávat vstupní údaje od uživatelů, vložíme do rámečků (instancí ovládacího prvku **GroupBox**) další potřebné instance jiných ovládacích prvků. Postupujte podle níže uvedeného algoritmu:

1. Do rámečku s názvem **gbMortgageInfo** vložte jednu instanci ovládacího prvku **Label**.

2. Do téhož rámečku přidejte ještě další tři instance uvedeného ovládacího prvku, které umísťujte pod sebe.

3. Vlastnost **Text** všech čtyř instancí upravte takto:

Label1	**Text** = Purchase Price:
Label2	**Text** = Down Payment:
Label3	**Text** = Interest Rate:
Label4	**Text** = Term (Years):

4. Vyberte všechny instance ovládacího prvku **Label**. Toho docílíte tak, že nejdříve klepnete na instanci **Label1**, stiskněte klávesu SHIFT a za jejího současného držení budete klepat na všechny další instance, tedy **Label2**, **Label3** a **Label4**.

5. Když jste byli úspěšní, měli byste nyní mít všechny čtyři instance ovládacího prvku **Label** vybrané. Otevřete nabídku **Format**, ukažte na položku **Align** a vyberte příkaz **Lefts**. Všimněte si, že všechny instance jsou teď zarovnané podél jejich levé strany. Mimochodem, zarovnání nalevo můžete uskutečnit i aktivací příslušné položky na panelu nástrojů **Layout**.

6. Pokračujte vložením dvou instancí ovládacího prvku **TextBox** do rámečku **gbMortgageInfo**. Textová pole umístěte tak, aby se každé z nich nacházelo napravo od prvních dvou instancí ovládacího prvku **Label**. Vlastnosti vložených instancí modifikujte tímto způsobem:

TextBox1	**Name** = txtPurchasePrice	Text = ""
TextBox2	**Name** = txtDownPayment	Text = ""

7. Na formulář vložte dvě instance ovládacího prvku **NumericUpDown**. Tyto instance umístěte napravo od spodních dvou popisků a pojmenujte je jako **numInterestRate** a **numTerm**.

8. Zarovnejte tyto čtyři instance nalevo tak, jak jste do udělali s instancemi ovládacího prvku **Label** v 5. bodě tohoto postupu.

I když můžeme dovolit uživatelům naší aplikace zadávat libovolné číselní hodnoty do textových polí **Purchase Price** a **Down Payment**, určitě nebudeme chtít tuto benevolenci uplatnit také při textových polích **Interest Rate** a **Term**. Kromě toho, náš zákazník se bude cítit lépe, když bude moci hodnotu úrokové míry zvětšovat po jedné čtvrtině. Ačkoliv můžeme všechny uvedené požadavky ošetřit pomocí programového kódu, máme k dispozici lepší řešení. Pomocí instancí nového ovládacího prvku **NumericUpDown** a nastavením několika málo vlastností splníme veškeré stanové podmínky, a to vše bez napsání jediného řádku kódu!

1. Vyberte instanci s názvem **numInterestRate** a nastavte její vlastnosti následovně:

DecimalPlaces	2
Increment	.25
Minimum	0
Maximum	20
Value	7

2. Označte instanci s názvem **numTerm** a upravte nastavení těchto vlastností:

Minimum	0
Maximum	50
Value	30

Pro tento případ jsme omezili hodnoty úrokové míry na platný interval <0.00 %, 20.00 %> s tím, že standardně bude vybrána sedmiprocentní úroková míra. Hodnota úrokové míry může být upravována po dílčích hodnotách, které jsou rovny 0.25 %. Abychom byli zcela úplní, musíme dodat, že lhůta splatnosti bude maximálně 50 let, standardně pak 30 let.

Rámeček **gbResults** bude využíván pro zobrazování výsledků kalkulací. Jeho design je takřka shodný se vzhledem rámečku **gbMortgageInfo**, kromě toho, že obsahuje čtyři textová pole místo dvou a rovněž dvě instance ovládacího prvku **NumericUpDown**. Jelikož by uživatel neměl mít možnost jakkoliv měnit vypočtené výsledky, budou tyto zobrazeny jenom pro čtení (**ReadOnly**).

1. Do rámečku **gbResults** vložte čtyři instance ovládacího prvku **Label**, zarovnejte je nalevo a upravte jejich vlastnost **Text** následovně:

Label1	**Text** = Loan Amount:
Label2	**Text** = Monthly Payment:
Label3	**Text** = Total Interest:
Label4	**Text** = Total Payments:

2. Na formulář přidejte čtyři textová pole (instance ovládacího prvku **TextBox**) a uspořádejte je tak, aby se každé textové pole nacházelo napravo od jednotlivých popisků. Zarovnejte je nalevo a upravte jejich vlastnosti takto:

TextBox1	**Name** = txtLoanAmount	**Text** = ""	**ReadOnly** = True
TextBox2	**Name** = txtMonthlyPayment	**Text** = ""	**ReadOnly** = True
TextBox3	**Name** = txtTotalInterest	**Text** = ""	**ReadOnly** = True
TextBox4	**Name** = txtTotalPayments	**Text** = ""	**ReadOnly** = True

Pro identifikaci sloupců v amortizačním plánu budeme muset přidat šest záhlaví. Budete-li chtít nastavit záhlaví, klepněte na instanci ovládacího prvku **ListView** pravým tlačítkem myši a z kontextové nabídky vyberte příkaz **Properties**.

1. Vyhledejte vlastnost **Columns** a klepněte na tlačítko se třemi tečkami (...). Vzápětí se objeví dialogové okno **ColumnHeader Collection Editor**.

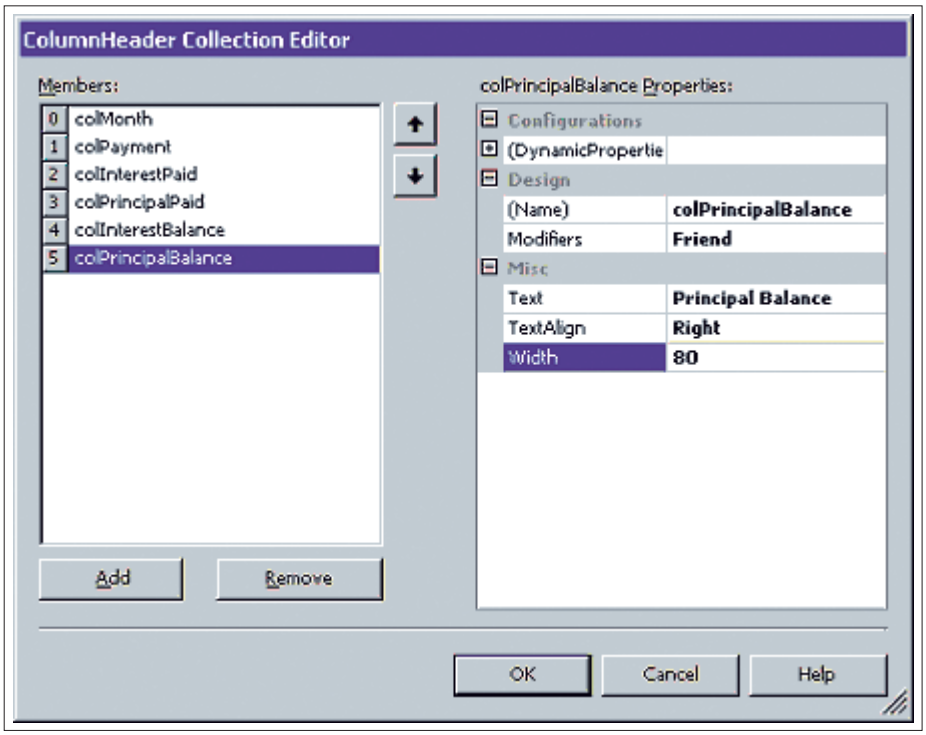
2. Klikněte na tlačítko **Add** a proveďte definici prvního sloupce, přičemž aktualizujte následující vlastnosti:

Name = colMonth	**Text** = Month	**Width** = 50

3. Přidejte dalších pět sloupců a jejich příslušné vlastnosti upravte takto:

Name = colPayment	**Text** = Payment	**Width** = 65
Name = colInterestPaid	**Text** = Interest	**Width** = 65
Name = colPrincipalPaid	**Text** = Principal	**Width** = 65
Name = colInterestBalance	**Text** = Interest Balance	**Width** = 75
Name = colPrincipalBalance	**Text** = Principal Balance	**Width** = 80

Finální podobu okna **ColumnHeader Collection Editor** můžete vidět na obr. 5.5.

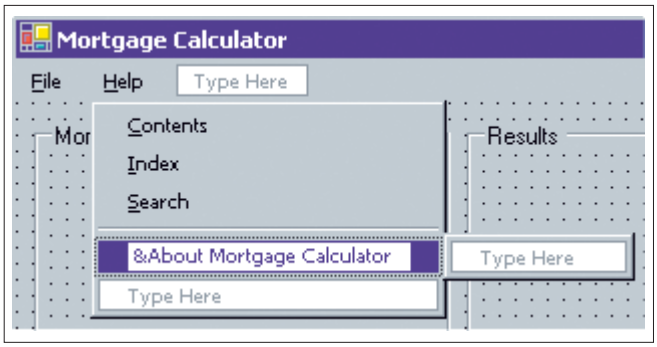


Obr. 5.5: Finální podoba okna **ColumnHeader Collection Editor**

NABÍDKY

Každá profesionální aplikace pro Windows je opatřena hlavním pruhem nabídek. Abychom také naší aplikaci vtiskli onen punc profesionality, použijeme i my pruh s nabídkami. Editor pro vytváření nabídek obsahoval již Visual Basic 6 a je zapotřebí říct, že tento editor nebyl vůbec špatný. Ovšem jakmile přijdete na chuť novému návrháři nabídek (**Menu Designer**) ve Visual Studiu .NET, zjistíte, že tento je mnohem lepší. Nový editor je nejenom vizuálnější, ale také se snadněji používá a vytvořené nabídky jsou vskutku profesionální.

Posuďte sami: Když můžeme v době návrhu aplikace na formulář interaktivně umísťovat popisky, textová pole či tlačítka, proč bychom měli pracovat jiným stylem při návrhu nabídek? Odpověď zní: I nabídky můžeme vytvářet pomocí známých technik. Tvorbu nabídek, a to jak hlavních, tak kontextuálních, velice usnadňují dvě nové komponenty, jimiž jsou **MainMenu** a **ContextMenu**. Instance komponenty **MainMenu** se neobjeví přímo na formuláři, ale na podnosu komponent (**Component Tray**). Jde o speciální region, jenž je vyhrazen pro úschovu komponent. Ihned po vložení instance komponenty **MainMenu** se aktivuje návrhář nabídek, jehož pomocí můžete interaktivně navrhovat i velice složité nabídky.



Obr. 5.6: Návrhář nabídek v akci

Abyste vytvořili položku nabídky, stačí když klepnete na jeden z regionů, které jsou označené popiskem **Type Here** a zadáte text, jenž se bude objevovat na dané položce nabídky. Výborné je, že okamžitě vidíte výsledky své práce.

Podobně jako v minulosti, i ve Visual Basicu .NET mohou popisky položek nabídek obsahovat znak ampersand (&), jenž lze umístit před jistý textový znak, pomocí něhož budou uživatelé moci získat rychlý přístup k této položce nabídky. Nové položky můžete vkládat klepnutím pravým tlačítkem myši a zvolením příkazu **Insert New** z kontextové nabídky. Budete-li chtít vložit oddělovače, použijte příkaz **Insert Separator**, který naleznete ve stejné kontextové nabídce. Naši aplikaci opatříte hlavním nabídkovým pruhem pomocí následujícího postupu:

1. Vložte do aplikace novou instanci komponenty **MainMenu**.

2. Když se zobrazí návrhář nabídek (**Menu Designer**), klepněte na popisek **Type Here** a zadejte **&File**.

3. Klikněte na popisek **Type Here** pod nově přidanou položkou a zadejte **&Print**. Poté stiskněte klávesu ENTER.

4. Následující položku nabídky naplňte textovým řetězcem E&xit a stiskněte klávesu TAB nebo ENTER.

5. Na položku **E&xit** klepněte pravým tlačítkem myši a z kontextové nabídky vyberte příkaz **Insert Separator**.

6. Vyberte položku **File** hlavní nabídky a klepněte na popisek **Type Here**, jenž se nachází napravo od této položky. Zde zadejte text **&Help**, který bude označovat nabídku nápovědy.

7. Do nabídky **Help** přidejte následující položky:

    1. **&Contents**
    2. **&Index**
    3. **&Search**
    4. **&About Mortgage Calculator**

8. Na položku nabídky **About Mortgage Calculator** klepněte pravým tlačítkem myši a z kontextové nabídky vyberte příkaz **Insert Separator**.

Kromě popisků položek nabídek můžete měnit i samotné názvy těchto entit. Budete-li tedy chtít upravit pojmenování té-které položky nabídky, klepněte na ní pravým tlačítkem myši a vyberte příkaz **Edit Names**. Vedle textu položky se zobrazí i její pojmenování, které můžete velice snadno modifikovat pouhým přepsáním! Když jste s provedenými změnami spokojeni, opětovně aktivujte příkaz **Edit Names**, čímž přejdete do standardního módu návrhu nabídek.

Jakmile jsme navrhli základní strukturu nabídek, můžeme přikročit k psaní programového kódu, jenž bude řídit program poté, co uživatel vybere jistou položku nabídky. Ještě předtím však změníme pojmenování položek nabídky tak, abychom zavedli určitý systém, jenž nám bude dovolovat snadnou orientaci a lepší správu nabídek.

1. Pravým tlačítkem myši klepněte na kteroukoliv položku nabídky a vyberte příkaz **Edit Names**. V tu chvíli se vedle popisků nabídek objeví i jejich názvy v hranatých závorkách.

2. Klikněte na položku **Exit** a změňte její jméno na **miExit**.

3. Podle uvedeného postupu upravte názvy i dalších položek. Přitom se snažte, aby byly názvy položek co možná nejvýstižnější, čitelné a stručné.

4. Jste-li se zadaným úkolem hotovi, ještě jednou aktivujte příkaz **Edit Names**, čímž skryjete názvy položek nabídek a zobrazíte jejich popisky.

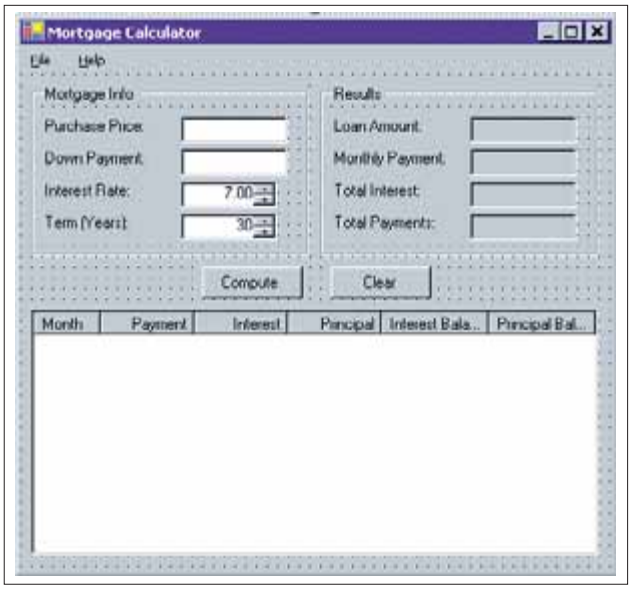
Na následujících řádcích si ukážeme, jak vytvořit zpracovatele události **Click** jedné položky nabídky. Kód, jenž bude uložen v těle tohoto zpracovatele, bude proveden pokaždé, když uživatel aktivuje příslušnou položku nabídky. Když pochopíte, jak vytvořit zpracovatele události **Click** pro jednu položku nabídky, budete ho moci vytvořit také u všech dalších položek.

1. Poklepejte, neboli dvakrát rychle klepněte, na položku **Exit**. Visual Basic .NET automaticky sestaví programovou konstrukci zpracovatele události **Click** této položky.

2. Do těla zpracovatele události zadejte příkaz **End**, jehož pomocí bude po klepnutí na položku nabídky naše aplikace ukončena. Výslední kód by měl mít tuto podobu:

```
Private Sub miExit_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles miExit.Click

End
End Sub
```

Návrh našeho formuláře je kompletní (obr. 5.7).



Obr. 5.7: Finální návrh formuláře

POŘADÍ, V JAKÉM ZÍSKÁVAJÍ INSTANCE ZAMĚŘENÍ (TAB ORDER)

Pořadí, v jakém získávají instance jednotlivých ovládacích prvků na formuláři zaměření, můžete ve Visual Basicu .NET určit snadněji, než v předchozí verzi tohoto programovacího nástroje. Všechny ovládací prvky obsahují vlastnost **TabIndex**, jejíž pomocí můžete explicitně determinovat postup získávání zaměření. Přestože je nastavování vlastnosti **TabIndex** zcela funkční, integrované prostředí skýtá ještě pohodlnější cestu. Určit pořadí, v jakém získávají instance zaměření totiž můžete provést i vizuálně, aniž byste byli nuceni manuálně zadávat konkrétní hodnoty. Podívejte se na následující postup:

1. Ujistěte se, že se aplikace nachází v režimu návrhu.

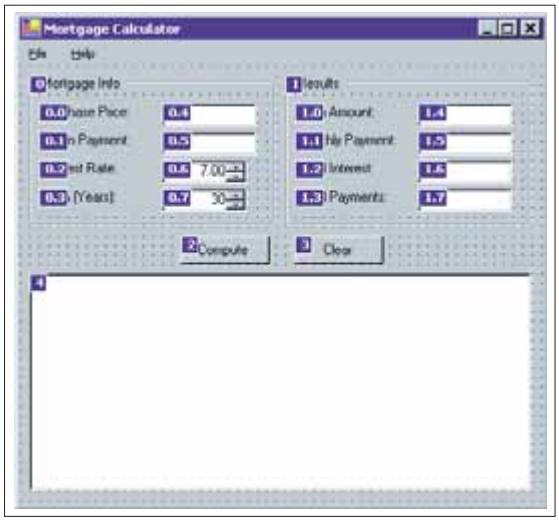
2. Klepněte kdekoliv na plochu formuláře, čímž jej označíte.

3. Otevřete nabídku **View** a klepněte na příkaz **Tab Order**. U všech instancí se objeví popisky s čísly, která představují pořadí, v jakém tyto instance získávají své zaměření.

4. Nyní klepněte na tu instanci, která má získat zaměření jako první. Jakmile klikněte na tuto instanci, změní se její číselní identifikace, přesněji hodnota její vlastnosti **TabIndex**. Poté aktivujte instanci, která získá zaměření ihned po té první. Tento postup opakujte až dokud nebudou ošetřeny všechny instance.

5. Pokud jste s novým nastavením pořadí zaměření instancí spokojeni, znovu vyberte nabídku **View** a aktivujte položku **Tab order**.

Ukázku použití volby **Tab Order** můžete vidět na obr. 5.8.



Obr. 5.8: Nastavování pořadí zaměření instancí ovládacích prvků

Je to opravdu jednoduché, že? Možná, že jednodušší to už ani být nemůže.

Další skutečností, na kterou byste měli pamatovat je, že hodnoty vlastnosti **TabIndex** instancí ovládacích prvků, které jsou uloženy v jistém kontejneru (např. rámečku), jsou nyní aktualizovány vždy, když dojde ke změně indexu zaměření tohoto kontejneru, a to buď přímo nebo nepřímo. Kupříkladu si můžete všimnout, že na obr. 5.8 jsou všechny instance uvnitř rámečků opatřeny indexy zaměření ve vztahu k nadřazeným prvkům, tedy rámečkům. Kdybychom změnili hodnotu vlastnosti **TabIndex** tlačítka **Compute** na 0, nejenom, že by tlačítko získalo zaměření jako první, ale také by se změnil index zaměření rámečku **Mortgage Info** na hodnotu 1. Tato změna by ovlivnila i všechny instance, které jsou součástí rámečku (hodnoty jejich vlastností **TabIndex** by byly 1.x). Samozřejmě, také index zaměření druhého rámečku (**Results**) by byl upraven na hodnotu 2. Společně s tím by byly aktualizovány také indexy zaměření všech instancí, které tento rámeček obsahuje. Tento způsob práce s instancemi ovládacích prvků výrazně zjednodušuje proces jejich rozvržení a ovládání.

Aplikaci můžete uvést do režimu běhu stisknutím klávesy **F5**, nebo klepnutím na tlačítko **Start** na panelu nástrojů **Standard**. Když se aplikace spustí, uvidíte, že formulář vypadá přesně tak, jak jsme jej navrhli. Pořadí získávání zaměření je správné, nabídky pracují (i když jsme vytvořili zpracovatele události jenom pro položku **Exit**) a instance ovládacího prvku **NumericUpDown** splňují kritéria, která budou vyžadovat naši uživatelé. Přestože se zdá, že všechno je v nejlepším pořádků, není tomu úplně tak. Náš formulář má totiž jeden nedostatek, jenž odhalíte jakmile najedete kurzorem myši na jeden z okrajů formuláře. Ano, rozměry formuláře je možné plynule měnit, ovšem my jsme ještě nepřidali žádný kód, který by prováděl korektní změnu pozice instancí jednotlivých ovládacích prvků. Na tuto eventualitu se blíže podíváme v následující části.

MODIFIKACE ROZMĚRŮ

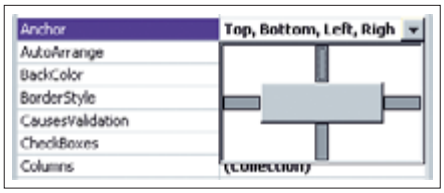
Jste-li ostříleným veteránem ve Visual Basicu, patrně vás již napadlo, že z této zapeklité situace existuje relativně snadná cesta ven. Vždyť přece můžeme změnit styl okrajů formuláře tak, že jeho rozměry nebudou moci být měněny. Ovšem já se ptám, proč bychom se měli takto omezovat? Možná, že mi řeknete, že po letech praxe v programování víte, že napsat kód pro přesné rozmístění a změnu rozměrů instancí ovládacích prvků je velice těžké a vůbec to nestojí za tu námahu. Dobrá, ovšem než vynesete konečný verdikt, zkuste se podívat na několik nových vlastností, kterými současné ovládací prvky ve Visual Basicu .NET disponují.

Všechny ovládací prvky z knihovny **Windows Forms** obsahují novou vlastnost s názvem **Anchor**. Pomocí této vlastnosti můžeme okraje instance ovládacího prvku připevnit k příslušným okrajům kontejneru, v němž je daná instance uložena. Je-li jednou instance připevněna, vzdálenost mezi jejím okrajem a okrajem kontejneru se nemění a zůstává tak konstantní. Implicitně je vlastnost **Anchor** nastavena na hodnotu **Top**, **Left**, což znamená, že všechny instance jsou umísťovány relativně k levému hornímu rohu kontejneru (instance se chovají podobně jako jejich předchůdci ve Visual Basicu 6). Použijeme-li novou vlastnost **Anchor**, můžeme poměrně snadno dosáhnout toho, aby se rozměry instance přizpůsobovaly rozměrům nadřazeného prku. Nyní se podívejme na praktickou aplikaci uvedené techniky.

1. Vraťte se zpět do režimu návrhu aplikace, označte instanci **lvAmortization** a zaměřte svoji pozornost na okno **Properties**.

2. Vyhledejte vlastnost **Anchor** a klepněte na šipku, abyste rozevřeli okno s konfiguračními volbami.

3. Vyberte všechny čtyři kotvící body. Kotvící bod vyberete tak, že jednoduše klepnete na příslušný grafický obdélníkový region. Jakmile je tento region aktivován, změní svoji barvu na tmavě šedou (obr. 5.9).



*Obr. 5.9: Použití vlastnosti **Anchor***

Jak se můžete dovtípit, instance si může zachovat konstantní vzdálenost mezi jednotlivými kotvícími body, i když se tyto změní, jenom tehdy, když sama změní svoji velikost.

Co se týče měnění rozměrů formuláře, chtěl bych vás ještě upozornit na jednu skutečnost. Je sice hezké, že můžeme uživatelům dovolit, aby mohli měnit rozměry formuláře s cílem získat více viditelného prostoru pro zobrazení dat v ovládacím prvku **ListView**, ovšem nesmíme jim poskytnout možnost, aby nechtěně zmenšili formulář natolik, že zmiznou i důležitá data. Jelikož je náš formulář navržen pro velikost 480x400 pixelů, budeme se muset nějak ujistit, že uživatelé nebudou moci nastavit menší rozměry pro šířku a výšku formuláře.

Následující postup ukazuje, jak lze zabránit tomu, aby se velikost formuláře nemohla dostat pod předem stanovený limit.

1. Vyberte formulář a stiskněte klávesu **F4**, čímž zobrazíte okno **Properties**.

2. Vyhledejte vlastnost **MinimumSize** a nastavte ji na hodnotu **480;400**. Jelikož je vlastnost **MinimumSize** složená, můžete také klepnout na symbol plus (+) a jednotlivě upravit hodnotu šířky (**Width**) a výšky (**Height**) formuláře (obr. 5.10).

*Obr. 5.10: Nastavení vlastnosti **MinimumSize***

Pomocí vlastnosti **MinimumSize** formuláře můžeme zajistit, že rozměry formuláře nebudou nikdy menší než 480x400 pixelů. Budou-li uživatelé chtít, můžou velikost formuláře ještě zvětšit, ovšem instance prvku **ListView** se bude této změně plynule přizpůsobovat.

PSANÍ PROGRAMOVÉHO KÓDU

Poté, co jsme si předvedli malou lekci vizuálního programování, se můžeme směle pustit do etapy poslední, kterou je psaní programového kódu. Co si budeme povídat, je to právě kód, jenž je duší naší aplikace.

Zpracovatele událostí **Click** tlačítek (instancí ovládacího prvku **Button**) vytvoříme pouhým poklepáním. V tomto směru je situace stejná jako ve Visual Basicu 6. Kód zpracovatele události **Click** tlačítka **btnClear** je skutečně jednoduchý, takže není zapotřebí jeho význam dlouze vysvětlovat. Všechno, co tento fragment kódu dělá, je to, že všechny instance uvede do výchozího stavu a posléze přesune zaměření na textové pole **txtPurchasePrice**. Programový kód, jenž je nutno přidat do těla zpracovatele, je uveden **tučným** písmem.

```
Private Sub btnClear_Click(ByVal sender As System.Object, _ ByVal e As
System.EventArgs) Handles btnClear.Click

' Reset all of the controls on the form
 txtPurchasePrice.ResetText()
 txtDownPayment.ResetText()
 numInterestRate.ResetText()
 numTerm.ResetText()
 txtLoanAmount.ResetText()
 txtMonthlyPayment.ResetText()
 txtTotalInterest.ResetText()
 txtTotalPayments.ResetText()
 lvAmortization.Items.Clear()
 txtPurchasePrice.Focus()

End Sub
```

Programová struktura zpracovatele události **Click** tlačítka **btnCompute** je poněkud složitější. Je to způsobeno tím, že tento kód realizuje důležité výpočty. Pokud je to možné, můžeme použít i existující zdrojový kód z aplikace našeho zákazníka. Kód jazyka Visual Basic 6 zkopírujeme a vložíme do zpracovatele události **Click** tlačítka ve Visual Basicu .NET. Po uskutečnění několika modifikací bude náš kód vypadat následovně:

```
Private Sub btnCompute_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnCompute.Click

Dim MonthlyPayment As Double
Dim MonthlyInterest As Single
Dim TotalInterest As Double
Dim TotalPayments As Double
Dim Months As Long
Dim InterestRate As Double
Dim Month As Long
Dim PrincipalPaid As Double
Dim InterestPaid As Double
Dim PrincipalBalance As Double
Dim InterestBalance As Double
Dim ListItem As New ListViewItem()

' Make sure the Purchase Price and Down Payment have values
If Len(txtPurchasePrice.Text) = 0 Then txtPurchasePrice.Text = 0
If Len(txtDownPayment.Text) = 0 Then txtDownPayment.Text = 0

' Set local variables based on inputs
Months = numTerm.Value * 12
PrincipalBalance = txtPurchasePrice.Text - txtDownPayment.Text
InterestRate = numInterestRate.Value
```

```
' Calculate the Monthly Interest
MonthlyInterest = CDbl(InterestRate) / 1200

' Calculate computation results
MonthlyPayment = (-MonthlyInterest * PrincipalBalance) / _
((MonthlyInterest + 1) ^ (-Months) - 1)
TotalPayments = MonthlyPayment * Months
TotalInterest = TotalPayments - PrincipalBalance

' Set "Results" fields
txtLoanAmount.Text = Format(PrincipalBalance, "Currency")
txtMonthlyPayment.Text = Format(MonthlyPayment, "Currency")
txtTotalInterest.Text = Format(TotalInterest, "Currency")
txtTotalPayments.Text = Format(TotalPayments, "Currency")

' Clear the ListView first
lvAmortization.Items.Clear()

' Calculate the monthly values and fill the listview
   Do Until PrincipalBalance <= 0
   Month = Month + 1
   InterestPaid = (PrincipalBalance * (1 + InterestRate / 1200)) _
      - PrincipalBalance
   PrincipalPaid = MonthlyPayment - InterestPaid
   InterestBalance = InterestBalance + InterestPaid
   PrincipalBalance = PrincipalBalance - PrincipalPaid

   ListItem = New ListViewItem(Month)
   ListItem.SubItems.Add(Format(MonthlyPayment, "Currency"))
   ListItem.SubItems.Add(Format(InterestPaid, "Currency"))
   ListItem.SubItems.Add(Format(PrincipalPaid, "Currency"))
   ListItem.SubItems.Add(Format(InterestBalance, "Currency"))
   ListItem.SubItems.Add(Format(PrincipalBalance, "Currency"))
   lvAmortization.Items.Add(ListItem)

   Loop
End Sub
```

Je to možná překvapující, ale je to tak. Zpracovatel události **Click** tlačítka ve Visual Basicu .NET pracuje s kódem jazyka Visual Basic 6. Mimochodem, jednou ze syntaktických změn Visual Basicu .NET je nutnost uzavírat argumenty funkcí a procedur Sub do kulatých závorek. Když však vložíte kód Visual Basicu 6 do editoru Visual Basicu .NET, závorky se přidají na příslušná místa zcela automaticky. Je-li to nutné, přidává Visual Basic .NET rovněž kulaté závorky i během psaní programového kódu.

SESTAVENÍ PROJEKTU

A je to! Naše aplikace je kompletní. Poté, co jsme začlenili všechny požadavky, které byly na aplikaci kladeny, můžeme se připravit na sestavení aplikace. Sestavení aplikace je velice snadné. Vše, co musíte udělat, je uvést aplikaci do režimu běhu (klávesa **F5**, nebo tlačítko **Start** na panelu nástrojů **Standard**). Ještě předtím, než se aplikace rozběhne, bude vytvořen její spustitelný soubor (.EXE).

O KROK DÁLE

Naše hypotekární kalkulačka je skutečně triviální aplikací, ovšem na její tvorbě jsme si demonstrovali schéma procesu vývoje aplikace ve Visual Basicu .NET. Je zcela nepopiratelné, že v mnoha skutečných aplikacích je implementace obchodní logiky daleko složitější a komplexnější. Přestože v naší ukázce jsme existující kód přenesli přes systémovou schránku, je možné, že programová implementace vaší obchodní logiky je uložena v COM komponentě. Budete určitě potěšeni, když se dozvíte, že Visual Basic .NET disponuje rozšířenou podporou pro spolupráci s COM komponenty. To mimo jiné znamená, že již vytvořené funkční moduly COM komponent můžete volat přímo i z nejnovější verze Visual Basicu. Platí to i opačně: COM komponenta může také spolupracovat s řízenou aplikací platformy .NET Framework.

Nabídku Visual Studia .NET tvoří rovněž mnoho standardních komponent, jejichž pomocí můžete monitorovat výkonnost systémů, tisknout či zaznamenávat prováděné události. Všechny tyto komponenty můžete přidávat do svých projektů a přímo s nimi pracovat. Více informací o stavbě aplikací založených na komponentech naleznete v dokumentu *RAD Component Creation (Tvorba komponent pomocí systému RAD)*.

ZÁVĚR

Doufám, že vás tato kapitola přesvědčila o tom, že proces budování standardních aplikací pro systém Windows ve Visual Basicu .NET se v zásadě významně neliší od jejich vývoje ve Visual Basicu 6. Je potěšující, že programátoři mohou využít svých stávajících znalostí a velice rychle se adaptovat na nové prostředí Visual Basicu .NET. I nová verze Visual Basicu obsahuje oblíbený systém rychlého vývoje aplikací (RAD), jenž se významně přičinil o to, že se z Visual Basicu stal nejpopulárnější programovací jazyk posledních let. Navíc, s jazykem Visual Basic .NET se můžete nyní dostat až na hranici programátorských možností.

6 Objektově orientované programování v jazyce Visual Basic .NET

Autor: Deborah Kurata, InStep Technologies, Inc.

SHRNUTÍ

Tato kapitola pojednává o technikách objektově orientovaného programování v jazyce Microsoft Visual Basic .NET.

ÚVOD

Předcházející verze programovacího jazyka Microsoft Visual Basic nabízely mechanizmus pro definování datových struktur v takzvaných uživatelsky definovaných datových typech (UDT). Uživatelsky definované datové typy zapouzdřovaly data, ovšem nijak neřešily otázku jejich zpracování. Systém zpracování dat byl definován v globálních standardních modulech, které byly často označovány jako BAS moduly (kvůli stejnojmenné souborové extenzi).

S vydáním Visual Basicu verze 4 se začalo pro vývojáře ve Visual Basicu blýskat na lepší časy. Jazyk Visual Basic udělal své první krůčky směrem k novému programovacímu stylu, jehož jméno bylo objektově orientované programování (zkráceně OOP). Jedním z předpokladů implementace tohoto stylu programování bylo i zařazení modulů tříd, v nichž docházelo k definici datových struktur nejenom pro uchovávání dat, ale i pro jejich zpracování. Zatímco přístup k datům bylo možné získat pomocí vlastností, metody dovolovaly provádět s předmětnými daty operace různého charakteru. Vývojáři získali vývojové prostředí založené na objektech.

Jak šel čas, z Visual Basicu 4 se záhy stal Visual Basic verze 6. Vývojáři se, kromě pokročilejší implementace koncepce OOP, setkali také s vývojovým prostředím, které bylo založené na komponentách (Component-Based Development, CBD). Vývojové prostředí založené na komponentách umožnilo programátorům vytvářet kompletní třívrstvé aplikace pro systém Microsoft Windows a Web. Tento styl vývoje se stal záhy natolik běžným, že společnost Microsoft sestavila modelové schéma, které je známé jako DNA architektura.

Rozdíl v implementaci principů objektově orientovaného programování, jenž můžete pozorovat mezi jazykem Visual Basic 6 a jeho .NET verzí, je skutečně obrovský. Visual Basic .NET přináší skutečné OOP, jak ostatně za chvíli uvidíte.

OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ

Abychom mohli o jakémkoliv programovacím jazyku říct, že je skutečně objektově orientovaný, musí vyhovovat následujícím kritériím:

- **Abstrakce**. Pomocí abstrakce je možné identifikovat skupinu objektů, které tvoří součást řešeného obchodního problému a tyto pak přenést do programového kódu.

- **Zapouzdření**. Princip zapouzdření říká, že objekt skrývá svoji vlastní interní implementaci.

- **Polymorfizmus**. Polymorfizmus umožňuje vytvářet větší počet implementací jedné metody. Kupříkladu dva rozdílné objekty mohou poskytovat metodu **Save**, ovšem konkrétní implementace metody je u každého z nich jiná.

- **Dědičnost**. Nejvíce vzrušení zcela jistě přináší zařazení dědičnosti. Visual Basic 5 uvedl koncepci dědičnosti prostřednictvím rozhraní, což v praxi znamenalo, že jste mohli opětovně používat rozhraní třídy, ovšem ne konkrétní implementaci této třídy. Visual Basic .NET představuje opravdovou implementaci dědičnosti, takže nyní vám již nic nebrání v tom, abyste mohli opětovně používat všechny prvky třídy.

Jak nejlépe naskočit na vlak s nápisem „OOP ve Visual Basicu .NET“, si ukážeme na následujících řádcích.

OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ V JAZYCE VISUAL BASIC .NET

Visual Basic .NET není jenom starý Visual Basic 6 s příchutí skutečné dědičnosti. Spíše můžeme říci, že Visual Basic .NET byl zcela přepracován tak, aby mohl splňovat všechna náročná objektově orientovaná kritéria. Ve skutečnosti takřka vše, s čím přijdete ve Visual Basicu .NET do styku, je objektem. Věřte mi, dokonce i k řetězcům a celým číslům můžete přistupovat jako k objektům.

Abychom si předvedli zmíněné skutečnosti, vytvořte nový projekt konzolové aplikace (**Console Application**). Do těla procedury Sub s názvem **Main** vložte tento fragment zdrojového kódu:

```
Dim i As Integer
MsgBox(i.MinValue)
```

První důležitou informací je, že i s celočíselnou proměnnou můžeme pracovat jako s objektem. To znamená, že můžeme využít vlastnosti a metody, které se objeví, když zadáme znak tečky (.) za názvem proměnné tohoto datového typu. Vyberte jednu z vlastností, například **MinValue**, jak je znázorněno ve výše zmíněné programové ukázce. Když poté spustíte aplikaci, uvidíte dialogové okno s minimální hodnotou, kterou lze do proměnné typu **Integer** uložit.

Dobrá, bázová knihovna tříd platformy .NET Framework obsahuje třídy pro interní datové typy. Ovšem co uživatelsky definované třídy? Samozřejmě, ve Visual Basicu .NET můžete vytvářet i své vlastní třídy. Vytváření a odvozování tříd je nyní velice snadné a zábavné zároveň, o čemž se přesvědčíte při vytváření další programové ukázky.

DEFINICE TŘÍDY

Hlavní účel, pro který vytváříme třídu, se ve Visual Basicu .NET nijak nezměnil. Řečeno jinými slovy, třídy vytváříme proto, abychom mohli používat jejich instance v našich aplikacích. Přitom není rozhodující, zdali se jedná o obchodní objekty, nebo jiné typy požadovaných objektů. Přecházíte-li z jazyka Visual Basic 6, budete se muset obeznámit zejména s novými prvky a syntaktickými modifikacemi.

Tvorba třídy ve Visual Basicu 6 spočívala ve vytvoření modulu třídy podle pravidla jedna třída, jeden modul třídy. Ve Visual Basicu .NET je však situace zcela odlišná. Nyní můžete vytvářet libovolný počet tříd uvnitř jednoho souboru. Dokonce můžete vytvářet i třídy v třídách, tedy vnořené třídy. Do takových dálav se prozatím pouštět nebudeme, protože naše první ukázka bude docela jednoduchá.

Technika přidání třídy do projektu jazyka Visual Basic .NET je velmi podobná té, kterou znáte z prostředí Visual Basicu 6. Nicméně, místo toho, abyste začínali s prázdným souborem, je pro vás připraven deklarační kód třídy:

```
Public Class CCustomer
End Class
```

- **Tip:** Podle současných konvencí společnosti Microsoft není před jméno třídy umísťován prefix v podobě písmene „C“. To znamená, že třída by měla být pojmenována jako **Customer** a ne jako **CCustomer**. I když je tato konvence uživatelsky přívětivější při vytváření objektových hierarchií v softwarových produktech jako je Microsoft Word a Excel, programátoři rozsáhlých podnikových řešení mohou pomocí prefixu získat lepší přehled o charakteru programového kódu.

Pokud budete chtít do stávajícího souboru s kódem třídy přidat kód další třídy, můžete to provést třeba takto:

```
Public Class CCustomer
End Class
Public Class CContact
End Class
```

- **Tip:** Obvykle by třída měla být uložena ve svém vlastním souboru. Více tříd byste měli do jednoho souboru umísťovat pouze tehdy, pokud jsou tyto třídy vzájemně přepojené. Příkladem by mohly být třeba třídy **Faktura** a **PoložkaNaFaktuře**, protože za normálních podmínek byste nikdy nepoužívali třídu **PoložkaNaFaktuře** osamoceně. Na druhé straně, když chcete oddělit kontakty zákazníků od jiných informací o zákaznících, můžete obě příslušné třídy uložit do separátních souborů.

Je-li třída deklarována pomocí příkazu **Class**, můžete do jejího těla přidávat vlastnosti a metody. Podobně jako ve Visual Basicu 6, i nyní můžete definovat vlastnost prostřednictvím deklarování soukromé proměnné a veřejné procedury typu **Property**. Definice vlastnosti **Name** probíhá ve Visual Basicu .NET následovně:

```
Private m_sName As String
Property Name() As String
  Get
     Return m_sName
  End Get
  Set(ByVal Value As String)
     m_sName = Value
  End Set
End Property
```

V těle vlastnosti se nacházejí jenom dvě procedury **Get** a **Set**. Procedura **Get** vrací hodnotu soukromého datového členu třídy, zatímco pomocí procedury **Set** je možné tomuto datovému členu hodnotu přiřadit. Visual Basic 6 nabízel ještě proceduru **Let**, která sloužila pro práci s interními datovými typy (procedura **Set** pracovala místo toho s objekty). Jelikož ve Visual Basicu .NET se setkáváme jenom s objekty, nejsou procedury **Let** nadále podporovány.

Všimněte si, že také syntaxe procedur vlastností doznala jistých změn. Tak především, bloky **Get** a **Set** jsou obsaženy uvnitř příkazů **Property-End Property**, což eliminuje možné problémy při návrhu programové konstrukce vlastnosti.

- **Tip**: Třídy v třívrstvých nebo n-vrstvých aplikacích nemusí disponovat žádnými vlastnostmi. Takovýto model návrhu tříd může být lépe použitelný při budování středných vrstev komponent.

Stavba jednoduché metody je ze syntaktického hlediska takřka shodná s implementací v předcházejících verzích jazyka Visual Basic. Jediným, na první pohled viditelným rozdílem, je příkaz **Return**. Příkaz **Return** můžete nyní používat pro vrácení návratové hodnoty funkce. Není tedy nutné, abyste používali poměrně dlouhý přiřazovací příkaz se jménem funkce nalevo od operátoru přiřazení. Níže uvedený výpis zdrojového kódu představuje jednoduchou metodu:

```
Public Function SayHello() As String
   If Name <> "" Then
      Return "Hello " & Name
   Else
      Return "Hello World"
   End If
End Function
```

V tuto chvíli máme vytvořenou třídu s jednou vlastností a metodou. Jak vidíte, výslední kód se velice podobá na kód Visual Basicu 6.

Na programu je průzkum dalších nových objektově orientovaných prvků jazyka Visual Basic .NET.

VYTVÁŘENÍ KONSTRUKTORŮ A DESTRUKTORŮ

Když jste ve Visual Basicu 6 vytvořili instanci třídy, byla generována událost **Initialize**. Do zpracovatele události **Initialize** jste mohli vložit programový kód, jehož úkolem byla inicializace objektu. Mohli jste třeba provést výchozí nastavení datových členů objektu, navázat spojení s databází či vytvořit jiné příbuzné objekty. Bohužel, zpracovatel události **Initialize** nedokázal pracovat s žádnými parametry, což bylo v mnoha případech poněkud omezující.

Visual Basic .NET uvádí konstruktory, které jsou podrobeny exekuci pokaždé, když dojde k vytvoření nové instance třídy. Tyto konstruktory jsou definovány prostřednictvím procedury Sub s názvem **New**:

```
Public Sub New()
    ' Perform initialization
   Debug.WriteLine("I am alive")
End Sub
```

Dobrou zprávou je, že konstruktoru můžete předávat i požadovaná vstupní data, což je nejenom lepší, ale i efektivnější. Konstruktory, které pracují s parametry, se označují jako parametrické konstruktory. Například:

```
Public Sub New(ByVal sName As String)
  'Assign the name
  Name = sName
  'Other initialization
  Debug.WriteLine(Name & " is alive")
End Sub
```

V této ukázce přijímá parametr konstruktoru jméno zákazníka. Toto jméno je posléze použito k inicializaci datového členu **Name**, ke kterému lze získat přístup pomocí příslušné vlastnosti.

Oba typy konstruktoru (bezparametrický i parametrický) mohou být definovány ve stejné třídě. Ve skutečnosti můžete do jedné třídy umístit tolik konstruktorů, kolik budete chtít, ovšem za předpokladu, že každý z nich bude disponovat jinou sadou parametrů. Tato technika práce s konstruktory je známá jako přetěžování konstruktorů. V závislosti od charakteru vstupních dat bude aktivována jedna z dostupných přetížených variant konstruktoru.
Pokud nechcete, nemusíte konstruktor explicitně definovat. V tomto případě bude vytvořen implicitní, bezparametrický konstruktor.

Visual Basic .NET nepoužívá událost **Terminate**. Na její místo nastoupil destruktor v podobě procedury Sub s názvem **Finalize**. Tento destruktor je volán v okamžiku, kdy automatická správa paměti usoudí, že objekt již není více používán. Mezi zrušením objektu a skutečným odstraněním objektu z operační paměti počítače může být jistý časový interval.

- **Tip**: V běžných situacích byste destruktor **Finalize** neměli vůbec používat, protože, jak bylo zmíněno, existuje mezi zrušením a odstraněním objektu jistá časová lhůta. Navíc, pokud je nutné provést také destruktor, je nutné realizovat také další operace. Místo **Finalize** destruktoru můžete používat raději **Dispose** destruktor.

Abyste dokázali lépe ovládat zdroje alokované instancí vaší třídy, začleňte do třídy rozhraní **IDisposable** a proveďte implementaci metody **Dispose**:

```
Implements IDisposable
Public Sub Dispose() Implements System.IDisposable.Dispose
  ' Perform termination
End Sub
```

Tento destruktor není aktivován automaticky, takže je nutné jej explicitně zavolat. Jak na to se dozvíte v další části tohoto textu.

- **Tip**: **Dispose** destruktor není sice vyžadován, ovšem jeho začlenění je více než doporučováno. Tento typ destruktoru můžete vložit do každé třídy, i do také, která ve skutečnosti nic nedělá. Vývojáři pak mohou volat metodu **Dispose** a provést uklízecí akce.

VYTVÁŘENÍ A LIKVIDOVÁNÍ OBJEKTU

Pomocí třídy můžeme vytvořit objekt, neboli instanci této třídy. Pokud jste chtěli vytvořit instanci třídy v jazyce Visual Basic 6, použili jste následující programový kód:

```
Private m_oCustomer As Ccustomer
Set m_oCustomer = New Ccustomer
```

Podobnou syntaxi můžete upotřebit také ve Visual Basicu .NET. Protože v podstatě vše, s čím přijdete v tomto jazyce do kontaktu, je objekt, již není zapotřebí realizovat dvě odlišné operace přiřazení. To znamená, že klíčové slovo **Set** již nebudete dále potřebovat. Abychom dosáhli rovnocenného výsledku i ve Visual Basicu .NET, použijeme tento fragment zdrojového kódu:

```
Private m_oCustomer As CCustomer
m_oCustomer = New CCustomer()
```

Všimněte si, že ve druhém řádku kódu se za názvem třídy nacházejí závorky. Kdyby třída disponovala parametrickým konstruktorem, mohli byste předávat hodnoty do tohoto konstruktoru takto:

```
m_oCustomer = New CCustomer("Acme Corporation")
```

Jazyk Visual Basic .NET přináší i další novinku: Nyní můžete uskutečnit deklaraci objektové proměnné a vytvoření objektu pomocí klíčového slova **New** v jednom příkazu:

```
Private m_oCustomer As CCustomer = New CCustomer()
```

Jakmile bude tento příkaz vykonán, dojde ke zrození objektu. Jste-li vyznavači rychlých řešení, zcela jistě vás potěší informace, že uvedený příkaz lze zkrátit, asi takto:

```
Private m_oCustomer As New CCustomer()
```

Přejete-li si předávat argument, pozměňte kód tímto způsobem:

```
Private m_oCustomer As New CCustomer("Acme Corporation")
```

- **Tip**: Příkaz, jenž provádí deklaraci odkazové proměnné a vytváří objekt, není možné na úrovni modulu uzavřít do bloku **Try...Catch**. Toto omezení v jisté míře snižuje užitečnost uvedeného stylu vytváření objektů. Proto se můžete uchýlit k osvědčenému schématu: Nejprve deklarujete objektovou proměnnou na úrovni modulu, a pak vytvoříte instanci třídy pomocí klíčového slova **New**. Příkaz pro vytvoření instance může být v tomto případě pod dohledem strukturované správy chyb.

Kód v deklarační části:

```
Private m_oCustomer As CCustomer
```

Kód v proceduře Sub:

```
Try
   m_oCustomer = New CCustomer()
Catch e As Exception
   Debug.WriteLine(e.Message)
End Try
```

Programový kód pro vytvoření objektu přidejte do procedury **Sub Main**. Je přitom jedno, který ze způsobů tvorby objektu si vyberete.

Když objekt již nepotřebujeme, můžeme aktivovat jeho metodu **Dispose** a uvolnit tak zdroje, které byly objektem alokovány (samozřejmě za předpokladu, že objekt tuto metodu implementuje). Poté můžete nastavit objektovou proměnnou na hodnotu **Nothing**, jak je uvedeno níže.

```
m_oCustomer.Dispose()
m_oCustomer = Nothing
```

Když automatická správa paměti zjistí, že objekt již není potřebný, odstraní jej z operační paměti počítače.

- **Tip**: Na rozdíl od Visual Basicu 6, v .NET prostředí není objekt zlikvidován v okamžiku, kdy je objektová proměnná nastavena na hodnotu **Nothing**. Objekt bude podroben destrukci až ve chvíli, určení které je plně v moci automatické správy paměti. Když automatická správa paměti rozhodne, že je zapotřebí objekt zlikvidovat, stane se tak bez ohledu na to, zdali jste objektovou proměnnou nastavili na hodnotu **Nothing** či nikoliv.

V této chvíli byste měli být schopni spustit ukázkovou aplikaci a zobrazit zprávu s informacemi o ladění v okně **Output**.

POUŽITÍ TŘÍDY SYSTEM.OBJECT

Každý objekt, jenž bude zrozen v řízeném prostředí platformy .NET Framework, je odvozen od bázové třídy s názvem **System.Object**. Tato třída je součástí hierarchicky uspořádané knihovny tříd platformy .NET Framework a obsahuje všechny základní vlastnosti a metody, kterým musí .NET objekt vyhovovat.

Veškeré veřejné vlastnosti a metody, které jsou definovány ve třídě **System.Object**, jsou obsaženy ve všech objektech, které vytvoříte. Kupříkladu, třída **System.Object** obsahuje standardní konstruktor. To znamená, že i když váš objekt nemá žádný konstruktor, bude moci být vytvořen, protože třída **System.Object** mu poskytne všechny informace o tom, jak se má vytvořit.

Mnoho veřejných vlastností a metod třídy **System.Object** disponuje předem určenou implementací. To tedy znamená, že k tomu, abyste je mohli použít, nemusíte psát žádný programový kód. Například:

```
m_oCustomer.ToString
```

Metoda **ToString** vrací název komponenty a třídy, která je asociována s instancí **m_oCustomer**. Budete-li chtít, můžete překrýt standardní chování této metody pomocí klíčového slova **Overrides**. Prostřednictvím překrytí můžete definovat svoji vlastní implementaci některých vlastností a metod bázové třídy **System.Object**.

```
Public Overrides Function ToString() As String
   Return Name
End Function
```

Nyní metoda **ToString** vrací místo názvu komponenty a třídy jméno zákazníka.

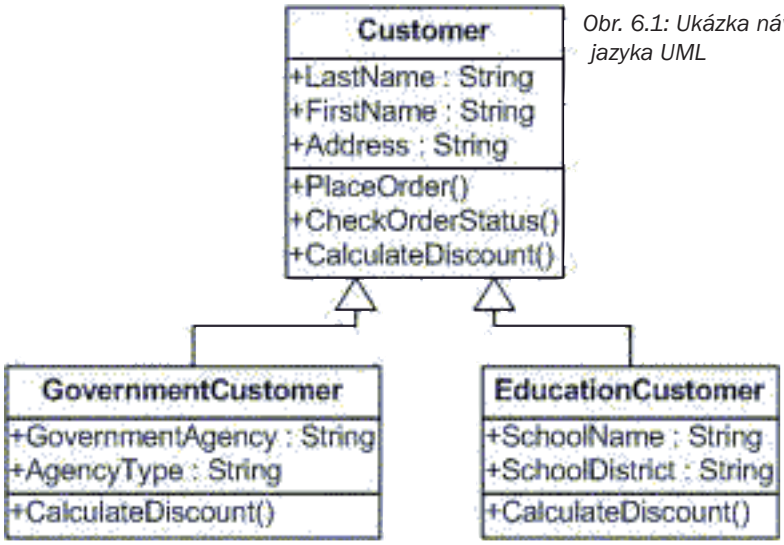
POROZUMĚNÍ DĚDIČNOSTI

Nejdiskutovanějším prvkem objektově orientovaného programování v jazyce Visual Basic .NET je bez jakýchkoliv pochyb dědičnost. Abyste pochopili oč jde, pomůžeme si rozšířenou verzí dříve uvedeného příkladu.

Předpokládejme, že již představená třída **Customer** (Zákazník) je nedílnou součástí kompletního fakturovacího systému. Systém eviduje jednotlivé zákazníky, řídí jejich objednávky a generuje příslušné výstupy v podobě faktur. Takto navržený a sestrojený systém byl odevzdán do rukou uživatelů. Jak se ovšem často stává, po několika týdnech používání systému přijdou uživatelé s dodatečnými požadavky na styl jeho práce. Povězme, že uživatelé nyní žádají, aby systém pracoval s odlišnými skupinami zákazníků podle následujícího schématu:

- Běžní zákazníci budou využívat standardní obchodní podmínky.
- Loajálnější zákazníci budou moci využívat také další obchodní podmínky, některé výhody a speciální skonta.
- Pro nejvěrnější zákazníky budou dostupné ještě další výhody, jako například vyšší slevy při odběru jistého množství tovaru.

Vzájemné vazby mezi všemi třemi skupinami zákazníků můžeme převést do vizuální podoby pomocí jazyka Unified Modeling Language (UML), jak je znázorněno na obr. 6.1. Jak ovšem budete reagovat na nově vzniklé návrhy vašich uživatelů?



Obr. 6.1: Ukázka návrhu systému pomocí modelovacího jazyka UML

Jedním řešením by mohlo být upravení existující třídy **Customer** tak, aby tato třída obsahovala relevantní informace o všech kategoriích zákazníků. Řízení aplikační logiky byste mohli dosáhnout použitím rozhodovacích příkazů **Select Case** či **If**. Takovýto přístup by však mohl být velice únavný a náročný, zejména pokud by požadované úpravy byly širšího charakteru.

Druhým řešením je vytvoření samostatní třídy pro každou skupinu zákazníků. Každá třída by přitom obsahovala jenom ty informace, které jsou vyžadovány konkrétním typem zákazníka. Je však docela možné, že některé informace se budou vyskytovat i ve více třídách, a pak bychom se nevyhnuli vzniku duplicitního programového kódu.

Opatříte-li si Visual Basic .NET, budete mít i třetí možnost: využití dědičnosti. Rozhodnete-li se vsadit na dědičnost, budete mít možnost definovat standardní třídu, která bude sloužit jako základní stavební kámen pro ostatní třídy. Tato třída je nazývána jako bázová třída, nebo také mateřská třída. V naší ukázce byste mohli vytvořit bázovou třídu s názvem **Customer**.

Máte-li sestrojenou bázovou třídu, můžete z ní odvodit specializované třídy. Tyto třídy se označují jako odvozené třídy, nebo také podtřídy. Tyto třídy dědí svoji funkcionalitu z mateřské třídy, ovšem je-li to zapotřebí, mohou překrýt stávající funkcionalitu bázové třídy a místo ní použít své vlastní zpracování.

Abychom si proces překrývání členů třídy představili, použijeme třídu **Customer** jako naši bázovou třídu. Kód v těle třídy ovšem obohatíme o metodu pro výpočet slevy z ceny:

```
Public Overridable Function CalculateDiscount _
(ByVal dAmt As Decimal) As Decimal
' Standard discount is no discount
' Return the passed in amount
Return dAmt
End Function
```

Metoda pracuje s klíčovým slovem **Overridable**. Pomocí tohoto klíčového slova je výslovně naznačeno, že kterákoliv odvozená třída bude moci překrýt tuto metodu a definovat tak svoji vlastní implementaci.

- **Tip**: Jestliže píšete kód třídy, která bude vystupovat jako bázová třída, ujistěte se, že definice všech metod, které budete chtít v budoucnosti překrýt, obsahují klíčové slovo **Overridable**.

Kód odvozené třídy má tuto podobu:

```
Public Class CEdCustomer: Inherits CCustomer
   Public Sub New()
      MyBase.New()
   End Sub
   Public Overrides Function CalculateDiscount _
(ByVal dAmt As Decimal) As Decimal
Dim newAmt As Decimal
      newAmt = dAmt * CDec(0.9)
      Return newAmt
   End Function
End Class
```

Klíčové slovo **Inherits** charakterizuje bázovou třídu. Všechny veřejné vlastnosti a metody bázové třídy jsou přístupné i odvozené třídě.

První metodou v této odvozené třídě je konstruktor. V těle konstruktoru je použito klíčové slovo **MyBase** pro aktivaci konstruktoru bázové třídy. Ačkoliv k volání konstruktoru mateřské třídy z podtřídy dochází automaticky při vytvoření instance odvozené třídy, můžete explicitně specifikovat aktivaci bázového konstruktoru.

Jak si můžete všimnout, v definici druhé metody je obsaženo klíčové slovo **Overrides**. Uvedené klíčové slovo říká, že tato metoda odvozené třídy překrývá svůj protějšek z bázové třídy. Tím pádem je možné, aby metoda odvozené třídy nabízela svou vlastní implementaci.

- **Tip**: Pokud byste neumístili do definice metody odvozené třídy klíčové slovo **Overrides**, Visual Basic .NET by předpokládal, že hodláte zastínit původní metodu. Zastíněná metoda je metoda odvozené třídy, která má stejné jméno jako metoda v bázové třídě, ovšem jejím účelem není tuto metodu překrýt.

Chcete-li vytvořit instanci odvozené třídy, můžete si vybrat z několika cest dalšího postupu:

- Deklarujte objektovou proměnnou, jejímž typem bude odvozená třída a poté vytvořte instanci této třídy:

```
Private m_oEdCustomer As CEdCustomer
m_oEdCustomer = New CEdCustomer()
```

- Deklarujte objektovou proměnnou, jejíž typem bude bázová třída a poté vytvořte instanci odvozené třídy:

```
Private m_oCustomer As CCustomer
m_oCustomer = New CEdCustomer()
```

Prostřednictvím první techniky deklarujeme objekt specifického typu, tedy objekt typu **CEdCustomer**. Odvozený objekt má přístup ke všem veřejným vlastnostem a metodám bázové třídy **Customer**, dále ke všem vlastnostem a metodám, které byly ve třídě **CEdCustomer** překryté nebo zastíněné. Konečně, tento objekt má rovněž přístup i ke všem veřejným vlastnostem a metodám třídy **EdCustomer**.

Ve druhé alternativě je deklarován objekt třídy **CCustomer**, což je technika, která připomíná polymorfní chování. Objektovou proměnnou je totiž možné opětovně použít pro jakýkoliv typ objektu třídy **Customer**. Nicméně, pokud budete deklarovat odvozený objekt pomocí bázové třídy, měli byste pamatovat na to, že tento objekt bude mít přístup k veřejným vlastnostem a metodám bázové třídy **Customer** a také k jakýmkoliv vlastnostem a metodám, které byly ve třídě **EdCustomer** překryty. Objekt proto nebude mít přístup k veřejným vlastnostem a metodám třídy **EdCustomer**, ani k zastíněným vlastnostem či metodám.

Při práci s dědičností existuje ještě několik málo zásad, které by neměly uniknout vaší pozornosti. Za prvé, věžte, že nejste limitováni pouze jednou úrovní dědičnosti. Hierarchie dědičnosti může být tak hluboká, jak jen budete potřebovat. Příslušné vlastnosti a metody budou v procesu dědičnosti přenášeny ze třídy, která se nachází na vyšším stupni hierarchie na třídu, která je umístěna na nižším hierarchickém stupni. Ve všeobecnosti platí pravidlo, podle kterého se specializace třídy zvyšuje tím více, čím níže je třída uložena v hierarchii dědičnosti. Kupříkladu byste mohli definovat třídu **HighSchoolEdCustomer**, která by byla odvozena od třídy **EdCustomer**, přičemž třída **EdCustomer** by byla podtřídou třídy **Customer**.

- **Tip**: Abyste minimalizovali složitost návrhu a správu hierarchie tříd, snažte se hierarchii dědičnosti tříd omezit maximálně na čtyři úrovně.

Skutečností, kterou byste měli neustále mít na paměti je, že každá třída může mít jenom jednoho přímého předka. To znamená, že třída **EdCustomer** nemůže dědit své charakteristiky od obou tříd (**Customer** a **Education**) současně. Této technice se říká jednoduchá dědičnost. Kromě ní existuje i vícenásobná dědičnost, ovšem ta není na platformě .NET Framework podporována. Je to však ke prospěchu věci: Vícenásobná dědičnost vedla mnohokrát k příliš přepjatému návrhu a daleko namáhavější údržbě systému tříd.

Koncepci dědičnosti tříd můžete použít zejména v následujících oblastech:

- **Můžete vytvářet objekty odlišných typů, které disponují podobnou funkcionalitou.** Například: Třídy **Educational Customer** a **Government Customer** jsou podtřídami třídy **Customer**.

- **Kolekcím objektů můžete přiřadit standardní chování.** Například: Jakýkoliv typ obchodního objektu je odvozen ze třídy **Business Object** (**BO**).

Dědičnost byste neměli používat, když:

- **Z bázové třídy potřebujete zavolat pouze jednu metodu.** V tomto případě byste měli tímto úkolem pověřit mateřskou třídu a ne odvozovat další třídu.

- **Budete potřebovat překrýt všechny metody bázové třídy.** V této situaci by bylo lepší, kdybyste využili možností rozhraní místo implementace dědičnosti.

- **Sémantika hierarchie není zřetelná.** Pakliže neexistuje jasná definice vztahu „je“, jako třeba: Loajální zákazník „je“ zákazník, potom je začlenění rozhraní vskutku lepším řešením. Uvažujme o tomto příkladu: Prodejce (vendor) disponuje svým jménem stejně jako zákazník (customer). Tudíž, aby třída **Vendor** mohla pracovat se jménem, mohla by být odvozena od třídy **Customer**. Tento vztah ovšem není zřejmý, protože nemůžeme říci, že prodejce „je“ zákazník. Sémantika není v tomto případě jasná, což znamená, že zde není vhodné implementovat dědičnost.

ZÁVĚR

Již dávno za námi jsou dny, když programování v jazyce Visual Basic znamenalo pouhé kreslené formulářů a poklepávání na několik tlačítek pro vytvoření kompletní aplikace. Již dávno za námi jsou dny, kdy se komunita programátorů dívala na vývojáře ve Visual Basicu jako na ty, kteří by se teprve rádi stali skutečnými programátory.

Vývojáři pracující ve Visual Basicu jsou nyní skutečnými profesionály svého řemesla. Jazyk Visual Basic .NET nám poskytl novou sadu profesionálních vývojových nástrojů, které jsou plně srovnatelné se softwarovou výbavou jiných profesionálních programátorů. Nyní nám již nic nebrání v tom, abychom se pustili do bujarých vod programování s dědičností a polymorfizmem. Prostřednictvím nových jazykových a syntaktických rozšíření můžete i nadále zůstat nejefektivnějšími programátory v tom nejvíce produktivním programovacím jazyce, jaký byl kdy stvořen.

7 Jak na ladění aplikací v jazyce Visual Basic .NET

Microsoft Corporation

Vztahuje se na:
Microsoft Visual Studio .NET
Microsoft Visual Basic .NET

SHRNUTÍ

V této části se naučíte, jaké nové postupy můžete provádět při odlaďování vašich aplikací. Přesněji řečeno, dozvíte se, jak používat ladící dialogová okna, jak uskutečňovat změny při ladění, či jak pracovat s dialogovými okny **Command** a **Output**. Visual Studio .NET přináší zcela přepracovaný model, podle něhož vývojáři pracují a ladí své aplikace. Mnohé z těchto změn budou zajímavé zejména pro vývojáře, kteří do prostředí vývojové platformy .NET Framework přicházejí z nižší verze jazyka Visual Basic.

Cíle

- Vytvořit a odladit jednoduchou standardní aplikaci pro operační systém Windows pomocí oken **Debug** a **Output**.
- Předvést rozličné použití okna **Command**.
- Obohatit zdrojový kód testovací aplikace o zarážky a záložky a ukázat, jak mohou být tyto nástroje užitečné.
- Poukázat na třídy **Debug** a **Trace** a jejich důmyslné metody.

ÚVOD

Při programování svých aplikací všichni používáme různé triky a tajné postupy, abychom jich zbavili co největšího množství chyb. Přestože prostředí programovacího jazyka Visual Basic 6 zahrnuje mnoho užitečných nástrojů, existovalo něco, co nám chybělo. Tyto přetrvávající potřeby uspokojil až jazyk Visual Basic .NET přidáním inovativních a brilantně provedených programových prvků, s jejichž pomocí se můžeme pustit do boje s počítačovými chybami.

Zatímco dovednosti jednoduchého, ovšem často používaného dialogového okna **Command** byly citelně rozšířeny, zarážky a záložky přinesly nový standard, který známe z prostředí Internetu. Již tak bohaté vývojové prostředí ještě dále obohacují třídy **Debug** a **Trace**. Všechna inovativní řešení mají společný cíl: Umožnit vývojářům ve Visual Basicu .NET snadnější ladění jejich počítačových aplikací. Ovšem nejenom to! Začleněním ladících nástrojů získává nyní vývoj aplikací ve Visual Basicu .NET zcela nový rozměr.

VYTVÁŘENÍ KALKULAČKY ŽIVOTNÍHO POJIŠTĚNÍ

Abychom si mohli předvést ladění aplikací v praxi, musíme mít po ruce nějakou aplikaci, kterou bychom mohli podrobit ladění. Proto nejprve vytvoříme jednoduchou aplikaci, která bude vypočítávat životní pojistku, jejíž výše bude odvozena od výše platu zaokrouhleného na nejbližší tisícovku. Budeme používat jednoduchý formulář s instancemi ovládacích prvků, jejichž pomocí získáme vstupní hodnoty pro vytvářené proměnné. Na formuláři se bude rovněž nacházet i tlačítko, po jehož stisknutí budou vypočteny výsledky. Poté aplikaci spustíme a prostřednictvím oken **Debug** a **Output** budeme sledovat provádění korekcí a modifikací programu.

Spusťte Microsoft Visual Studio .NET, na stránce **Start Page** klepněte na tlačítko **New Project**, nebo otevřete nabídku **File**, ukažte na položku **New** a vyberte příkaz **Project**. Jako typ projektu vyberte **Visual Basic Projects** a jako projektovou šablonu zvolte **Windows Application**. Aplikaci pojmenujte jako **LifeInsCalc**. Chcete-li, můžete vybrat vhodnou složku, do níž budou uloženy soubory aplikačního projektu. Po provedení všech změn nezapomeňte aktivovat tlačítko OK.

Nyní byste se měli ocitnout v režimu návrhu aplikace, přičemž implicitně by mělo být zobrazené okno s formulářem **Form1**. Hodnotu vlastnosti **Text** formuláře změňte na **Life Insurance Cost Calculator**. Dále se řiďte následujícími instrukcemi:

1. Na formulář přetáhněte pět instancí ovládacího prvku **Label** a uložte je v levé části formuláře.

2. Hodnoty vlastností **Text** těchto instancí upravte takto:

Label1	**Text** = Employee´s Annual Salary:
Label2	**Text** = Rounded Salary:
Label3	**Text** = Ins. Cost per $1000:
Label4	**Text** = Total Cost of Insurance:
Label5	**Text** = Bi-weekly Cost of Insurance:

3. Pokračujte tím, že na formulář přidáte 4 instance ovládacího prvku **TextBox**. Tyto instance umístěte napravo od popisků **Employee´s Annual Salary**:, **Rounded Salary**: **Total Cost of Insurance** a **Bi-weekly Cost of Insurance**. Vlastnosti nově přidaných instancí modifikujte následujícím způsobem:

TextBox1	**Name** = txtSalary	**Text** = 0.0	
TextBox2	**Name** = txtRoundSalary	**Text** = “”	**ReadOnly** = True
TextBox3	**Name** = txtTotalCost	**Text** = “”	**ReadOnly** = True
TextBox4	**Name** = txtBiWeeklyCost	**Text** = “”	**ReadOnly** = True

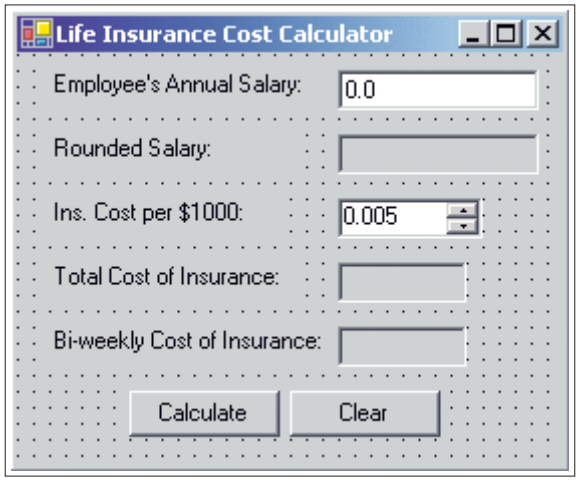
4. Jednu instanci ovládacího prvku **NumericUpDown** umístěte vedle popisku **„Ins. Cost per $1000:“**. Instanci přejmenujte na **numUnitCost**. Kromě toho upravte i její další vlastnosti:

DecimalPlaces	3
Increment	.001
Minimum	0
Value	.005

5. Na formulář přidejte dvě tlačítka (instance ovládacího prvku **Button**), která umístěte vedle sebe pod textová pole (instance ovládacího prvku **TextBox**). Vlastnosti **Name** a **Text** tlačítek pozměňte dle níže uvedeného vzoru:

Button1	**Name** = btnCalculate	**Text** = Calculate
Button2	**Name** = btnClear	**Text** = Clear

Je-li to třeba, upravte také velikost formuláře tak, aby se jeho vzhled shodoval s formulářem, jenž je zobrazen na obr. 7.1.



Obr. 7.1: Formulář v režimu návrhu

Když jsme úspěšně navrhli základní vizuální rozvržení formuláře, můžeme se soustředit na psaní programového kódu. Poklepejte na tlačítko **btnClear**, nacož Visual Basic .NET vygeneruje programovou kostru zpracovatele události **Click** tohoto tlačítka. Do těla zpracovatele události uložíme kód, jenž bude odpovědný za nastavení všech instancí do výchozího stavu a za přesunutí zaměření na textové pole **txtSalary**. Zde je vzpomínaný výpis kódu:

```
Private Sub btnClear_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnClear.Click

      ' Clear the controls on the form
      txtSalary.ResetText()
      txtRoundSalary.ResetText()
      txtTotalCost.ResetText()
      txtBiWeeklyCost.ResetText()
      numUnitCost.ResetText()
      txtSalary.Focus()

End Sub
```

Kód zpracovatele události **Click** tlačítka **btnCalculate** bude pochopitelně mnohem složitější, protože bude provádět veškerou požadovanou činnost. Nejprve bude kód analyzovat částku zapsanou v textovém poli **Salary**, a poté ji zaokrouhlí na nejbližší tisícovku. Výsledek poté vynásobí jednotkovými náklady pojištění. Produktem této kalkulace bude celková částka, která se bude objevovat v textovém poli **Total Cost**.

Poklepejte na tlačítko **btnCalculate** a do zpracovatele události **Click** vložte následující fragment zdrojového kódu:

```
Private Sub btnCalculate_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnCalculate.Click

      ' The local calculation variables
      Dim EmpSalary As Double
      Dim RoundSalary As Single
      Dim UnitCost As Double
      Dim TotalCost As Single
      Dim BiWeeklyCost As Single

      ' Set local variables based on inputs
      EmpSalary = CDbl(txtSalary.Text)
      UnitCost = CDbl(numUnitCost.Value)
      ' Round the Salary up to the nearest thousand
      RoundSalary = Int(EmpSalary)

      ' Calculate the Total Insurance Cost
      TotalCost = UnitCost * RoundSalary
      BiWeeklyCost = TotalCost / 26

      ' Update the Rounded Salary and Insurance Cost Textboxes
      txtRoundSalary.Text = RoundSalary
      txtTotalCost.Text = TotalCost
      txtBiWeeklyCost.Text = BiWeeklyCost

End Sub
```

V tuto chvíli je naše aplikace připravena k otestování. Abychom aplikaci uvedli do režimu běhu, můžeme klepnout na tlačítko **Start** na panelu nástrojů **Standard**, nebo můžeme jednoduše stisknout klávesu **F5**. Vyzkoušejte kalkulačku, přičemž použijte i poněkud netradiční částky pro výši platu, jako třeba 21 189,99.

Při běhu aplikace si můžete všimnout několik zjevných problémů. Textové pole **Rounded Salary** nepracuje správně, protože neprovádí zaokrouhlování na nejbližší tisícovku. Musíme se tedy pokusit tuto chybu odladit. Podívejme se, jak nám při ladění může Visual Basic .NET pomoci.

NABÍDKA DEBUG

Nabídka **Debug** je přirozeným vstupním bodem do světa ladění počítačových aplikací. Jakmile tuto nabídku otevřete, zpozorujete, že jsou zde přítomny volby, které se ve Visual Basicu 6 vyskytovaly uvnitř menu **Run**. Jsou zastoupeny všechny „staré známé“ příkazy jako třeba **Start**, **Restart**, **Break** a další. Některé příkazy však byly přejmenovány. Kupříkladu příkaz **Start with Full Compile** se nyní jmenuje **Start without Debugging**. Začněte-li jednou s laděním, zcela jistě postřehnete, že příkaz **End** má nyní svého dvojníka s názvem **Stop Debugging**.

Některé příkazy, které v prostředí jazyka Visual Basic 6 obsazovaly nabídku **View**, jsou nyní dostupné z nabídky **Debug**. Mezi těmito příkazy můžete najít **Immediate Window**, **Call Stack**, **Watch** či **Locals Window**.

Pamatujte na to, že některé volby nabídky **Debug** nemusí být v režimu návrhu viditelné. Tyto se objeví až ve fázi ladění aplikace v režimu běhu.

ROBUSTNÍ OKNO COMMAND

Okno **Command** je možné použít pro zasílání příkazů, nebo pro ladění a vypočítávání výrazů v integrovaném vývojovém prostředí (IDE). Okno **Command** může vystupovat ve dvou módech: **Command** a **Immediate**.

Mód **Command** je využíván pro přímé spouštění příkazů uvnitř prostředí Visual Studia .NET, dále pro nepřímou exekuci příkazů nabídek, a také pro aktivaci příkazů, které se v nabídkách vůbec nenacházejí. Budete-li chtít otevřít okno **Command**, otevřete menu View, ukažte na položku **Other Windows** a poté klepněte na příkaz **Command Window**. Zdá-li se vám uvedený postup poněkud zdlouhavý, lépe vám poslouží klávesová zkratka **CTRL+ALT+A**.

Ukažme si, jak můžete použít okno **Command** pro zobrazení dialogového okna **New File**. Za běžných okolností byste nejspíše postupovali přes nabídku **File**, ovšem stejného cíle dosáhnete i když zadáte do okna **Command** tento příkaz:

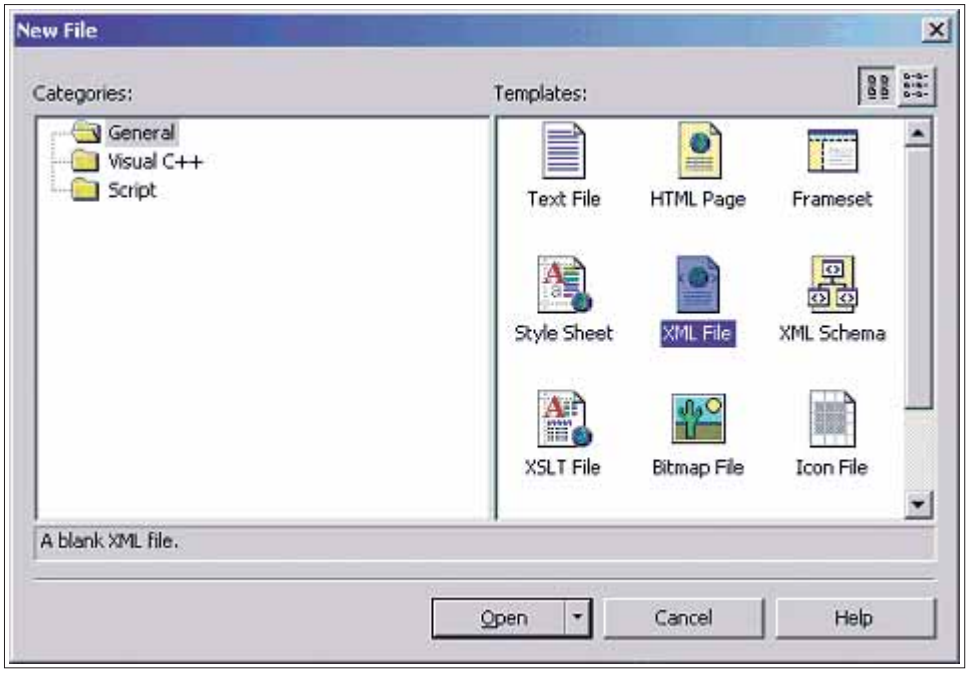
```
>File.NewFile
```

Velice užitečné je to, že jakmile zapíšete první písmeno, technologie IntelliSense® zobrazí plovoucí okno, jehož pomocí můžete příkaz vybrat, a tudíž i rychleji použít. Pokud se ptáte, proč raději nepoužít myš, na následujících řádcích naleznete odpověď.

Během zadávání příkazů v módu **Command** je zcela nejlepší to, že můžete příslušné příkazy opatřit i argumenty. Zvážíte-li tuto možnost, rychle pochopíte, že se vám tak otevírá zcela nová dimenze použitelnosti. Pokračujme však v předchozí ukázce a povězme, že budeme chtít, aby byl rovnou vytvořen nový XML soubor. Můžeme použít tento příkaz:

```
>File.NewFile lifeins /t:"General\XML File"
```

Uvedený příkaz vytváří nový XML soubor s názvem **lifeins.xml**, jenž je založen na projektové šabloně XML File. Argument **/t:templatename** pracuje stejně, jako kdybyste použili položky nabídky, přičemž jméno kategorie je od jména šablony oddělené zpětným lomítkem (\).



Obr. 7.2: Šablona XML souboru

Jak uvidíte za chvíli, mód **Immediate** se používá spíše pro ladící účely. V tomto módu lze vyhodnocovat výrazy, spouštět příkazy, zobrazovat hodnoty proměnných a mnoho dalšího.

OKNO COMMAND: MÓD IMMEDIATE

Mód **Immediate** okna **Command** vám dovoluje zadávat výrazy, spouštět příkazy, získávat hodnoty proměnných a měnit tyto hodnoty (v některých případech). Tyto aktivity je možné provádět během ladění aplikaci, což využijeme za chvíli, kdy budeme používat zarážky a realizovat změny v ukázkové aplikaci.

Abyste mohli aktivovat mód **Immediate** okna **Command**, vyberte nabídku **Debug**, ukažte na položku **Windows** a klepněte na příkaz **Immediate**. Alternativně můžete použít i klávesovou zkratku **CTRL+ALT+I**. Když se okno **Command** přepne do **Immediate** módu, text v titulkovém pruhu okna se změní na **Command Window – Immediate**.

Když vývojáři IDE pracovali na módu **Immediate**, zcela jistě chtěli, aby vaše práce v tomto módu byla tak rychlá a efektivní, jak jen to jde. Proto se i v módu **Immediate** můžete setkat s technologii IntelliSense.

Nacházíte-li se v módu **Immediate**, kurzorové klávesy šipka nahoru a šipka dolů již neslouží pro pohyb mezi předcházejícími příkazy, ale místo toho vám nyní dovolují rolovat všemi zadanými příkazy. Můžete tedy poměrně snadno kopírovat předcházející příkazy, nebo jejich části tak, že je vyberete rolováním a upravíte podle potřeby. Pro spuštění příkazu stiskněte klávesu ENTER.

Pokud chcete zadávat příkazy a jste právě v **Immediate** módu, není nutné, abyste otevírali okno **Command**. Stačí, když před každý příkaz uvedete symbol větší než (>). Kupříkladu, přepnutí módu **Immediate** na mód **Command** můžete provést pomocí tohoto příkazu:

```
>cmd
```

A naopak, pro opětovnou cestu zpět do **Immediate** módu použijte následující příkaz:

```
>immed
```

Abychom si ukázali práci v **Immediate** módu, spustíme naši testovací aplikaci, a poté vybereme z nabídky **Debug** příkaz **Break All**. Tím pozastavíme exekuci programových instrukcí programu, takže nyní můžeme testovat několik výrazů v okně **Immediate Command**.

Aplikujte klávesovou zkratku **CTRL+G**, abyste ověřili, zdali je okno **Immediate Command** otevřeno. Nyní spusťte program klepnutím na tlačítko **Start**, nebo stisknutím klávesy **F5**. Dále pozastavte běh programu klepnutím na tlačítko **Break All** na panelu nástrojů **Debug**, nebo vyberte stejnojmennou položku z nabídky **Debug**. Okno **Immediate Command** by se v této chvíli mělo zobrazit pod oknem editoru pro zápis kódu.

Nyní se pokusíme obnovit běh pozastavené aplikace pomocí příkazu. Jelikož jsme doposud používali mód **Immediate**, nesmíme zapomenout umístit před příkaz symbol větší než (>). IntelliSense vám pomůže, ovšem s identifikací většiny příkazů nebudete mít potíže, protože odrážejí schéma příkazů v systému nabídek. Zadejte následující příkaz:

```
>Debug.Restart
```

Pokud bychom chtěli, stejným způsobem bychom mohli i zastavit provádění kódu aplikace:

```
>Debug.BreakAll
```

Zcela jistě se shodneme na tom, že okno **Command** skýtá zcela nový pohled na proces ovládání integrovaného vývojového prostředí v jazyce Visual Basic .NET.
Níže uvedený příkaz zastaví běh programu a provede návrat do režimu návrhu aplikace:

```
>Debug.StopDebugging
```

Mód **Immediate** budeme dále testovat až za chvíli. Nejprve si však představíme okna **Output** a **Breakpoints**.

OKNO OUTPUT

Toto okno zobrazuje stavové zprávy, které se vážou k rozličným prvkům integrovaného vývojového prostředí. Okno je tvořeno několika panely, ke kterým lze přistupovat prostřednictvím rozevíracího seznamu, jenž je umístěn těsně pod titulkovým pruhem okna. Dostupné panely umožňují zobrazovat zprávy o procesu sestavení projektu či ladící zprávy. Rovněž mohou být nakonfigurovány tak, aby prováděly výstup z .bat nebo .com souborů, které by se normálně zobrazovaly v příkazovém řádku MS-DOS.

Otevření okna **Output** je jednoduché: Vyberte nabídku **View**, ukažte na položku **Other Windows** a klepněte na položku **Output**. Jste-li příznivci klávesových zkratek, můžete použít formulku **CTRL+ALT+O**.

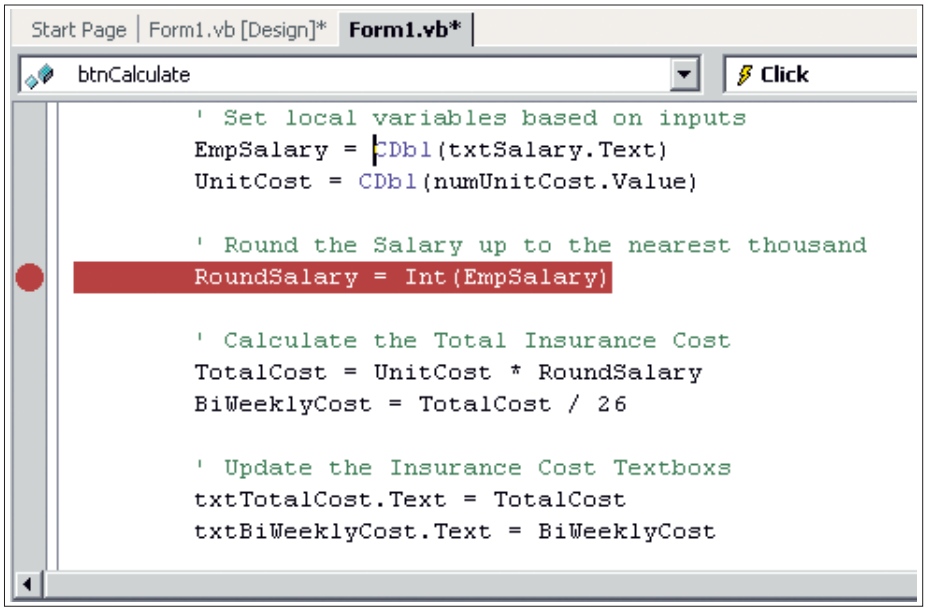
Ještě jednou spusťte ukázkovou aplikaci (Kalkulačka životního pojištění). Přitom si všimněte zpráv, které se budou v tuto chvíli zobrazovat v okně **Output**. Jakmile se bude aplikace inicializovat, objeví se panel **Build** se zprávami, které se vážou k sestavení a startu aplikace. Společně se zobrazením hlavního formuláře kalkulačky se v okně **Output** objeví panel **Debug**.

Není-li okno **Output** z nějakého důvodu viditelné, aktivujte klávesovou zkratku **CTRL+ALT+O**, nebo vyhledejte záložku tohoto okna.

Jak vidíte, panel **Debug** okna **Output** zobrazuje informace související s během aktuální počítačové aplikace.

ZARÁŽKY

Zarážky pořád představují výjimečný způsob zastavení běhu programu na jistých, předem definovaných bodech. Aplikace zarážek není nijak složitá. Jednoduše stačí, když vyhledáte požadovaný fragment zdrojového kódu a klepnete na šedou plochu levého panelu okna editoru pro zápis kódu. Ukázku tohoto postupu znázorňuje obr. 7.3.



Obr. 7.3: Přidávání zarážky do programového kódu

Jakmile provedete aplikaci zarážky, v levém panelu se objeví indikátor zarážky, jenž má podobu červeného kruhu. Pokud klepnete na tento indikátor pravým tlačítkem myši, zobrazí se kontextová nabídka s dalšími volbami.

Předtím, než budeme pokračovat, bychom se měli přistavit u techniky, které se říká **Outlining**. Pomocí této techniky lze organizovat programový kód do samostatných, lépe čitelných a řiditelných jednotek. Budete-li chtít vytvořit takovouto jednotku kódu postupujte následovně:

1. Vyhledejte předmětný zdrojový kód a vyberte jej do bloku.

2. Na vybraný blok klepněte pravým tlačítkem myši, nacož se objeví kontextová nabídka.

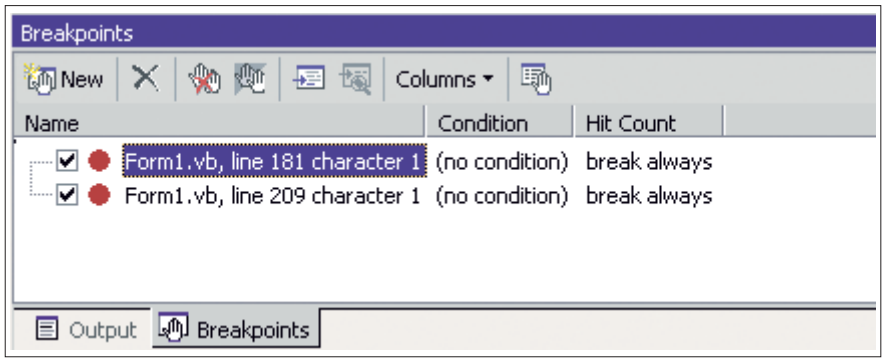
3. Ukažte na položku **Outlining** a vyberte příkaz **Hide Selection**.

Příkaz **Hide Selection** provede svinutí bloku kódu do jednoho regionu, před kterým bude umístěn indikátor (+ ...). Tento indikátor můžete použít pro opětovné rozvinutí skrytého bloku programového kódu. Skrytý kód můžete však i rychle prohlížet tak, že najedete kurzorem myši na již zmíněný indikátor (obr. 7.4).

Obr. 7.4: Rychlé prohlížení skrytého programového kódu

Dialogové okno **Breakpoints** je praktické zejména v tom smyslu, že vám poskytuje přehled o všech označených zarážkách. Kromě toho můžete také upravovat nastavení, která jsou asociována s každou z dostupných zarážek. Otevření okna **Breakpoints** zabezpečíte pomocí klávesové zkratky **CTRL+ALT+B**. Pro úplnost dodejme, že můžete rovněž vybrat nabídku **Debug**, ukázat na položku **Windows** a klepnout na položku **Breakpoints**.

Okno **Breakpoints** obsahuje panel s tlačítky a seznam dostupných zarážek (obr. 7.5).



Obr. 7.5: Okno Breakpoints

OPRAVUJEME APLIKACI KALKULAČKA ŽIVOTNÍHO POJIŠTĚNÍ

Dobrá, nyní můžeme umístit zarážku do zpracovatele události **Click** tlačítka **btnCalculate** v místě, kde dochází k přiřazení hodnoty do proměnné **RoundSalary**. Takto můžeme použít mód **Immediate** okna **Command** pro uskutečnění testů s několika výrazy. Uvidíme, která varianta bude pro rozřešení našeho problémů optimální. Poklepejte na tlačítko **Calculate** a vyhledejte následující fragment zdrojového kódu:

```
' Round the Salary up to the nearest thousand
RoundSalary = Int(EmpSalary)
```

Na druhý řádek aplikujte zarážku. Poté se ujistěte, že okno **Immediate Command** je otevřené (**CTRL+G**) a spusťte aplikaci za účelem testování a ladění.

Když se zobrazí formulář aplikace, zadejte do prvního textového pole hodnotu 31121,98. Klikněte na tlačítko **Calculate** a všimněte si, že v okamžiku, kdy byla dosažena zarážka, objeví se okno editoru pro zápis kódu společně s oknem **Immediate Command**.

Najedete-li kurzorem myši na kteroukoliv proměnnou, v bublinovém okně se zobrazí aktuální hodnota této proměnné. Hodnota proměnné **RoundSalary** by měla být rovna 0.0, protože tento řádek kódu ještě nebyl zpracován. V okně **Immediate Command** můžeme vyhodnotit několik výrazů a provést napravení programové chyby.

Svět pojištění je úžasný svým vlastním matematickým stylem. V tomto případě, bez ohledu na zadanou hodnotu, musí být tato upravena na další přírůstek hodnoty 1000. Bude-li výše platu 20000,99, musí být zaokrouhlena na 21000,00.

Do textového pole okna **Immediate Command** zadejte přiřazovací příkaz **RoundSalary** = **Int(EmpSalary)** a stiskněte klávesu ENTER. Ukážete-li nyní kurzorem myši na proměnnou **RoundSalary**, uvidíte, že hodnota proměnné je 31121,0. Hodnotu přiřazenou do proměnné **RoundSalary** můžete zjistit, když do okna **Immediate Window** zadáte příkaz **?RoundSalary**.

Nyní můžeme začít s realizací všech typů testů, a tak zjišťovat, pomocí kterého výrazu dosáhneme kýžený výsledek:

```
?int((EmpSalary + 1000)/1000) * 1000
```

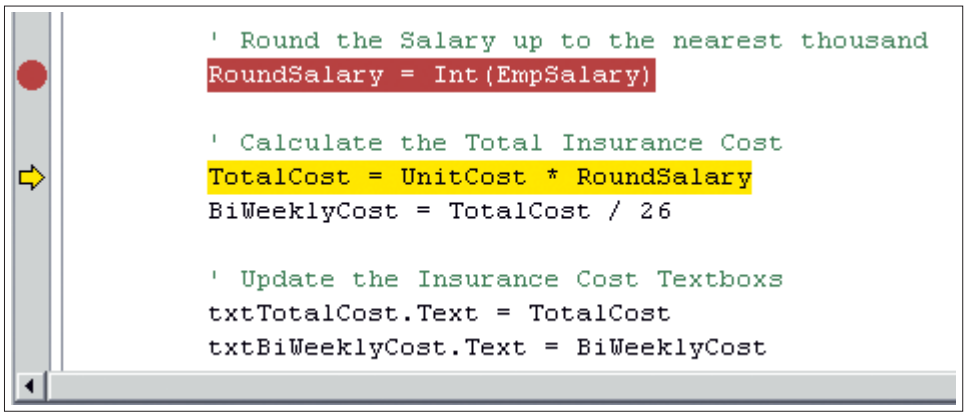
Výslední hodnota je v tomto případě 32000, což sice vypadá celkem slibně, ovšem když nahradíme proměnnou **EmpSalary** různými potenciálními částkami, budeme moci odkrýt skrytá problémová místa:

```
?int((20000 + 1000)/1000) * 1000
```

Nyní je výsledek 21000, jenž není správný ani podle pojišťovací matematiky. Naštěstí existuje metoda **Ceiling** z jmenného prostoru **Math**, která vrací nejmenší celé číslo, které je větší nebo rovno specifikované číselní hodnotě. Použijeme-li tuto metodu podle níže uvedeného vzoru, dospějeme k žádanému výsledku:

```
RoundSalary = Math.Ceiling(EmpSalary/1000)*1000
```

A je to! Bez ohledu na zadanou hodnotu bude vypočten správně zaokrouhlený výsledek. Když jsme provedli nastavení proměnné **RoundSalary**, můžeme přeskočit vykonání přiřazovacího příkazu v kódu a místo toho přesunout exekuci na místo, které si určíme. Umístěte kurzor myši na zarážku a táhněte žlutou šipku směrem dolů až dokud nedosáhnete řádku, v němž dochází k přiřazení hodnoty do proměnné **TotalCost** (obr. 7.6).



Obr. 7.6: Určení místa exekuce

Žlutá šipka ukazuje na řádek s programovým kódem, jenž bude proveden jako první v okamžiku, kdy bude obnoven běh spuštěné aplikace. Vyberte nabídku **Debug** a klepněte na položku **Continue**, nebo klepněte na tlačítko **Start** na panelu nástrojů **Debug**. Po obnovení běhu aplikace si můžete všimnout správných výpočtů v jednotlivých textových polích.

Je jisté, že budete chtít upravit také řádek kódu, v němž dochází k přiřazení hodnoty do proměnné **RoundSalary**, ovšem zde je nutné mít na paměti změnu oproti jazyku Visual Basic 6. Ve Visual Basicu 6 jste mohli ve většině případů provádět modifikace kódu během okamžiků, kdy byl běh spuštěné aplikace pozastaven. Po vykonání změn jste mohli běh aplikace obnovit, aniž by bylo nutné provést její restart. Visual Basic .NET takovéto chování nepřipouští: Jakékoliv změny kódu, které uskutečníte v režimu pozastavení běhu aplikace vyžadují, abyste opětovně sestavili aplikaci. Jen pak se mohou objevit vámi provedené modifikace. Z uvedeného plyne, že programová vlastnost, označovaná jako **edit and continue** (upravit a pokračovat) již není nadále podporována.

Ukončete běh aplikace a proveďte následující změnu v kódu zpracovatele události **Click** tlačítka **btnCalculate**.

```
' Round the Salary up to the nearest thousand
RoundSalary = Math.Ceiling(EmpSalary / 1000) * 1000
```

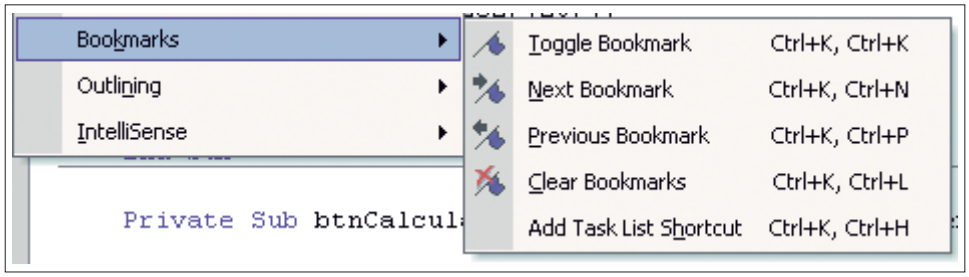
Poté, co jste modifikovali zdrojový kód, uveďte aplikaci znovu do běhu a zjistěte vliv provedené změny.

Přestože je naše ukázková aplikace poněkud jednoduchá, jakékoliv aplikaci netrvá dlouho, než se z ní stane složitý celek. Proto se podíváme na další nástroj, jenž vám pomůže se lépe orientovat v spletité softwarové džungli, kterou si sami při programování vytvoříte.

KDE JSEM BYL?

Pravděpodobně nejohromnější inovaci ve Visual Studiu .NET představuje implementace záložek. Každý, kdo musel někdy vytvářet speciální „komentáře“ za účelem nalezení fragmentu kódu pohřbeného v tisících řádcích programových symbolů, přijde záložkám na chuť velice rychle. Koncepce používání záložek je vám jistě známá z prostředí webových stránek. Nyní s nimi ovšem můžete pracovat i v jazyce Visual Basic .NET.

Budete-li chtít přidat záložku, otevřete nabídku **Edit** a ukažte na položku **Bookmarks**. Okamžitě se rozvine podřazená nabídka, jak můžete vidět na obr. 7.7.

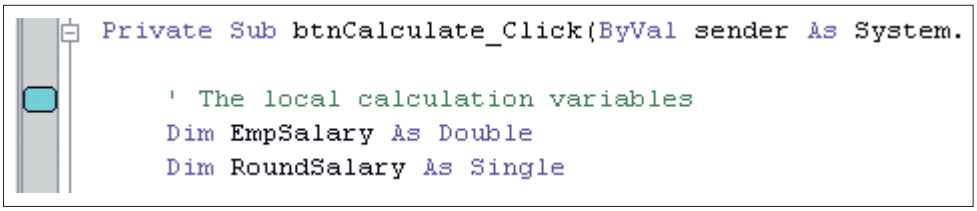


*Obr. 7.7: Podnabídka nabídky **Bookmarks***

Tlačítka pro práci se záložkami můžete najít i na panelu nástrojů **Text Editor** (obr. 7.8).

*Obr. 7.8: Panel tlačítek **Text Editor***

Použití záložek si nejlépe předvedeme, když stávající kód obohatíme ještě o pár řádků. Poklepejte na tlačítko s nápisem **Calculate**, čímž otevřete editor pro zápis kódu. Umístěte kurzor myši na řádek s komentářem *The local calculation variables* a klepněte na tlačítko **Toggle Bookmarks**, nebo dvakrát aktivujte klávesovou zkratku **CTRL+K**. Na levém panelu by se měl objevit indikátor záložky (obr. 7.9).

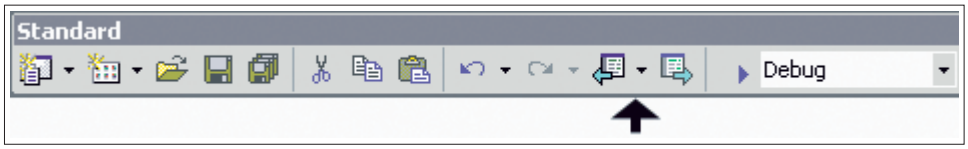


Obr. 7.9: Umístění záložky

Přejděte na řádek s komentářem *Round the Salary...* a i zde přidejte jednu záložku. Přidávání záložek je docela zábavné, že? Nestyďte se a přidejte ještě několik záložek kdekoliv chcete. Všechny záložky, které jste vytvořili jsou dočasně přidány do vašeho zdrojového kódu.

Jestliže se chcete vrátit k dočasné záložce, klepněte na tlačítko **Next Bookmark** nebo **Previous Bookmark**. Pokud raději pracujete s klávesovými zkratkami, můžete stisknout kombinaci zkratek **CTRL+K** a **CTRL+N** pro následující záložku, a **CTRL+K** a **CTRL+P** pro předcházející záložku.

Další možností, jak se vrátit na místo označené záložkou, je použít tlačítka **Navigate Backward**, respektive **Navigate Forward**, která jsou uložena na panelu nástrojů **Standard** (obr. 7.10).



*Obr. 7.10: Tlačítka **Navigate Backward** a **Navigate Forward***

Záložku můžete odstranit opětovným klepnutím na tlačítko **Toggle Bookmark**, nebo dvojitým stisknutím klávesové zkratky **CTRL+K** v okamžiku, kdy kurzor myši spočívá na řádku se záložkou.

Klepnete-li na tlačítko **Clear Bookmarks**, odstraníte všechny vytvořené záložky. Stejného účinku dosáhnete, když použijete kombinaci klávesových zkratek **CTRL+K** a **CTRL+I**.

POUŽITÍ TŘÍD DEBUG A TRACE

Třídy **Debug** a **Trace** nabízejí informace o výkonnosti aplikace. Implicitně se všechny zprávy, které tyto třídy produkují, zobrazují v okně **Output** integrovaného vývojového prostředí Visual Studia .NET. Třídy **Debug** a **Trace** můžete v rámci jedné aplikace používat odděleně nebo společně, ovšem rozhodnutí o tom, které zprávy se budou objevovat v okně **Output** leží na konfiguračních nastaveních projektu. Když se projekt nachází v konfiguraci **Debug Solution Configuration**, obě třídy budou své zprávy posílat do okna **Output**. Pracuje-li projekt v konfiguračním režimu **Release Solution Configuration**, do okna **Output** bude své výstupy směrovat pouze třída **Trace**.

Nyní je vhodný okamžik při ověření vaší aktuální projektové konfigurace. Postupujte podle těchto instrukcí:

1. Otevřete, nebo vyhledejte okno **Solution Explorer** (**CTRL+R**).

2. Na název vyvíjeného projektu klepněte pravým tlačítkem myši, čímž získáte přístup ke kontextové nabídce.

3. Klepněte na položku **Properties**. Za okamžik uvidíte dialogové okno **Property Pages**.

4. V levém panelu otevřete složku **Configuration**.

5. Ujistěte se, že šipka směruje na položku **Debugging**.

6. Z rozevíracího seznamu **Configuration** vyberte volbu **Active** (**Debug**) nebo **Debug** a poté aktivujte tlačítko OK.

Třídy **Debug** a **Trace** sdílí většinu svých metod, jako například **WriteLine** či **Assert**. Několik metod si přiblížíme a poukážeme na jejich užitečnost.

Poklepejte na tlačítko **btnCalculate** a za veškerý kód, jenž se nachází v jeho zpracovateli události **Click**, vložte tyto řádky programového kódu:

```
Debug.WriteLine("Debug Information - Starting")
Debug.Indent()
Debug.WriteLine("The rounded salary is " & RoundSalary, "Field")
Debug.Unindent()
Debug.Assert(RoundSalary <> 0.0, "No Salary to Calculate Insurance.")
Debug.WriteLine("Debug Information - Ending")
```

Uvedený výpis kódu nám pomůže prozkoumat některé metody třídy **Debug**. Pomocí metody **WriteLine** lze zasílat a zobrazovat zprávy v okně **Output**. Zobrazování informací sice nedělá žádné potíže ani metodě **Write**, ovšem jenom metoda **WriteLine** vkládá za textem i znak nového řádku (metoda **Write** touto schopností nedisponuje).

Aktivací metody **Indent** zabezpečíte odsazení všech následujících zpráv, které se budou zobrazovat v okně **Output**. Naopak, budete-li chtít odsazení zrušit, sáhněte nejspíš po metodě **Unindent**. Použitím odsazení můžete vizuálně seskupovat informace, mezi kterými existují jisté vzájemné vazby.

Metoda **Assert** pracuje následovně: Nejprve vyhodnotí výraz, jenž tvoří první argument, a když je tento nepravdivý (vyhodnocen na hodnotu **False**), zobrazí hodnotu druhého argumentu v okně **Output**. Jako přídavek je rovněž zobrazeno i modální dialogové okno se zprávou, jménem projektu a číslem příkazu **Debug.Assert**. Okno obsahuje rovněž tři tlačítka:

- **Abort** (Přerušit): Ukončí běh aplikace.
- **Retry** (Opakovat): Aplikace vstoupí do ladícího módu.
- **Ignore** (Přeskočit): Aplikace bude pokračovat ve své činnosti.

Předtím, než může aplikace pokračovat, musí uživatel aktivovat některé z výše uvedených tlačítek.

Ještě než aplikaci uvedete do režimu běhu, stiskněte kombinaci kláves **CTRL+ALT+O** a ujistěte se, že okno **Output** je otevřené. Nyní klikněte na tlačítko **Start**, nebo stiskněte klávesu **F5**. Aplikace se spustí a vy můžete otestovat práci nově přidaného programového kódu. Klepněte na tlačítko s nápisem **Calculate** a všimněte si zprávy v okně **Output**.

Debug Information – Starting

Field: The rounded salary is 0

––– DEBUG ASSERTION FAILED –––

––– Assert Short Message –––

No Salary to Calculate Insurance.

––– Assert Long Message –––

a tak dále...

Okno **Output** zobrazuje zprávy podle scénáře, který byl definován. Metoda **Assert** by měla otevřít modální dialog, jenž bude oznamovat, že nebyla zadána žádná částka k počítání. Vyberte tlačítko **Ignore** (Přeskočit). Dialogové okno se zavře a aplikace bude pokračovat ve svém běhu.

ZÁVĚR

Na jednoduché aplikaci jsme si demonstrovali použití nových a rozšířených nástrojů pro ladění aplikací, které jsou součástí prostředí Visual Studia .NET. Možnosti správy chyb a ladění aplikací v jazyce Visual Basic .NET jsou takřka nekonečné. Pomineme-li pár změn v terminologii či úpravy umístění položek nabídek, můžeme říci, že všechny časem prověřené a pro vývojáře ve Visual Basicu známé nástroje zůstaly poplatné svému původnímu významu. Nově zařazené prvky pomáhají nejenom minimalizovat běžné postupy realizované mnoha vývojáři, nýbrž také otevírají dveře pro vývoj automatizovaných ladících nástrojů.

8 Datové vazby a objekty pro přístup k datům v knihovně Windows Forms

Autor: Rockford Lhotka, Magenic Technologies

SHRNUTÍ

Rocky Lhotka vám ukáže, jak psát programový kód pro lepší podporu práce s daty, jenž můžete přidávat do vašich obchodních tříd či tříd kolekcí.

Ukázkový soubor **vbobjectbinding.exe** si můžete stáhnout z adresy
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnadvnet/html/vbnet02252003.asp

Když vývojáři Microsoftu vytvářeli rozhraní knihoven Windows Forms a Web Forms, pamatovali na to, aby byla práce s daty tak flexibilní a přínosná, jak jen to jde. Přestože s databázemi jste ve Visual Basicu mohli pracovat již před mnoha lety, teprve nyní je interakce s daty skutečně použitelná pro vývoj širokého spektra počítačových aplikací.

Jednou z klíčových inovací je, že interakci s daty lze realizovat nejenom přes **DataSet**, nýbrž také pomocí objektů, struktur a kolekcí objektů a struktur. Vytváření vazeb mezi databázovými objekty a instancemi ovládacích prvků na formulářích nevyžadují nyní žádnou dodatečnou práci na straně vývojáře. Vše je konfigurováno zcela automaticky.

Interakce s daty ve Web Forms je pouze jednosměrná (někdy označována také jako „jenom pro čtení“). Řečeno jinými slovy, nejprve jsou z datového zdroje, jímž může být třeba **DataSet**, objekt nebo kolekce, získána potřebná data, která jsou pak zobrazena pomocí instancí ovládacích prvků na straně klienta. Proces interakce s daty má tedy předem stanovený styl chování, přičemž nevyžaduje žádnou extra námahu, nakolik vývojáři vytvářejí pouze objekty nebo uživatelské rozhraní.

Práce s daty v prostředí Windows Forms je obousměrná (tedy určená „pro čtení i zápis“), a tím pádem i komplexnější. V tomto případě jsou získána data zobrazena v prvcích grafického uživatelského rozhraní, ovšem jakákoliv změna zobrazených dat se promítne i do původního datového zdroje. I zde platí, že většinu práce obstará inteligentní software za vás, ovšem pokud budete chtít aplikovat pokročilé techniky interakce s daty, budete přece jenom muset přikročit k psaní několika fragmentů zdrojového kódu.

Cílem této kapitoly je ukázat vám, jak můžete do vašich stávajících tříd začlenit podporu pro práci a komunikaci s datovými zdroji pomocí nových dovedností, které nabízí knihovna Windows Forms. Mezi tyto dovednosti patří:

- Schopnost naprogramovat objekt nebo kolekci tak, aby při změně dat tato entita informovala prvky uživatelského rozhraní o nové, aktualizované situaci.
- Umožnění prvku **DataGrid** řádné navázání spojení s prázdnou kolekcí.
- Lokální editování odvozených objektů v prvku **DataGrid**.
- Dynamické přidávání nebo odstraňování odvozených objektů v prvku **DataGrid**.

Pokud budeme pracovat s jednoduchými objekty, můžeme pro notifikaci změny datových hodnot implementovat speciální události. Přidáním těchto událostí docílíme toho, že uživatelské rozhraní aplikace bude zobrazovat aktualizovaná data pokaždé, když dojde k jejich změně. Rovněž si ukážeme, jak porozumět procesu notifikace uživatelského rozhraní v případě, kdy dojde k porušení pravidla validace kvůli nově zadaným údajům. Nekorektní implementace validace může totiž způsobit, že interakce s daty nemusí dosáhnout očekávaného výsledku.

Navíc, existuje několik volitelných programových prvků, které je možné začlenit do kolekcí. Kolekce jsou běžně vázané na instance ovládacích prvků jako je třeba **DataGrid**. Když vytvoříme a implementujeme typově silnou (strongly-typed) kolekci objektů, můžeme pomocí prvku **DataGrid** inteligentně pracovat s naší kolekcí a jejími odvozenými objekty. Kromě toho je také možné implementovat rozhraní **IBindingList**, jehož pomocí může naše kolekce chytře realizovat různorodé akce s daty v instanci ovládacího prvku **DataGrid**.

Nakonec, vývojová platforma nám dovoluje používat i objekty, které jsou umístěny uvnitř kolekce. Těmto objektům říkáme odvozené objekty, nebo také synovské objekty. Odvozené objekty mohou implementovat rozhraní **IEditableObject**, takže **DataGrid** může bez jakýkoliv potíží komunikovat s objekty během lokální editace. Odvozené objekty mohou rovněž implementovat rozhraní **IDataErrorInfo**. To znamená, že jestliže dojde k porušení pravidel validace, **DataGrid** může označit ten řádek databáze, v němž došlo k chybě.

JEDNODUCHÁ UKÁZKA INTERAKCE S DATY V PROSTŘEDÍ WINDOWS FORMS

Proces navázání spojení mezi vlastnostmi objektu a vlastnostmi instance ovládacího prvku na formuláři není nijak složitý. Uvažujme o následující třídě s názvem **Order**:

```
Public Class Order
   Private mID As String = ""
   Private mCustomer As String = ""

   Public Property ID() As String
      Get
         Return mID
      End Get
      Set(ByVal Value As String)
         mID = Value
      End Set
   End Property

   Public Property Customer() As String
      Get
         Return mCustomer
      End Get
      Set(ByVal Value As String)
         mCustomer = Value
      End Set
   End Property

End Class
```

Jediným speciálním kódem je deklarace datových členů třídy:

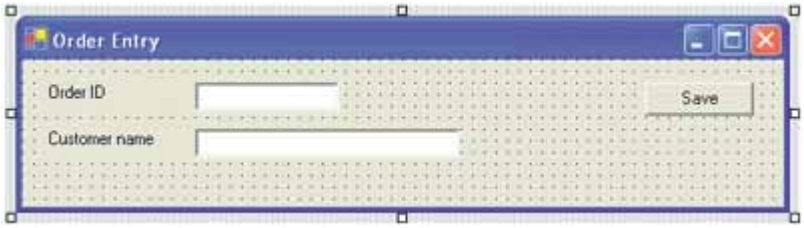
```
Private mID As String = ""
Private mCustomer As String = ""
```

Všimněte si, že proměnné jsou při své deklaraci inicializovány na implicitní hodnoty. Tato explicitní inicializace proměnných není ve Visual Basicu .NET typická, protože tento jazyk provádí automatickou inicializaci všech řádně deklarovaných proměnných.

Důvodem, proč explicitně inicializujeme výše uvedené proměnné je, že pokud bychom tak neudělali, pokus o navázání spojení s daty by nebyl úspěšný. Automatická inicializace proměnných se totiž neuskuteční v době, kdy se bude datový zdroj snažit komunikovat s naším objektem, což zapříčiní vznik chyby a generování chybové výjimky v okamžiku, kdy přijde požadavek na získání hodnot z neinicializovaných proměnných.

Nicméně, pokud budeme proměnné explicitně inicializovat, tato inicializace se uskuteční ještě předtím, než dojde k interakci mezi daty a naším objektem. Z uvedeného tedy plyne, že proměnné jsou řádně inicializovány v okamžiku, kdy budou v procesu vázání dat získány jejich hodnoty, čímž se úspěšně vyhneme vzniku chybové výjimky za běhu programu.

Když připravíme podobný formulář jako je na obr. 8.1, budeme moci poměrně snadno provést spojení vlastností objektu s vlastnostmi instancí ovládacích prvků v době zavádění formuláře do operační paměti počítače.



Obr. 8.1: Jednoduchý formulář pro práci s objekty třídy ***Order***

Programový kód, jenž má na starosti přepojení objektu třídy **Order** a formuláře by mohl vypadat třeba takto:

```
Private mOrder As Order

Private Sub OrderEntry_Load(ByVal sender As System.Object, _
           ByVal e As System.EventArgs) Handles MyBase.Load

   mOrder = New Order()

   txtID.DataBindings.Add("Text", mOrder, "ID")
   txtCustomer.DataBindings.Add("Text", mOrder, "Customer")

End Sub
```

Tajemství spočívá ve skutečnosti, že každý ovládací prvek z knihovny Windows Forms disponuje kolekcí s názvem **DataBindings**. Tato kolekce obsahuje seznam vazeb, které existují mezi vlastnostmi ovládacího prvku a vlastnostmi datového zdroje (nebo datových zdrojů). Zajímavým vedlejším efektem tohoto přístupu je, že vývojáři mohou provádět vzájemná spojení mezi vlastnostmi datového zdroje a vlastnostmi instancí několika odlišných ovládacích prvků. Dokonce je možné, aby bylo provedeno spojení mezi různými vlastnostmi instance ovládacího prvku a vlastnostmi většího počtu datových zdrojů.

Pomocí jednoduchého schématu navazování spojení s daty můžeme vytvářet docela souhrnné návrhy uživatelského rozhraní aplikací. Například v ukázkovém programovém kódu této kapitoly si předvedeme, jak provést navázání vlastnosti **Enabled** tlačítka **Save** na vlastnost **IsValid** obchodního objektu. Prostřednictvím tohoto postupu můžeme zaručit, že tlačítko bude přístupné uživateli jenom ve chvíli, když bude objekt připraven k uložení svého stavu.

Bude dobré, když si zapamatujete, že proces vytváření spojení s daty je obousměrný. Je tedy možné nejenom zobrazit všechna relevantní data objektu na formuláři, ovšem jakákoliv změna, kterou uživatel se zobrazenými daty provede, se okamžitě promítne do stavu objektu. Kupříkladu, jestliže uživatel změní hodnotu v textovém poli **txtID**, nová situace se dotkne samotného objektu v okamžiku, kdy uživatel stiskne tabulátor pro přesun zaměření na další instanci jistého ovládacího prvku. Nově zadanými daty je objekt aktualizován pomocí bloku **Set** v kódu vlastnosti. To je příjemné, neboť to znamená, že náš stávající kód v proceduře vlastnosti (**Property**) je automaticky vyvolán. Pro zabezpečení obousměrného procesu interakce s daty není potřebná žádná další aktivita programátora.

OZNAMOVÁNÍ ZMĚN VLASTNOSTÍ

Poté, co jsme si ukázali, jak jednoduše je možné provést spojení mezi objektem a instancemi ovládacích prvků, se můžeme blíže podívat na to, jak rozšířit dovednosti našeho objektu tak, aby byl schopen automaticky oznamovat změnu vlastností. Předestřená problémová situace je následovní: Jestliže jiný kód naší aplikace uskuteční změnu dat objektu, instance ovládacích prvků nedisponují žádným mechanizmem, jehož pomocí by se mohly o realizované změně dat dozvědět. Výsledkem proto bude ztráta synchronizace mezi objektem a prvky uživatelského rozhraní, která vyústí do zobrazování nesprávných údajů.

To, co potřebujeme, je najít cestu, jak přimět objekt k tomu, aby o změně svých dat informoval instance ovládacích prvků na formuláři. Cestu směrem k cíli představují události, které můžeme deklarovat a vyvolávat z prostředí našeho objektu. Když provedeme spojení mezi instancí ovládacího prvku a vlastností našeho objektu, automaticky bude rozpoznáván vznik události se jménem **propertyChanged**, kde **property** je název specifické vlastnosti objektu.

Naše třída **Order** definuje vlastnost **ID**. Když navážeme vlastnost **ID** na instanci ovládacího prvku, vytvořená datová vazba bude naslouchat události **IDChanged**. Bude-li tato událost vyvolána naším objektem, pomocí datové vazby bude automaticky obnoven obsah všech instancí, které jsou navázány na tento objekt.

Rozšiřme tedy třídu **Order** o deklaraci dvou událostí:

```
Public Class Order
    Public Event IDChanged As EventHandler
    Public Event CustomerChanged As EventHandler
```

Všimněte si, že typem událostí je delegát s názvem **EventHandler**. Takováto deklarace je nutná proto, aby mohla datová vazba rozeznat událost. Kdybychom události nedeklarovali uvedeným způsobem, v okamžiku, kdy by se datová vazba pokusila komunikovat s naším objektem, by byla generována chybová výjimka za běhu programu.

Delegát **EventHandler** je součástí standardního událostního modelu, jenž je uplatňován v prostředí Windows Forms. Definuje událost pomocí dvou parametrů: **sender**, což je objekt, který vyvolal vznik události a **EventArgs**, jenž představuje objekt s dodatečnými daty, které může zpracovatel události při své práci potřebovat.

Poté, co dokončíme deklaraci událostí, se musíme ujistit, že tyto události budou generovány vždy, když dojde k modifikaci hodnot příslušných vlastností. Kód pro vyvolání události bychom mohli umístit do bloku **Set** procedury vlastnosti **Property**, třeba takto:

```
Public Property ID() As String
    Get
        Return mID
    End Get
    Set(ByVal Value As String)
        mID = Value
        RaiseEvent IDChanged(Me, New EventArgs())
    End Set
End Property
```

Měli byste mít na paměti skutečnost, že kdykoliv dojde ke změně hodnoty proměnné **mID** v těle třídy, je zapotřebí generovat odpovídající událost. Většina tříd obsahuje kód, jenž modifikuje interní proměnné (kromě změny hodnot těchto proměnných v blocích **Set** vlastností). Jednoduše řečeno, je-li změněna hodnota proměnné, musíme provést vyvolání příslušné události.

Pro lepší názornost si předvedeme další programovou ukázku. Předpokládejme, že náš objekt **Order** bude disponovat kolekcí objektů **LineItem**. Tuto kolekci budeme implementovat později, ovšem nyní se soustřeďme na fragment kódu třídy **LineItem**, v němž dochází k deklaraci veřejných událostí a privátních datových členů:

```
Public Class LineItem
  Public Event ProductChanged As EventHandler
  Public Event QuantityChanged As EventHandler
  Public Event PriceChanged As EventHandler
  Public Event AmountChanged As EventHandler

  Private mProduct As String
  Private mQuantity As Integer
  Private mPrice As Double
```

Při pohledu na výpis zdrojového kódu jste jistě postřehli, že pracujeme se čtyřmi událostmi (každá událost připadá na jednu vlastnost), ovšem deklarované jsou jenom tři soukromé proměnné. Vlastnost **Amount** bude vypočítávat součin hodnot proměnných **mQuantity** a **mPrice**:

```
Public ReadOnly Property Amount() As Double
   Get
      Return mQuantity * mPrice
   End Get
End Property
```

I když je vlastnost **Amount** určena pouze pro čtení (klíčové slovo **ReadOnly**), může se její hodnota měnit. Ve skutečnosti se návratová hodnota vlastnosti změní vždy, když dojde k modifikaci hodnot jedné z proměnných **mQuantity** a **mPrice**. Abychom naznačili, že hodnota vlastnosti se změnila, můžeme výslovně vynutit generování události. Kupříkladu, když se změní hodnota proměnné **mPrice**.

```
Public Property Price() As Double
   Get
      Return mPrice
   End Get
   Set(ByVal Value As Double)
      mPrice = Value
      RaiseEvent PriceChanged(Me, New EventArgs())
      RaiseEvent AmountChanged(Me, New EventArgs())
   End Set
End Property
```

Kromě toho, že generujeme událost **PriceChanged** v důsledku změny vlastnosti **Price**, rovněž vyvoláváme i událost **AmountChanged**, protože jsme nepřímo ovlivnili také změnu vlastnosti **Amount**. Je tedy zřejmé, jak ostražití musíme být, abychom garantovali, že události budou generovány ve vhodné chvíli.

Abychom byli zcela přesní, volání události **AmountChanged** není striktně nezbytné. Když propojíme instance ovládacích prvků formuláře s vlastnostmi objektu, datová vazba bude reagovat na události ve stylu **propertyChanged** pro každou vlastnost, se kterou jsou instance přepojeny. Je-li vyvolána jakákoliv událost, všechny instance, které jsou navázány na objekt, budou aktualizovány.

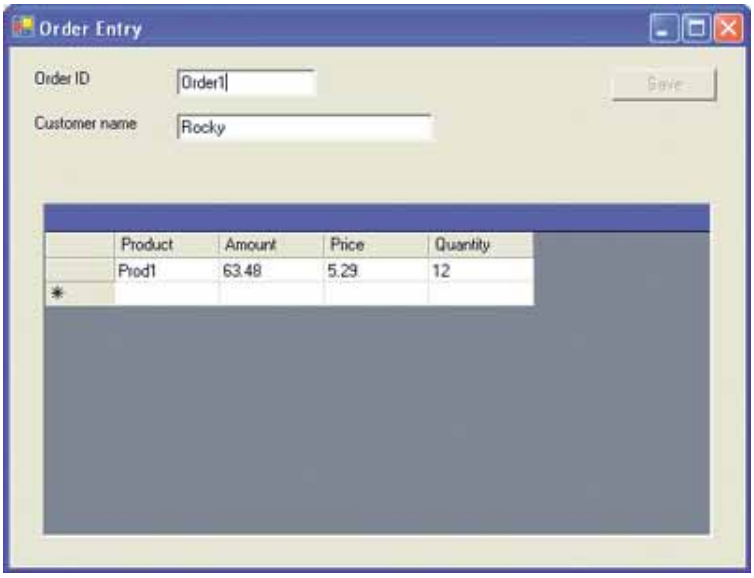
Řečeno jinými slovy, jestliže náš formulář bude disponovat instancemi, které budou napojeny na vlastnosti **Price** a **Amount**, vyvolání události **PriceChanged** způsobí aktualizaci nejenom instance, která je navázána na vlastnost **Price**, nýbrž také instance, která je napojena na vlastnost **Amount**.

Zmíněný styl má jednu nevýhodu, která se projevuje ve skutečnosti, že uživatelské rozhraní se stává poněkud závislé na implementaci objektu. Povězme, že se později rozhodneme provádět propojení pouze s vlastností **Amount**, ovšem v tomto případě naše uživatelské rozhraní nebude pracovat správně, protože jednoduše nedojde k vyvolání události **AmountChanged**. Abychom předešli podobným potížím, nejlépe uděláme, když budeme deklarovat a generovat událost **propertyChanged** pro každou vlastnost našeho objektu.

Zbytek zdrojového kódu třídy **LineItem** můžete najít v programové ukázce, kterou si můžete volně stáhnout z Internetu (přesná adresa je uvedena na začátku kapitoly).

VYTVÁŘENÍ VAZEB S TYPOVĚ SILNOU (STRONGLY-TYPED) KOLEKCÍ

Jak jsme již vzpomenuli, náš objekt **Order** obsahuje kolekci objektů **LineItem**. Abychom umožnili uživatelům snadno přidávat, odstraňovat či měnit objekty **LineItem**, můžeme naše uživatelské rozhraní rozšířit o instanci ovládacího prvku **DataGrid**, kterou posléze navážeme na tuto kolekci objektů. Ukázku uživatelského rozhraní můžete vidět na obr. 8.2.



*Obr. 8.2: Formulář s instancí prvku **DataGrid**, která je svázána s kolekcí objektů*

Vzájemné propojení instance ovládacího prvku **DataGrid** a pole nebo kolekce objektů je možné realizovat pomocí jednoho řádku kódu:

```
Dim arLineItems As New ArrayList()
dgLineItems.DataSource = arLineItems
```

Ačkoliv zobrazování dat v datové mřížce pracuje spolehlivě, neposkytuje nám využití všech možností, které jsme od instance prvku **DataGrid** očekávali. Důvod je jednoduchý: Základní pole nebo kolekce tříd nenabízí dostatečné množství informací pro instanci prvku **DataGrid**. Instance proto nemůže pracovat stylem, jenž můžeme využít při propojení s objektem **DataTable**.

Abychom lépe využili dovedností instance ovládacího prvku **DataGrid**, můžeme vytvořit objekt s typově silnou kolekcí. Poté bychom měli být schopni provést následující akce:

- Realizace propojení s prázdnou kolekcí
- Upozornění instance prvku **DataGrid** na změny kolekce
- Dynamické přidávání a odstraňování odvozených objektů
- Lokální editování odvozených objektů

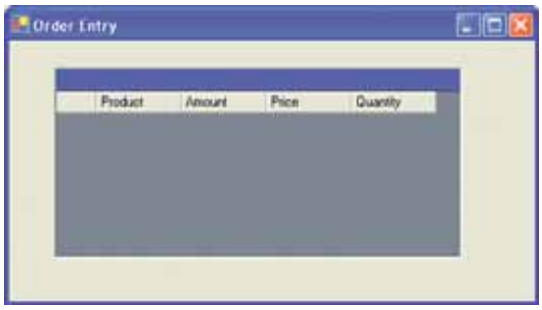
Nyní vyřešme tyto problémy jednou provždy. První z nich, jenž říká o propojení s prázdnou kolekcí, je relativně snadno vyřešitelný. Problémem je, že instance prvku **DataGrid** potřebuje vědět, kolik sloupců je zapotřebí pro zobrazení dat z datového zdroje. Propojíme-li instanci s jednoduchým polem **ArrayList**, nebo objektem kolekce, jak může instance zjistit požadovanou informaci?

Odpověď je následovní: Instance se „podívá“ na první položku datové kolekce a poté použije mechanizmus reflexe, aby získala seznam veřejně přístupných vlastností a datových členů této položky (může jít o strukturu, objekt, nebo primitivní datový typ, jakým je třeba **Integer**).

Pokud je ovšem kolekce prázdná, popsaná technika nebude fungovat a instance prvku **DataGrid** nebude schopna automaticky sestavit seznam sloupců. Výsledkem bude, že instance nebude naplněna žádným obsahem a bude tedy naprosto prázdná (obr. 8.3).

*Obr. 8.3: Instance prvku **DataGrid**, která je propojená s prázdnou kolekcí*

Vše, co musíme udělat pro zdárné vyřešení prvního problému, je vytvořit uživatelskou, typově silnou kolekci s výchozí vlastností s názvem **Item**. Instance prvku **DataGrid** může použít typ vlastnosti **Item** pro získání seznamu specifických veřejně přístupných vlastností a datových členů tohoto typu. Proto již nepotřebuje analyzovat první položku v kolekci pro zjištění seznamu sloupců, a tudíž nám nic nebrání v tom, abychom propojili instanci s prázdnou kolekcí. A vše bude pracovat tak, jak má (obr. 8.4).



*Obr. 8.4: Instance prvku **DataGrid** propojená s prázdnou, typově silnou kolekcí*

Když se podíváte na předcházející dva obrázky a porovnáte je, je zřejmé, že typově silná kolekce je preferovanějším řešením, protože pomocí ní zobrazuje instance prvku **DataGrid** seznam záhlaví sloupců, i když prozatím nejsou přítomné žádné další položky.

Vytváření typově silných kolekcí je přímočaré. Následující kroky nastiňují základní proces:

- Do stávajícího projektu přidejte novou třídu
- Odvoďte třídu od třídy **System.Collections.CollectionBase**
- Implementujte výchozí typově silnou vlastnost **Item**
- Implementujte typově silné metody **Add** a **Remove**

Kupříkladu, v programové ukázce můžete najít třídu **LineItem**, která implementuje typově silnou kolekci objektů **LineItem**. Použitím vlastnosti **Item** je možné propojit instanci prvku **DataGrid** s prázdnou kolekcí:

```
Default Public ReadOnly Property Item(ByVal index As Integer) _
        As LineItem
   Get
      Return CType(list(index), LineItem)
   End Get
End Property
```

Vlastnost je typově silná a vrací objekty typu **LineItem**. Další důležitou věcí je, že vlastnost je deklarována pomocí klíčových slov **Default** a **Property** (ne **Function**). Vytváření datových vazeb explicitně vyžaduje uvedenou deklaraci pro svoji zdárnou realizaci.

Programová ukázka obsahuje také typově silné implementace metod **Add** a **Remove**, které jsou součástí jakékoliv typově silné kolekce, ovšem nejsou vyžadovány pro úspěšné završení procesu vytváření datových vazeb.

První problémovou situaci jsme rozřešili poměrně rychle a přitom jsme nemuseli psát ani mnoho řádků zdrojového kódu. V následujících podkapitolách prozkoumáme řešení dalších potenciálních problémových oblastí.

IMPLEMENTACE ROZHRANÍ IBINDINGLIST

Datové vazby jsou vytvářeny pomocí formálního schématu, podle něhož může kolekce indikovat svoji změnu. Tento rys je možné začlenit pomocí implementace rozhraní **IBindingList**, které obsahuje událost **ListChanged**. Kdykoliv dojde ke změně kolekce (kvůli přidání, odstranění nebo změně položky), měla by být vyvolána tato událost, aby sdělila datové vazbě, že spodní vrstva dat byla modifikována.

Rozhraní **IBindingList** podporuje víc, než jen jednu formu notifikace změny dat. Tab. 8.1 představuje volitelné prvky, které můžeme využít za předpokladu, že budeme implementovat toto rozhraní:

Tab. 8.1

VOLITELNÝ PROGRAMOVÝ RYS	CHARAKTERISTIKA
Notifikace změny	Informuje uživatelské rozhraní aplikace o jakékoliv změně kolekce (přidávání, odstraňování a editace jednotlivých položek).
Automatické přidávání položek	Dovoluje instanci prvku **DataGrid** vkládat nové položky z kolekce v okamžiku, kdy uživatel dosáhne konce mřížky.
Automatické odstraňování položek	Dovoluje instanci prvku **DataGrid** odstraňovat položky z kolekce v okamžiku, kdy uživatel stiskne klávesu **Delete**.
Lokální editování položek	Dovoluje instanci prvku **DataGrid** vykonávat lokální úpravu položek v kolekci (je rovněž vyžadováno, aby položka implementovala rozhraní **IEditableObject**, o němž si povíme později v této kapitole).
Vyhledávání	Aktivuje metodu **Find** pro prohledání kolekce a zjištění, zdali se v kolekci nachází specifická položka.
Třídění	Aktivuje metodu **Sort**, která provádí třídění kolekce podle jednotlivých sloupců.

Je na vývojářích, aby implementovali každou z těchto metod. Rozhraní **IBindingList** pouze definuje jednotlivé vlastnosti, metody a události, ovšem samotný implementační kód musíme napsat sami. Naštěstí, implementace všech uvedených programových rysů není povinná, nýbrž volitelná. Rozhraní **IBindingList** definuje sadu vlastností s návratovou hodnotou typu **Boolean**, které můžeme použít pro určení, jaké programové rysy jistá kolekce implementuje. Zde je jejich seznam:

- **SupportsChangeNotification**
- **AllowNew**
- **AllowEdit**
- **AllowRemove**
- **SupportsSearching**
- **SupportsSorting**

Budete-li chtít implementovat ten-který programový rys, nastavte příslušnou vlastnost na hodnotu **True**. V programové ukázce, kterou si můžete stáhnout z Internetu, najdete kolekci **LineItems**, která implementuje notifikaci změny, přidávání, upravování a odstraňování položek. Na druhou stranu, podpora vyhledávání a třídění nebyla zařazena, a tudíž návratovou hodnotou těchto vlastností je hodnota **False**.

Základní implementace rozhraní vyžaduje použití klíčového slova **Implements**:

```
Public Class LineItems
   Inherits CollectionBase
   Implements IBindingList
```

Když přidáme tento příkaz, naznačujeme, že provádíme implementaci rozhraní, a proto musíme začlenit implementaci všech metod, které toto rozhraní definuje. To však zahrnuje také metody, které možná nebudeme chtít v dané chvíli implementovat, jako třeba **SortDirection** (implementaci třídění jsme vynechali). Podstatné je, že do těl metod, které nevyužíváváme, nemusíme psát žádný zdrojový kód, ovšem je nutné, abychom vytvořili programovou kostru těchto metod. Ukázku nabízí již zmiňovaná metoda **SortDirection**:

```
Public ReadOnly Property SortDirection() _
       As System.ComponentModel.ListSortDirection _
       Implements System.ComponentModel.IBindingList.SortDirection
   Get

   End Get
End Property
```

NOTIFIKACE ZMĚNY

Jelikož zavádíme podporu pro notifikaci změn, budeme muset datovou vazbu informovat vždy, když dojde ke změně dat kolekce. To znamená, že pro ošetření stavů, když bude přidána, odstraněna či upravena jakákoliv položka, bude zapotřebí napsat několik fragmentů programového kódu.

V prvním kroku potvrdíme náš zájem o podporu notifikace změn:

```
Public ReadOnly Property SupportsChangeNotification() _
     As Boolean Implements _
        System.ComponentModel.IBindingList.SupportsChangeNotification
   Get
      Return True
   End Get
End Property
```

Dále budeme pokračovat deklarováním události **ListChanged**, jak je ostatně definováno v rozhraní:

```
Public Event ListChanged(ByVal sender As Object, _
ByVal e As System.ComponentModel.ListChangedEventArgs) _
Implements System.ComponentModel.IBindingList.ListChanged
```

Vše, co budeme muset udělat, je vyvolat tuto událost vždy, když dojde ke změně kolekce. To není tak těžké, jak se může zdát, protože třída **CollectionBase** definuje sadu metod, které můžeme překrýt. Tak budeme přesně vědět, kdy nastala změna kolekce. Potřebné metody prezentuje následující seznam:

- **OnClearComplete**
- **OnInsertComplete**
- **OnRemoveComplete**
- **OnSetComplete**

Událost **ListChanged** můžeme vyvolat zevnitř kterékoliv z uvedených metod. Kupříkladu, tuto událost můžeme generovat v okamžiku, kdy bude přidána nová položka:

```
Protected Overrides Sub OnInsertComplete(ByVal index As Integer, _
      ByVal value As Object)

   RaiseEvent ListChanged( _
      Me, New ListChangedEventArgs(ListChangedType.ItemAdded, index))

End Sub
```

Náš objekt kolekce bude nyní oznamovat prvkům uživatelského rozhraní veškeré změny, což znamená, že všechny instance ovládacích prvků, které jsou závislé na datech (jako třeba **DataGrid**) mohou kdykoliv přesně odrážet aktuální stav naší kolekce.

EDITOVÁNÍ A ODSTRAŇOVÁNÍ POLOŽEK

Rozhraní **IBindingList** definuje vlastnosti **AllowEdit** a **AllowRemove**, jejichž pomocí můžeme determinovat aktivaci programových rysů, jimiž jsou lokální editování a dynamické odstraňování položek z kolekce. Rozhraní **IBindingList** neposkytuje definici žádných jiných metod, které by uvedené programové dovednosti podporovaly. Prostřednictvím vlastností můžeme jednoduše aktivovat, či deaktivovat podporu pokročilého chování instancí ovládacího prvku **DataGrid**.

Pokud nastavíme vlastnost **AllowEdit** na hodnotu **True**, musíme se ujistit, že odvozené objekty budou podporovat lokální editování dat prostřednictvím implementování rozhraní **IEditableObject** (o něm si povíme později). Nebudou-li odvozené objekty implementovat toto rozhraní, nebudou sice generovány žádné chybové výjimky, ovšem rovněž tak se nebudeme moci těšit z řádně odvedené práce. Řešením je, samozřejmě, implementování uvedeného rozhraní.

PŘIDÁVÁNÍ POLOŽEK

Pomocí instancí ovládacího prvku **DataGrid** lze dynamicky přidávat nové položky do datových kolekcí. Tento programový prvek je značně uživatelsky přívětivý, neboť dovoluje uživateli jednoduše přejít na konec mřížky a zadávat nová data do automaticky přidaného nového řádku. Tato vlastnost je závislá od podpory lokálního editování dat. Když povolíme přidávání položek, musíme rovněž aktivovat i možnost jejich editování. To znamená, že vlastnost **AllowEdit** musí vracet hodnotu **True** a všechny odvozené objekty musí implementovat rozhraní **IEditableObject**.

Rozhraní **IBindingList** definuje vlastnost **AllowNew**, která, pokud je aktivovaná, musí vracet hodnotu **True**. Rozhraní dále definuje i metodu **AddNew**, kterou je rovněž zapotřebí implementovat. Tato metoda musí obsahovat kód pro vytvoření nového odvozeného objektu, jeho přidání do kolekce a jeho vrácení v podobě návratové hodnoty.

Nový objekt **LineItem** můžeme vytvořit následovně:

```
Public Function AddNew() As Object Implements _
System.ComponentModel.IBindingList.AddNew

   Dim item As New LineItem()
   list.Add(item)
   Return item

End Function
```

Odvozené objekty musíme vytvářet bez jakéhokoliv uživatelského vstupu, což je zřejmě nejobtížnější partií celé akce. Samotná instance prvku **DataGrid** vyžaduje přidání odvozených objektů, z čehož plyne, že musíme být schopní tyto objekty vytvářet programově, a to na požádání (jak ukazuje programový kód).

Některé návrhy odvozených objektů vyžadují přítomnost datového konstruktoru, který se využívá pro datovou inicializaci těchto objektů. Takovéto objektové modely nebudou v tomto případě pracovat, protože jednoduše neexistuje žádný způsob, jak získat informace o odvozených objektech od uživatele ještě předtím, než jsou tyto objekty vytvořeny.

Když jsme implementovali vlastnost **AllowNew** a metodu **AddNew**, instance prvku **DataGrid** umožní uživateli přesun na úpatí datové mřížky a následující automatické přidání nových odvozených objektů do kolekce (a mřížky) pro pozdější úpravu.

IMPLEMENTACE ROZHRANÍ IEDITABLEOBJECT V ODVOZENÝCH TŘÍDÁCH

Jak jsme si již pověděli, pro lokální editaci dat je zapotřebí nejenom implementace rozhraní **IBindingList** v uživatelských kolekcích, ale také implementace rozhraní **IEditableObject** v odvozených třídách.

Rozhraní **IEditableObject** vypadá na první pohled až klamlivě jednoduše. Definuje pouze tři metody, jak můžete vidět v tab. 8.2.

Tab. 8.2

METODA	DEFINICE
BeginEdit	Metoda je volána datovou vazbou, přičemž indikuje start editačního procesu. Rovněž objekt informuje o potřebě vytvoření obrazu jeho aktuálního stavu.
CancelEdit	Metoda je volána datovou vazbou, přičemž indikuje, že editační proces je u konce. Rovněž objekt informuje o potřebě uvedení jeho hodnot do původního stavu.
EndEdit	Metoda je volána datovou vazbou, přičemž indikuje, že editační proces je u konce a že objekt by si měl ponechat své změněné hodnoty.

KOPÍROVÁNÍ DAT

Implementace rozhraní **IEditableObject** vyžaduje vytvoření obrazu aktuálního stavu objektu. Řečeno jinak, je třeba nějakým způsobem udělat kopii hodnot všech instančních proměnných objektu. Existuje několik cest pro obdržení kopie hodnot instančních proměnných. Bohužel, tato problematika již přesahuje rámec našeho povídání. V této kapitole použijeme snad nejjednodušší alternativu: Jednoduše zkopírujeme hodnoty instančních proměnných do nové sady proměnných:

```
mOldProduct = Product
mOldPrice = Price
mOldQuantity = Quantity
```

Obnovení hodnot uskutečníme tak, že budeme realizovat proces kopírování inverzním směrem.

ZJIŠŤOVÁNÍ, ZDALI JSME NOVÍ

Abychom mohli rozhraní **IEditableObject** důkladně implementovat, musíme vědět, zdali byl odvozený objekt nově přidán do kolekce. Důvodem je skutečnost, že nový odvozený objekt bude zapotřebí odstranit z kolekce ve chvíli, kdy uživatel aktivuje klávesu ESC pro zrušení editace. Na druhé straně, jestliže uživatel upravuje dřívější odvozený objekt a poté stiskne klávesu ESC, nebudeme chtít objekt odstranit z kolekce. Jediné co bude nutné udělat, je obnovit stav objektu do předcházející podoby.
Pro vyřešení nastíněného problému budeme potřebovat proměnnou, jejíž pomocí budeme sledovat skutečnost, zdali je objekt nově vytvořený či nikoliv. Název deklarované proměnné bude **mIsNew** a tuto proměnnou budeme již při deklaraci explicitně inicializovat na hodnotu **True**:

```
  Private mIsNew As Boolean = True
```

Jakmile bude dokončen první editační proces, do proměnné bude uložena hodnota **False**. K této situaci dojde při zavolání metod **CancelEdit** a **EndEdit**.

METODA BEGINEDIT

Možná to pro vás bude zajímavé, ovšem během editačního procesu je metoda **BeginEdit** volána několikrát. Podle nápovědy softwarové vývojové sady (Software Development Kit, SDK) platformy .NET Framework bychom měli věnovat svoji pozornost pouze prvnímu volání této metody, přičemž ostatní volání můžeme ignorovat. To také uděláme. Pomůže nám deklarace a použití proměnné datového typu **Boolean** s názvem **mEditing**, jak je uvedeno níže:

```
Public Sub BeginEdit() Implements _
        System.ComponentModel.IEditableObject.BeginEdit
   If Not mEditing Then
      mEditing = True
      mOldProduct = Product
      mOldPrice = Price
      mOldQuantity = Quantity
   End If
End Sub
```

Tato proměnná bude nastavena na hodnotu **False** v případě volání metod **CancelEdit** a **EndEdit**, takže kterýkoliv z budoucích editačních procesů bude patřičně pracovat.

METODA ENDEDIT

Hledáte-li metodu, jejíž implementace bude nejsnadnější, právě jste ji našli. Ano, implementace metody **EndEdit** je skutečně bezproblémová, protože tato metoda je aktivována ve chvíli, kdy je editační proces dokončen a jediné co je zapotřebí udělat, je zachovat změny, které byly provedeny s daty. Abychom poukázali na to, že editační proces byl ukončen, uložíme do proměnné **mEditing** hodnotu **False**:

```
Public Sub EndEdit() Implements _
        System.ComponentModel.IEditableObject.EndEdit
   mEditing = False
   mIsNew = False

End Sub
```

Všimněte si, že v těle této metody dochází také ke změně hodnoty proměnné **mIsNew** (do této proměnné je uložena hodnota **False**). V tuto chvíli víme, že uživatel akceptoval odvozený objekt, jenž již není nadále nový (přinejmenším z perspektivy vytváření datové vazby a instance ovládacího prvku **DataGrid**).

METODA CANCELEDIT

Metoda **CancelEdit** je možná nejsložitější ze všech tří popisovaných metod. Metoda má toho na práci skutečně dost: Nejenom, že musí obnovit hodnoty objektu do podoby hodnot, které byly uschovány při volání metody **BeginEdit**, ovšem ještě k tomu musí zjistit, zdali je odvozený objekt novým objektem. Pokud je odvozený objekt nový, je zapotřebí se ujistit, že byl odstraněn z kolekce.

Abychom objektu kolekce sdělili fakt, že chceme zahájit odstraňování, deklarujeme následující událost:

```
Friend Event RemoveMe(ByVal LineItem As LineItem)
```

Událost pracuje s jedním parametrem, jehož typem je **LineItem**, což znamená, že ve skutečnosti předáváme referenci na samotný odvozený objekt. Takto se můžeme ujistit, že kód kolekce bude vědět, kterou položku má odstranit.

Kód metody **CancelEdit** má tuto podobu:

```
Public Sub CancelEdit() Implements _
         System.ComponentModel.IEditableObject.CancelEdit

   mEditing = False
   Product = mOldProduct
   Price = mOldPrice
   Quantity = mOldQuantity
   If mIsNew Then
      mIsNew = False
      RaiseEvent RemoveMe(Me)
   End If

End Sub
```

Začínáme oznámením, že editační proces je dokončen, což signalizujeme nastavením proměnné **mEditing** na hodnotu **False**. Pokračujeme obnovením stavu objektu na hodnoty, které byly uloženy při volání metody **BeginEdit**. Hodnota každé vlastnosti je obnovena do své předcházející podoby. Nakonec provádíme kontrolu proměnné **mIsNew**, abychom zjistili, zdali máme do činění s nově přidaným odvozeným objektem. Je-li tomu tak, vyvoláváme událost **RemoveMe**, která řekne kolekci, že objekt by měl být odstraněn.

ZPRACOVÁVÁNÍ UDÁLOSTI REMOVEME

Jelikož odvozené objekty mohou nyní vyvolávat události kolekce, musíme rozšířit programový kód uživatelské třídy kolekce tak, aby bylo možné ošetřit vznik požadované události. V první etapě přidáme do třídy kolekce **LineItems** metodu, která bude vystupovat jako zpracovatel události. Kód metody bude tedy vykonán pokaždé, když bude generována událost:

```
Private Sub RemoveChild(ByVal Child As LineItem)

    list.Remove(Child)

End Sub
```

Metoda nedělá nic světoborného: Jednoduše z kolekce odstraňuje specifikovaný odvozený objekt. Všimněte si, že metoda není opatřena žádnou klausulí **Handles**. Jak ale propojíme událost a metodu, tedy zpracovatele této události? Cílem k úspěchu je použití příkazu **AddHandler**. Tento příkaz je docela mocný, protože nám umožňuje dynamicky, tedy za běhu programu, realizovat vzájemné propojení mezi událostí a jejím zpracovatelem. Použití příkazu je užitečné zejména při vytváření tříd kolekcí, protože pomocí něj můžeme pracovat s událostmi odvozených objektů v okamžiku, kdy je požadovaný objekt přidán do kolekce.

Ve třídě kolekce již máme metodu **OnInsertComplete**, v níž dochází ke generování události **ListChanged**. Do těla této metody můžeme přidat kód příkazu **AddHandler** (tudíž bude jisté, že uvedený příkaz bude aktivován pro jakýkoliv nově přidaný odvozený objekt):

```
Protected Overrides Sub OnInsertComplete(ByVal index As Integer, _
        ByVal value As Object)
AddHandler CType(value, LineItem).RemoveMe, AddressOf RemoveChild
RaiseEvent ListChanged( _
        Me, New ListChangedEventArgs(ListChangedType.ItemAdded, index))

End Sub
```

Když bude nový odvozený objekt přidán do kolekce, vytvoříme propojení mezi událostí **RemoveMe** objektu a zpracovatelem události **RemoveChild**. Stiskne-li uživatel klávesu ESC při umístění kurzoru na řádku s nově přidanou položkou, tento odvozený objekt bude automaticky odstraněn z kolekce pomocí uvedeného mechanizmu.

IMPLEMENTACE ROZHRANÍ IDATAERRORINFO V ODVOZENÝCH TŘÍDÁCH

Na závěr si představíme implementaci rozhraní **IDataErrorInfo**, jehož pomocí můžeme vtisknout odvozeným třídám schopnost informovat instance ovládacího prvku **DataGrid** o skutečnosti, zdali je datová položka platná či nikoliv. Rozhraní budeme implementovat v naší odvozené třídě **LineItem**.

Rozhraní **IDataErrorInfo** definuje dvě metody, jak uvádí tab. 8.3.

Tab. 8.3

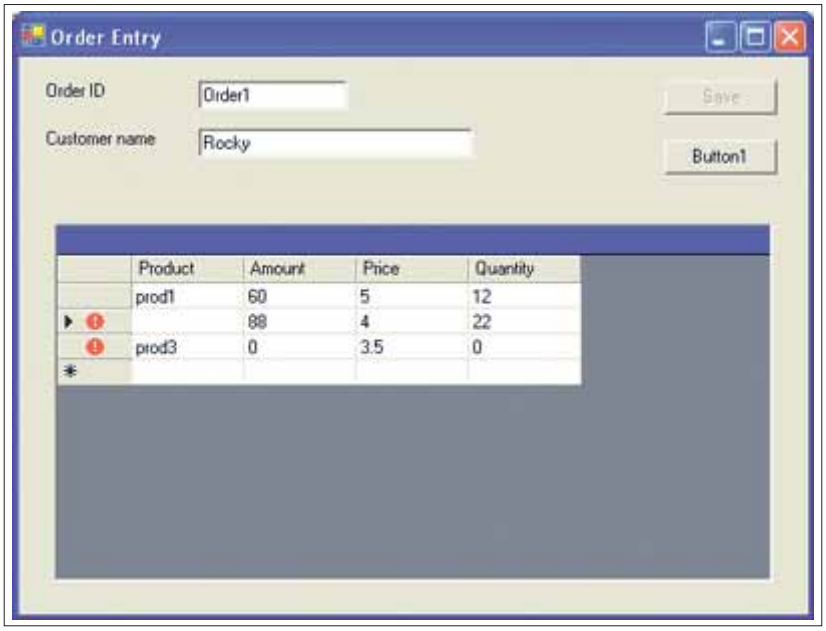
METODA	CHARAKTERISTIKA
Error	Vrací textovou zprávu, která charakterizuje problém s objektem (prázdný textový řetězec říká, že nedošlo k žádným problémům).
Item	Vrací textovou zprávu, která charakterizuje problém s určitou vlastností nebo datovým členem objektu (prázdný textový řetězec říká, že nedošlo k žádným problémům).

Klíčovou metodou je metoda **Error**. Prostřednictvím metody **Item** můžeme získat podrobnější informace, ovšem instance prvku **DataGrid** využívá text vrácený metodou **Error** pro analýzu, zdali je datová položka platná či nikoliv. Jak můžete vidět v programové ukázce, provádíme implementaci pouze metody **Error**:

```
Private ReadOnly Property [Error]() As String Implements _
        System.ComponentModel.IDataErrorInfo.Error
   Get
      If Len(Product) = 0 Then Return "Product name required"
      If Quantity <= 0 Then Return "Quantity must be greater than zero"
      Return ""
   End Get
End Property
```

Metoda ověřuje dvojici obchodních pravidel pro určitou datovou položku: Každá datová položka musí disponovat svým jménem a určením svého množství, jehož hodnota musí být větší než nula. Není-li kterákoliv z těchto podmínek splněna, metoda vrací textovou zprávu, která blíže popisuje charakter vzniklého problému. Na druhé straně, pokud je objekt platný, návratovou hodnotu metody tvoří prázdný textový řetězec.

Výsledkem je skutečnost, že instance prvku **DataGrid** graficky znázorňuje, které datové položky jsou platné, jak ukazuje obr. 8.5.



*Obr. 8.5: Instance prvku **DataGrid** graficky indikuje platnost datových položek*

Ještě jedna věc: Všimněte si, že vlastnost **Error** je definována jako soukromá (**Private**). Kdybychom ji udělali veřejně přístupnou (**Public**), textová zpráva o chybě by byla zobrazena v sloupci instance prvku **DataGrid**, takže uživatel by mohl jasně vidět, co se stalo s datovou položkou.

ZÁVĚR

Proces vytváření datových vazeb a propojení konečně dospěl. Implementace tohoto procesu ať už v prostředí Web Forms či Windows Forms je praktická a užitečná v mnoha různých scénářích. Jednou z nejvíce ceněných výhod je, že nyní můžeme vytvářet datové vazby s objekty a kolekcemi, nejenom s instancemi prvků typu **DataSet** a příbuznými objekty ADO.NET.

Tato kapitola vám ukázala, jak lze pomocí pár řádků programového kódu realizovat datové vazby mezi ovládacími prvky Windows Forms a obchodními objekty a třídami kolekcí. Již tedy není nutné, abychom i nadále zůstávali s jednoduchými technologiemi pro přístup k datům. Pro dnešní dobu je charakteristická příchuť objektově orientovaného programování, a to i při práci s daty, tak se nebojte tento styl programování využít!

9

C++ → C#: Co potřebujete vědět pro přechod z C++ k C#

Autor: Jesse Liberty, Microsoft Corporation

Abyste mohli plně využít informačního potenciálu této kapitoly, měli byste dobře znát programovací jazyk C++.
Úroveň vědomostní náročnosti: 3

http://msdn.microsoft.com/msdnmag/issues/0900/csharp/default.asp

SHRNUTÍ

Programovací jazyk C# staví na syntaxi a sémantice jazyka C++, přičemž dovoluje programátorům pracujícím s jazykem C objevit a získat výhody platformy .NET a společného běhového prostředí (Common Language Runtime, CLR). Ačkoliv je velmi pravděpodobné, že přechod z jazyka C++ do prostředí C# bude hladký a plynulý, přesto byste měli být obeznámeni s několika programovacími prvky, jejichž význam či syntaxe byly pozměněny (tyto změny se týkají kupříkladu operátoru **new**, struktur, konstruktorů a destruktorů). Tato kapitola prozkoumává také nově zavedené programovací prvky, s nimiž jazyk C# přichází. Za všechny vzpomeňme alespoň podporu automatické správy paměti, cyklus **foreach** či implementaci rozhraní. Když se obeznámíte s rozhraními, řeč přijde i na vlastnosti, pole a bázovou knihovnu tříd. Na závěr se podíváme na provádění asynchronních vstupně-výstupních operací, použití atributů a reflexe, dále probádáme sadu vestavěných typů a ukážeme si jejich použití.

Přibližně každých deset let musí vývojáři obětovat svůj drahocenný čas a energii pro získání pracovních zkušeností s novým programovacím prostředím. Na začátku osmdesátých let minulého staletí to byl jazyk C a operační systém Unix, zatímco počátkem devadesátých let se do popředí dostal systém Windows s C++. V dnešních dnech je hlavní proud tvořen všudypřítomnou vývojovou platformou Microsoft .NET Framework a moderním programovacím jazykem C#. Migrace směrem k novému programovacímu jazyku a prostředí je vždy časově náročná a vyžaduje notnou dávku dodatečného úsilí, ovšem v tomto případě jsou vynaložené náklady daleko převýšeny přínosy. Dobrou zprávou je, že analytická a návrhová fáze vyvíjených projektů mohou ve skutečnosti zůstat nezměněny, což znamená, že v tomto směru je použití jazyka C# a platformy .NET Framework takřka shodné s vývojovou sadou tvořenou Windows a C++. Zřetelné odlišnosti však existují při dosahování stanovených cílů za pomoci nového vývojového prostředí. V této kapitole si ukážeme, jak udělat z programátora v jazyce C++ vývojáře v C#.

Mnoho článků (kupříkladu *Sharp New Language: C# Offers the Power of C++ and Simplicity of Visual Basic, Nový jazyk: C# nabízí sílu C++ a jednoduchost Visual Basicu*) poukazuje na vylepšení, která programovací jazyk C# nabízí. Namísto opakování již napsaného se budeme soustředit na nejmarkantnější změny, které souvisejí s přechodem z jazyka C++ směrem k C#. Jednoduše řečeno, budeme si povídat o tom, jak vypadá migrační proces z neřízeného (unmanaged) prostředí do prostředí, které je spravováno společným běhovým prostředím (toto prostředí se označuje jako řízené (managed)). Rovněž vás budu varovat před spoustou pastí a pastiček, které číhají na neopatrného programátora v C++. Stranou však nezůstane ani popis nových prvků programovacího jazyka, které ovlivní styl psaní vašich aplikací.

PŘECHÁZÍME DO ŘÍZENÉHO PROSTŘEDÍ

Jazyk C++ byl navržen jako nízkoúrovňový, na platformě nezávislý, objektově orientovaný programovací jazyk. C# je navzdory tomu programovací jazyk vyšší úrovně, přičemž samozřejmě také splňuje náročná kritéria OOP. Přechod do řízeného prostředí vyžaduje, abyste poněkud změnili svůj styl nahlížení na programování. Jazyk C# vám dovoluje převzít do rukou přesnou kontrolu nad během kódu. Kromě toho, C# je velmi úzce spjat s vývojovou platformou .NET Framework, která vám podá pomocnou ruku při provádění takřka všech programových operací, od jejichž realizace bude práce vaší aplikace závislá. Sečteno a podtrženo, s pomocí moderní vývojové platformy a jazyka C# se můžete soustředit spíše na řešení problémů než na jejich specifickou implementaci.

Ukažme si malý příklad. S jazykem C++ jste měli ohromnou kontrolu nad vytvářením či rozvržením vašich objektů. Objekty jste mohli vytvářet na zásobníku, na hromadě, nebo dokonce na specifické paměťové adrese použitím operátoru umístění **new.**

Přechod do řízeného prostředí platformy .NET Framework znamená jistou ztrátu kontroly. Můžete si sice vybrat typ nově vytvářeného objektu, ovšem rozhodnutí o tom, kde bude tento objekt vytvořen je implicitní, a tedy plně v kompetenci společného běhového prostředí. Instance jednoduchých typů pro práci s celými či desetinnými čísly jsou vždy vytvářeny na zásobníku (pokud nejsou obsaženy uvnitř jiných objektů), zatímco instance tříd mají své stále místo na hromadě. Nemůžete kontrolovat, kde na hromadě je objekt vytvořen, nemůžete získat jeho adresu a nemůžete jej ani přesně umístit na určitou paměťovou adresu. (Existují sice cesty, jak obejít tato omezení, ovšem ty vás vedou na pomyslnou mez programování.)

Rovněž nemůžete skutečně ovládat dobu existence vašich objektů. Jazyk C# nepracuje se žádnými destruktory. Finalizace objektů je nedeterministická, což znamená, že uvolňování objektů z paměti počítače má na starosti automatický správce paměti, jenž provede úklid objektu ve chvíli, kdy tento není již nadále používán.

Již samotná programová struktura jazyka C# odráží principy vývojového rámce .NET Framework. To znamená, že jazyk C# nativně nepodporuje vícenásobnou dědičnost a také se zde nesetkáte se šablonami, protože implementace koncepce vícenásobné dědičnosti je v řízeném prostředí děsivě složitá. Běhové prostředí nedisponuje ani podporou techniky zvané generics.

Vestavěné primitivní datové typy jazyka C# nejsou ničím jiným, nežli referencí na systémové datové typy společného běhového prostředí. Například, typ **int** v C# je navázán na systémový typ **System.Int32**. Datové typy, které jsou dostupné v jazyce C#, nejsou definovány tímto jazykem, nýbrž standardními pravidly společného typového systému. Ve skutečnosti, pokud budete chtít v jazyce C# odvodit třídu z bázové třídy, která byla napsána ve Visual Basicu, budete muset využívat a uplatňovat jenom ta syntaktická pravidla, která upravuje společný typový systém a jazyková specifikace. Mimochodem, tato pravidla musejí dodržovat všechny programovací jazyky platformy .NET Framework.

Na druhé straně, řízené prostředí a společné běhové prostředí přinášejí velké množství hmatatelných pozitivních rysů. Vedle automatické správy paměti a jednotného typového systému, jenž implementují všechny .NET-kompatibilní programovací jazyky, získáváte především výtečně rozšířený a na komponentech založený programovací jazyk, který plně podporuje správu verzí a přístup k metadatům prostřednictvím mechanizmu reflexe. C# také podporuje pozdní vázání objektů, enumerace a vlastnosti, jakožto i události a delegáty (typově bezpečné funkční ukazatele).

Klíčovým prvkem řízeného prostředí je, koneckonců, vývojový rámec .NET Framework. Ačkoliv je pravda, že tento rámec je přístupný pro všechny .NET programovací jazyky, pouze jazyk C# byl navržen speciálně pro vývoj široké škály aplikací, které využívají bohatou sadu tříd, rozhraní a objektů.

PASTI, NÁSTRAHY A LÉČKY

Programovací jazyk C# je skutečně podobný jazyku C++, což sice vytváří dobré výchozí podmínky pro přechod z jazyka C++, ovšem na druhé straně se tato vlastnost může stát i zdrojem potenciálních nástrah na cestě směrem k C#. Když budete v jazyce C# psát stoprocentní kód jazyka C++, budete mít problémy s kompilací, nebo ještě hůř, kód nebude provádět to, co jste od něj očekávali. Většina sémantických změn mezi C++ a C# je triviálních (kupříkladu za deklarací třídy se nyní nepíše středník a funkce **Main** je nyní obdařena velkým začátečním písmenem). Naprostou většinu syntaktických chyb je kompilátor schopen odhalit velmi rychle, což znamená, že jim na tomto místě nemusíme věnovat více prostoru. Nicméně, je určitě potřebné poukázat na pár signifikantních změn, které mohou způsobit větší potíže.

ODKAZOVÉ A HODNOTOVÉ DATOVÉ TYPY

Jazyk C# rozlišuje mezi hodnotovými a odkazovými datovými typy. Primitivní typy (**int**, **long**, **double** atd.) a struktury představují hodnotové datové typy, zatímco třídy jsou odkazovými, nebo také referenčními typy. Instance hodnotových typů uchovávají své hodnoty na zásobníku, podobně jak to dělají i proměnné v C++, dokud nejsou přidruženy k instancím odkazových typů. U instancí referenčních datových typů rozlišujeme dvě entity, jimiž jsou odkazová neboli objektová proměnná a samotný objekt. Zatímco odkazová proměnná je umístěna na zásobníku a obsahuje adresu objektu, objekt samotný je uložen na řízené hromadě. Funkce odkazové proměnné je tedy dosti podobná ukazatelům v tradičním C++. Hodnoty instancí hodnotových typů jsou parametrům metod předávány hodnotou (prostřednictvím vytvoření kopie původních hodnot), zatímco hodnoty instancí odkazových typů, tedy reference, jsou předávány odkazem (je poskytnuta originální reference).

STRUKTURY

Struktury v C# jsou velice odlišné od těch, se kterými pracujete v jazyce C++. V C++ vystupují struktury jako třídy, kromě toho, že výchozí koncepce dědičnosti a přístupu k členům je veřejná a ne soukromá. Struktury v jazyce C# jsou od tříd zcela odlišné. Struktury v C# jsou navrženy pro zapouzdření objektů "lehké váhy". Struktury patří mezi hodnotové datové typy (ne odkazové typy), a tudíž jsou předávány hodnotou. Navíc, disponují jistými omezeními, kterým třídy nepodléhají. Kupříkladu, zdejší struktury jsou zapečetěné, což znamená, že není možné od struktur odvodit jiné struktury. Struktury také nemohou mít žádnou jinou bázovou třídu kromě třídy **System.ValueType**, která je odvozená od třídy **System.Object**. Struktury nemohou disponovat ani výchozím (bezparametrickým) konstruktorem.

Na druhé straně jsou však struktury efektivnější než třídy, protože je lze výhodně použít pro tvorbu nenáročných objektů. Jestliže vám nevadí, že struktury jsou zapečetěné a neodradí vás ani skutečnost, že jde o hodnotové datové typy, vězte, že struktury představují pro vytváření malých objektů lepší alternativu nežli třídy.

VŠE JE ODVOZENO OD BÁZOVÉ TŘÍDY SYSTEM.OBJECT

Vše, s čím se v jazyce C# střetnete, je odvozeno od mateřské třídy **System.Object**. To platí tak pro třídy, jako i instance hodnotových datových typů, jimiž jsou třeba struktury či instance typů pro úschovu celých čísel. Třída **System.Object** nabízí užitečné metody, mezi něž patří například metoda **ToString**. Tuto metodu můžete použít společně s metodou **System.Console.WriteLine**, která je funkčním ekvivalentem jazyka C# pro funkci **cout** známou z C++. Metoda **ToString** je přetížená a je schopna pojmout tak textové řetězce, jako i pole objektů.

Pokud budete používat metodu **WriteLine**, můžete pracovat i se substitučními parametry, podobně jak jste to dělali při psaní funkce **printf**. V tuto chvíli předpokládejme, že **myEmployee** je instancí uživatelsky definované třídy **Employee** a **myCounter** je instancí uživatelsky definované třídy **Counter**. Když zapíšete tento programový kód:

```
Console.WriteLine("The employee: {0}, the counter value: {1}",
                  myEmployee, myCounter);
```

Metoda **WriteLine** zavolá virtuální metodu **Object.ToString** pro každý objekt, a poté nahradí substituční parametry textovými řetězci, které představují návratové hodnoty této metody. Jestliže třída **Employee** nepřekryje metodu **ToString**, bude aktivována její výchozí implementace (odvozená z třídy **System.Object**), která vrátí jméno třídy v podobě textového řetězce. Třída **Counter** může překrýt metodu **ToString** tak, aby tato vracela celočíselnou hodnotu. Bude-li tomu tak, výstup by mohl vypadat následovně:

```
The employee: Employee, the counter value: 12
```

Co se stane, když metodě **WriteLine** předáte celá čísla? I když nemůžete zavolat metodu **ToString** na celočíselných hodnotách, pomůže vám kompilátor. Ten totiž implicitně vytvoří instanci třídy **System.Object**, do které vloží kopii celočíselné hodnoty (tato technika je známá pod názvem boxing). Když metoda **WriteLine** zavolá metodu **ToString**, objekt vrátí textovou reprezentaci celočíselní hodnoty, jak je ukázáno v níže uvedené programové ukázce s názvem "Using Classes".

Programová ukázka Using Classes

```
using System;

// A class which does not override ToString
public class Employee
{

}

// A Class which does override ToString
public class Counter
{
    private int theVal;

    public Counter(int theVal)
    {
        this.theVal = theVal;
    }

    public override string ToString()
    {
        Console.WriteLine("Calling Counter.ToString()");
        return theVal.ToString();
    }
}

public class Tester
{
   // Note that Main() has a capital M
  // and is a static member of the class
  public static void Main()
  {
      // create an instance of the class
      Tester t = new Tester();

      // call the non-static member
     // (must be through an instance)
     t.Run();
  }

 // the non-static method which demonstrates
 // calling ToString and boxing
 public void Run()
 {
    Employee myEmployee = new Employee();
    Counter myCounter = new Counter(12);
    Console.WriteLine("The employee: {0}, the counter value: {1}",
                                myEmployee, myCounter);

    // note that integer literals and variables are boxed
    int myInt = 5;
    Console.WriteLine("Here are two integers: {0} and {1}", 17, myInt);
 }
}
```

REFERENČNÍ A VÝSTUPNÍ PARAMETRY

V jazyce C#, podobně jako v C++, může mít metoda pouze jednu návratovou hodnotu. Toto omezení můžete v C++ obejít pomocí předávání ukazatelů nebo referencí parametrům metody. Volaná metoda může změnit hodnotu parametrů a nově upravené hodnoty jsou přístupné i pro volající metodu.

Předáte-li metodě referenci, získáte přístup k originálnímu objektu podobně, jako když předáte referenci nebo ukazatel v kódu jazyka C++. Nicméně, popsaná technika nefunguje s instancemi hodnotových typů. Budete-li chtít předat hodnotu instance hodnotového typu odkazem, musíte před příslušný parametr umístit klíčové slovo **ref**.

```
public void GetStats(ref int age, ref int ID, ref int yearsServed)
```

Dobře si zapamatujte, že klíčové slovo **ref** musíte použít nejenom v deklaraci metody, nýbrž také při jejím volání.

```
Fred.GetStats(ref age, ref ID, ref yearsServed);
```

Nyní můžete deklarovat hodnoty pro parametry **age**, **ID** a **yearsServed** ve volající metodě, poté je předat metodě volané a následně získat upravené hodnoty.

C# vyžaduje jednoznačné přiřazení, což znamená, že lokální proměnné **age**, **ID** a **yearsServed** musejí být inicializovány ještě předtím, než dojde k volání metody **GetStats**. Takovýto přístup je zbytečně komplikovaný, protože tyto proměnné používáte jenom pro uchování upravených hodnot, které vzejdou z metody **GetStats**. Naštěstí, tento problém lze velice jednoduše vyřešit. Jazyk C# poskytuje klíčové slovo **out**, které můžete použít pro předávání neinicializovaných proměnných odkazem. Pomocí klíčového slova **out** můžete přesně realizovat své záměry:

```
public void GetStats(out int age, out int ID, out int yearsServed)
```

Také klíčové slovo **out** musí být použito i při samotném volání metody.

```
Fred.GetStats(out age,out ID, out yearsServed);
```

POUŽITÍ KLÍČOVÉHO SLOVA NEW

V C++ jste pomocí klíčového slova **new** uskutečňovali vytváření objektů na hromadě. V jazyce C# existují v tomto směru značné rozdíly. Pokud použijete klíčové slovo **new** ve spojení s odkazovými typy, vytvoříte objekty na hromadě, ovšem v kombinaci s hodnotovými typy jsou objekty vytvořeny na zásobníku a jsou aktivovány jejich konstruktory.

Ve skutečnosti můžete vytvořit i instanci struktury na zásobníku, ovšem buďte opatrní! Použijete-li klíčové slovo **new**, bude vytvořený objekt inicializován. Pokud ovšem vytvoříte instanci bez něj, budete muset všechny členy struktury inicializovat sami, a to vše ještě předtím, než tuto instanci použijete (předtím, než ji předáte metodě). Nebudou-li splněny tyto podmínky, kompilátor vás na vzniklé chyby upozorní sám. Takže ještě jednou: Jednoznačné přiřazení vyžaduje, aby byl každý objekt řádně inicializován (viz podkapitolu *Inicializace objektů* dále v tomto textu).

Inicializace objektů

```
using System;

 // a simple structure with two
// member variables and a constructor
public struct Point
 {
     public Point(int x, int y)
     {
          this.x = x;
          this.y = y;
     }
```

```
     public int x;
     public int y;
 }

public class Tester
 {
     public static void Main()
     {
          Tester t = new Tester();
          t.Run();
     }

     public void Run()
     {
          Point p1 = new Point(5,12);
          SomeMethod(p1); // fine

          Point p2; // create without calling new

          // this won't compile because the member
          // variables of p2 have not been initialized
          // SomeMethod(p2);

          // initialize them by hand
          p2.x = 1;
          p2.y = 2;

          SomeMethod(p2); // fine

}

// a method we can pass the Point to
private void SomeMethod(Point p)
{
    Console.WriteLine("Point at {0} x {1}",
       p.x, p.y);
 }

}
```

VLASTNOSTI

Většina programátorů v C++ se snaží deklarovat datové členy tříd jako soukromé. Tato forma skrývání dat podporuje zapouzdření a dovoluje vám také upravit implementaci třídy, aniž byste porušili rozhraní, na které spoléhají vaši klienti. Za běžných okolností chcete, aby měli uživatelé vašich tříd možnost obdržet a potažmo i pozměnit hodnoty specifikovaných datových členů. Programátoři v C++ proto vytvářejí metody, jejichž práce spočívá v přístupu a modifikaci hodnot soukromých členských proměnných.

V prostředí jazyka C# jsou vlastnosti prvotřídní součástí třídy. Je to celkem zajímavé, protože zatímco pro tvůrce třídy vystupují vlastnosti jako metody, klienti s nimi pracují jako s členskými proměnnými. Takovéto uspořádání je perfektní, protože můžete svým klientům umožnit přístup k datových členům třídy s klidným svědomím, že jste dodrželi principy zapouzdření a skrývání dat.

Pokud budete chtít začlenit do svých tříd vlastnosti, můžete postupovat tak, jako v následující programové ukázce. Zde vidíte třídu **Employee** s vlastností **Age**, jejíž pomocí lze přistupovat a případně také modifikovat hodnotu datového členu třídy s názvem **age**.

```
public int Age
{
     get
     {
         return age;
     }
     set
     {
         age = value;
     }
 }
```

Klíčové slovo **value** slouží pro potřeby vlastnosti a můžete jej využívat implicitně. Když napíšete kód

```
Fred.Age = 17;
```

kompilátor uloží hodnotu 17 do datového členu třídy.

Pokud chcete vytvořit vlastnost určenou pouze ke čtení, vynechejte blok **set**. Tuto eventualitu demonstruje implementace vlastnosti **YearsServed**:

```
public int YearsServed
{
   get
   {
       return yearsServed;
   }
}
```

Použití bloků get a set

```
private void Run()
{
    Employee Fred = new Employee(25,101,7);
    Console.WriteLine("Fred's age: {0}",
           Fred.Age);
     Fred.Age = 55;
     Console.WriteLine("Fred's age: {0}",
           Fred.Age);
     Console.WriteLine("Fred's service: {0}",
           Fred.YearsServed);
     // Fred.YearsServed = 12; // not allowed!
}
```

Hodnotu Fredova věku získáte prostřednictvím vlastnosti **Age**. Tuto vlastnost můžete rovněž použít pro přiřazení nové hodnoty do datového členu **age**. Podobným způsobem můžete vyhledat i hodnotu vlastnosti **YearsServed**, ovšem tuto již nemůžete měnit. Zrušíte-li okomentování posledního řádku kódu, kompilátor nebude schopen kód přeložit.

Rozhodnete-li se později získat hodnotu pro datový člen **age** z databáze, budete muset upravit implementaci bloku **get** vlastnosti **Age**. Změny, které provedete, však žádným způsobem neovlivní práci klientů vaší třídy.

POLE

Pro uskutečňování operací s poli nabízí jazyk C# speciální třídu, se kterou je práce s poli inteligentnější, než je tomu v C++. Kupříkladu, není možné zapisovat nové hodnoty mimo hraničních mezí pole. Navíc, třída **Array** má ještě chytřejšího bratránka s názvem **ArrayList**, jehož pomocí lze vytvářet pole, která dovedou dynamicky měnit svoji velikost podle potřeb programu.

V jazyce C# se setkáte se třemi typy polí: jednorozměrná pole, vícerozměrná pravidelná pole (podobná vícerozměrným polí z C++) a vícerozměrné nepravidelná pole (pole polí).

Jednorozměrné pole lze vytvořit následovně:

```
int[] myIntArray = new int[5];
```

Vytvořené pole můžete inicializovat takto:

```
int[] myIntArray = { 2, 4, 6, 8, 10 };
```

Pole 4x3, tedy pole o čtyřech řádcích a třech sloupcích, můžete vytvořit pomocí tohoto programového kódu:

```
int[,] myRectangularArray = new int[rows, columns];
```

Pole můžete inicializovat i takto:

```
int[,] myRectangularArray =
{
    {0,1,2}, {3,4,5}, {6,7,8}, {9,10,11}
};
```

Při vytváření vícerozměrných nepravidelných polí stačí, když určíte pouze jednu dimenzi pole:

```
int[][] myJaggedArray = new int[4][];
```

A poté vytvoříte každé z vnitřních polí, asi takto:

```
myJaggedArray[0] = new int[5];
myJaggedArray[1] = new int[2];
myJaggedArray[2] = new int[3];
myJaggedArray[3] = new int[5];
```

Vzhledem k tomu, že pole jsou odvozena ze třídy **System.Array**, nabízejí množství užitečných metod, kromě jiných také metody **Sort** a **Reverse**.

INDEXER

V C# lze vytvářet sady objektů, s nimiž můžete pracovat jako s poli. Představte si, že byste chtěli vytvořit seznam, jenž by byl naplněn textovými řetězci, které by byly posléze zobrazovány. Bylo by hezké, kdybychom mohli k jednotlivým položkám seznamu přistupovat pomocí indexu jako při práci s poli.

```
string theFirstString = myListBox[0];
string theLastString = myListBox[Length-1];
```

Tento styl práce nám umožňuje programová konstrukce, které se říká **indexer**. Indexer se podobá na vlastnost, ale podporuje syntaxi operátoru index.

Operátor index

```
public string this[int index]
{
      get
      {
           if (index < 0 || index >= myStrings.Length)
           {
              // handle bad index
           }
           return myStrings[index];
      }
      set
      {
           myStrings[index] = value;
      }
}
```

Programový kód ve třídě **ListBox** ukazuje, jak implementovat velice jednoduchou třídu **ListBox** s podporou indexování.

Třída ListBox

```
using System;

       // a simplified ListBox control
public class ListBoxTest
{
     // initialize the list box with strings
     public ListBoxTest(params string[] initialStrings)
     {
          // allocate space for the strings
          myStrings = new String[256];

          // copy the strings passed in to the constructor
         foreach (string s in initialStrings)
         {
              myStrings[myCtr++] = s;
         }
 }

 // add a single string to the end of the list box
 public void Add(string theString)
 {
      myStrings[myCtr++] = theString;
 }
```

```
 // allow array-like access
 public string this[int index]
 {
     get
     {
          if (index < 0 || index >= myStrings.Length)
          {
              // handle bad index
          }
          return myStrings[index];
     }
     set
    {
        myStrings[index] = value;
     }
 }

 // publish how many strings you hold
 public int GetNumEntries()
 {
     return myCtr;
 }

   private string[] myStrings;
   private int myCtr = 0;
}

public class Tester
{
     static void Main()
     {
          // create a new list box and initialize
          ListBoxTest lbt = new ListBoxTest("Hello", "World");

          // add a few strings
          lbt.Add("Who");
          lbt.Add("Is");
          lbt.Add("John");
          lbt.Add("Galt");
          // test the access
          string subst = "Universe";
          lbt[1] = subst;

          // access all the strings
          for (int i = 0;i<lbt.GetNumEntries();i++)
          {
              Console.WriteLine("lbt[{0}]: {1}",i,lbt[i]);
          }
     }
 }
```

ROZHRANÍ

Pojem rozhraní ve světě softwaru ztělesňuje dohoda, podle níž bude realizována interakce mezi dvěma typy. Když typ vystaví své rozhraní, říká kterémukoliv potenciálnímu klientovi: "Garantuji podporu následujících metod, vlastností, událostí a indexerů".

C# je objektově orientovaný programovací jazyk, což znamená, že tyto dohody jsou zapouzdřeny do entit, jimž se říká rozhraní. Klíčové slovo **interface** deklaruje odkazový typ, jenž zapouzdřuje jistou dohodu.

Z pojmového hlediska bychom mohli rozhraní přirovnat k abstraktní třídě. Odlišností je, že abstraktní třída slouží jako bázová třída pro rodinu všech podtříd, zatímco rozhraní lze míchat s vybranými třídami v hierarchii dědičnosti.

ROZHRANÍ IENUMERABLE

Když se vrátíme k předcházející programové ukázce, můžeme konstatovat, že by bylo docela příjemné, kdybychom byli schopni vytisknout textové řetězce ze třídy **ListBoxText** pomocí cyklu **foreach**. Tento programový rys můžeme přidat tak, že budeme implementovat rozhraní **IEnumerable**, které lze implicitně použít pro konstrukce typu **foreach**. Rozhraní **IEnumerable** lze implementovat v každé třídě, ve které je potřebné použít funkcionalitu cyklů **foreach** či enumerací.

Rozhraní **IEnumerable** definuje jedinou metodu s názvem **GetEnumerator**, která má na starosti vracení specifické implementace tohoto rozhraní.

Třída s názvem **Enumerator** musí implementovat metody rozhraní **IEnumerator**. Implementace metod může být provedena přímo třídou kontejneru, nebo separátní třídou. Posledně zmíněný přístup je ve všeobecnosti upřednostňován, protože umožňuje zapouzdření odpovědnosti do třídy **Enumerator**.

Třídu **Enumerator** přidáme do třídy **ListBoxTest**, kterou jste mohli vidět ve výše uvedené programové ukázce. Jelikož je třída **Enumerator** specifická pro naši kontejnerovou třídu (třída **ListBoxEnumerator** musí disponovat informacemi o třídě **ListBoxTest**), provedeme privátní implementaci, která bude obsažena uvnitř třídy **ListBoxText**.

V této verzi je třída **ListBoxTest** definována tak, aby mohla implementovat rozhraní **IEnumerable**. Toto rozhraní musí vracet **Enumerator**.

```
public IEnumerator GetEnumerator()
{
    return (IEnumerator) new ListBoxEnumerator(this);
}
```

Všimněte si, že metoda předává enumerátoru aktuální objekt třídy **ListBoxTest** (this). Tak bude moci enumerátor provést enumeraci požadovaného objektu třídy **ListBoxTest**.

Třída, která implementuje **Enumerator** se jmenuje **ListBoxEnumerator** a jde o soukromou třídu, která je definována uvnitř třídy **ListBoxTest**.

Instance třídy **ListBoxTest**, která má být podrobena enumeraci, je předána konstruktoru ve formě argumentu a posléze je odkaz na ní přiřazen do členské proměnné **myLBT**. V těle konstruktoru rovněž dochází k uložení hodnoty -1 do členské proměnné, čímž je poukázáno na skutečnost, že proces enumerace ještě nebyl zahájen.

```
public ListBoxEnumerator(ListBoxTest theLB)
{
    myLBT = theLB;
    index = -1;
}
```

Následující metoda **MoveNext** uskutečňuje inkrementaci indexu a kontrolu, zdali nebyla překročena mez objektu, jenž je právě analyzován. Pokud byla mez překročena, je vrácena hodnota **false**, v opačném případě je to hodnota **true**.

```
public bool MoveNext()
{
    index++;
    if (index >= myLBT.myStrings.Length)
       return false;
    else
       return true;
}
```

Pro získání posledně přidaného textového řetězce je implementovaná vlastnost **Current**. Tohle rozhodnutí je svévolné. V jiných třídách by vlastnost **Current** měla význam, jenž by jí přisoudil samotný programátor dle svých potřeb. Ovšem v tomto případě je výslovně definováno, že každý enumerátor musí být s to vrátit aktuální člen, jelikož práce enumerátorů spočívá právě v schopnosti získat aktuální členy.

```
public object Current
{
    get
    {
        return(myLBT[index]);
    }
}
```

A to je vše. Volání cyklu **foreach** přivádí enumerátor a používá jej pro enumeraci prvků pole. Jelikož cyklus **foreach** bude zobrazovat každý textový řetězec, bez ohledu na to, zdali byla přidána smysluplná hodnota, pozměnili jsme inicializaci proměnné **myStrings**.

```
myStrings = new String[8];
```

POUŽITÍ BÁZOVÉ KNIHOVNY TŘÍD

Abyste získali lepší představu o tom, jak se jazyk C# liší od jazyka C++ a jak vyřešit problémy související s přechodem do nového prostředí, prozkoumáme méně triviální ukázku. Sestrojíme třídu, která bude sloužit pro čtení rozsáhlého textového souboru, jehož obsah bude zobrazovat na obrazovce počítače. Náš program bude pracovat s více vlákny, což znamená, že během čtení dat ze souboru budeme moci dále pracovat s aplikací.

S pomocí C++ byste pravděpodobně postupovali tak, že byste vytvořili jedno vlákno pro čtení dat ze souboru a další vlákno pro uskutečňování dalších úkonů. Ačkoli by tato vlákna měla pracovat nezávisle, bude asi nutné začlenit podporu synchronizace provádění programových operací. Všechny nastíněné úkoly můžete realizovat i v jazyce C#, ovšem s tím rozdílem, že většinu svého času nebudete muset věnovat psaní kódu pro vytvoření vláken, protože platforma .NET Framework nabízí velice mocný mechanizmus pro asynchronní realizaci vstupně-výstupných operací.

Asynchronní podpora provádění vstupně-výstupných operací je zabudována přímo do samotného společného běhového prostředí (CLR) a její použití je takřka stejně jednoduché, jako je tomu při programování standardních tříd pro práci s proudy znaků. Nejprve dáme kompilátoru vědět, že hodláme využít dovedností objektů z několika jmenných prostorů:

```
using System;
using System.IO;
using System.Text;
```

Když do projektu začleníme odkaz na jmenný prostor **System**, neznamená to, že se automaticky vytvoří odkazy i na vnořené jmenné prostory. Odkaz na každý jmenný prostor nižší hierarchie proto musíme explicitně importovat pomocí direktivy **using**. Jak můžete vidět, budeme potřebovat ještě další dva jmenné prostory: **System.IO** pro realizaci vstupně-výstupných operací a **System.Text** pro podporu kódování ASCII znaků bajtového proudu.

Kroky, které musíte uskutečnit pro napsání programu, jsou až překvapivě jednoduché. Je to způsobeno tím, že platforma .NET Framework provede většinu práce za vás. Pro uskutečnění asynchronních vstupně-výstupních operací použijeme metodu **BeginRead** třídy **Stream**. Tato metoda načte data do vyrovnávací paměti a poté, ve chvíli, kdy jsou načtená data připravena k zpracování, aktivuje metodu zpětného volání (callback).

Do vyrovnávací paměti bude zapotřebí uložit pole bajtů a kromě toho musíme vytvořit delegáta pro metodu zpětného volání. Za tímto účelem budeme uvnitř třídy **AsynchIOTester** deklarovat dvě privátní členské proměnné:

```
public class AsynchIOTester
{
    private Stream inputStream;
    private byte[] buffer;
    private AsyncCallback myCallBack;
```

Členská proměnná **inputStream** je typu **Stream**, z čehož plyne, že do této proměnné bude možné uložit odkaz na instanci třídy **Stream**. Tato instance disponuje metodou **BeginRead**, které je nutné předat vyrovnávací paměť (**buffer**) a také delegáta (**myCallback**). Delegáta si můžete představit jako typově bezpečný funkční ukazatel. Jazyk C# disponuje vestavěnou podporou delegátů, takže jejich použití je flexibilní a intuitivní současně. Prostřednictvím delegáta bude možné zavolat delegovanou metodu ve chvíli, kdy budou načtena všechna data ze souboru na disku, tedy v okamžiku, kdy budou data připravena ke zpracování. Během čekání na načtení dat můžeme uskutečňovat další programové operace (v našem případě provádíme cyklus, v němž dochází k inkrementaci celočíselné proměnné od 1 do 50 000, ovšem v reálné aplikaci byste měli dát přednost poněkud smysluplnějším činnostem).

Náš delegát je deklarován pomocí delegáta typu **AsyncCallback**, což je přesně to, co očekává metoda **BeginRead** třídy **Stream**. Delegát typu **AsyncCallback** je deklarován ve jmenném prostoru **System** následovně:

```
public delegate void AsyncCallback (IAsyncResult ar);
```

Z deklarace můžeme vyvodit, že náš delegát může být asociován s jakoukoliv metodou, která nevrací žádnou hodnotu (**void**) a pracuje s parametrem, jenž přijímá odkaz na instanci, která implementuje rozhraní **IAsyncResult**. Když je za běhu programu zavolaná metoda, společné běhové prostředí zabezpečí předání správného odkazu, ovšem vy musíte deklarovat metodu:

```
void OnCompletedRead(IAsyncResult asyncResult)
```

V konstruktoru třídy **AsynchIOTester** provedeme spojení delegáta:

```
AsynchIOTester()
{
    •••
    myCallBack = new AsyncCallback(this.OnCompletedRead);
}
```

Do členské proměnné **myCallback** (která je typu **AsyncCallback** a byla deklarována výše) je přiřazen odkaz na instanci delegáta, která byla vytvořena zavoláním konstruktoru delegáta **AsyncCallback** a předáním metody, kterou chcete asociovat s delegátem.

Nyní se podívejme, jak celý program pracuje krok za krokem. V metodě **Main** vytváříme instanci třídy **AsynchIOTester** a pomocí metody **Run** ji nařizujeme, aby se spustila:

```
public static void Main()
{
   AsynchIOTester theApp = new AsynchIOTester();
   theApp.Run();
}
```

Aplikace operátoru **new** způsobí aktivaci konstruktoru třídy. V konstruktoru dochází k otevření souboru pomocí metody **OpenRead**, která po splnění svého záměru vrací objekt třídy **Stream**. V dalším kroku alokujeme prostor pro vyrovnávací paměť a aktivujeme mechanizmus zpětného volání.

```
AsynchIOTester()
{
   inputStream = File.OpenRead(@"C:\MSDN\fromCppToCS.txt");
   buffer = new byte[BUFFER_SIZE];
   myCallBack = new AsyncCallback(this.OnCompletedRead);
}
```

Abychom začali s asynchronním čtením dat specifikovaného souboru, zavoláme v metodě **Run** metodu **BeginRead**.

```
inputStream.BeginRead(
   buffer,             // where to put the results
   0,                  // offset
   buffer.Length,      // how many bytes (BUFFER_SIZE)
   myCallBack,         // call back delegate
   null);              // local state object
```

Poté zahajujeme výčet iterací cyklu **for**:

```
for (long i = 0; i < 50000; i++)
{
   if (i%1000 == 0)
   {
       Console.WriteLine("i: {0}", i);
   }
}
```

Když je čtení dat ze souboru u konce, společné běhové prostředí aktivuje metodu zpětného volání.

```
void OnCompletedRead(IAsyncResult asyncResult)
{
```

První věcí, kterou musíme v metodě **OnCompleteRead** udělat, je zjistit, kolik bajtů již bylo přečteno. Toho docílíme aktivací metody **EndRead** instance třídy **Stream**, které předáme odkaz na objekt rozhraní **IAsyncResult**, jenž byl vrácen společným běhovým prostředím.

```
int bytesRead = inputStream.EndRead(asyncResult);
```

Výsledkem volání metody **EndRead** je počet přečtených bajtů souboru. Je-li zjištěný počet větší než nula, převedeme obsah vyrovnávací paměti do podoby textového řetězce, který posléze zobrazíme v příkazovém řádku. Poté opětovně zavoláme metodu **BeginRead** pro realizaci jiné asynchronní čtecí operace.

```
if (bytesRead > 0)
{
    String s = Encoding.ASCII.GetString(buffer, 0, bytesRead);
    Console.WriteLine(s);
    inputStream.BeginRead(buffer, 0, buffer.Length,
                          myCallBack, null);
}
```

Nyní můžeme během čtení dat provádět jinou programovou činnost (v tomto případě jde o počítání do padesát tisíc). Když je čtení dat ze souboru u konce, můžeme na tuto skutečnost patřičně reagovat (v tomto případě zobrazujeme data v příkazovém řádku). Kompletní zdrojový kód této programové ukázky je uložen v souboru **AsynchIO.cs**, jenž si můžete stáhnout z internetové adresy, která je uvedena na začátku této kapitoly.

Všechny parciální aktivity, které souvisejí s realizací asynchronně prováděných vstupně-výstupných operací, jsou zabezpečovány prostřednictvím společného běhového prostředí. Ještě zajímavější je však čtení dat přes síť, jak se dozvíte v následující podkapitole.

ČTENÍ DAT SOUBORU PŘES SÍŤ

Čtení dat souboru přes síť je v jazyce C++ poměrně náročné programátorské cvičení. Vývojová platforma .NET Framework ovšem nabízí i pro tuto netriviální činnost dokonalou podporu. Ve skutečnosti spočívá čtení dat souboru přes síť pouze v jiném použití třídy **Stream** z bázové knihovny tříd.

Začněme vytvořením instance třídy **TCPListener**, která bude naslouchat na TCP/IP portu (portu 65000 v našem případě).

```
TCPListener tcpListener = new TCPListener(65000);
```

Když je objekt třídy **TCPListener** zrozen, můžeme jej požádat, aby začal naslouchat.

```
tcpListener.Start();
```

Nyní počkáme na žádost klienta o spojení.

```
Socket socketForClient = tcpListener.Accept();
```

Metoda **Accept** objektu třídy **TCPListener** vrací objekt třídy **Socket**, jenž reprezentuje rozhraní soketu Berkeley a který je vázaný na přesně stanovený koncový bod (jímž je v tomto případě klient). Metoda **Accept** je synchronní metoda, takže svoji návratovou hodnotou nevrátí dříve, než obdrží žádost o spojení. Jestliže je soket připojen, je možné začít s procesem zasílání souboru klientovi.

```
if (socketForClient.Connected)
{
• • •
```

Dále musíme vytvořit objekt třídy **NetworkStream**, přičemž instančnímu konstruktoru předáme soket.

```
NetworkStream networkStream = new NetworkStream(socketForClient);
```

Pokračujeme tvorbou objektu třídy **StreamWriter**, podobně jak jsme to dělali před chvílí, jenom s tím rozdílem, že tentokrát předáme konstruktoru instanci třídy **NetworkStream**, kterou jsme právě vytvořili.

```
System.IO.StreamWriter streamWriter =
   new System.IO.StreamWriter(networkStream);
```

Data, která budou zapsána do tohoto proudu, budou poslána klientovi prostřednictvím sítě. Kompletní zdrojový kód můžete nalézt v souboru **TCPServer.cs**, jenž je k dispozici na Internetu.

VYTVÁŘENÍ KLIENTA

Klient provádí zrození instance třídy **TCPClient**, která představuje spojení s hostitelem dle protokolu TCP/IP.

```
TCPClient socketForServer;
socketForServer = new TCPClient("localHost", 65000);
```

S instancí třídy **TCPClient** můžeme vytvořit objekt třídy **NetworkStream**, jenž posléze použijeme pro vytvoření objektu třídy **StreamReader**.

```
NetworkStream networkStream = socketForServer.GetStream();
System.IO.StreamReader streamReader =
   new System.IO.StreamReader(networkStream);
```

Nyní budeme číst data z proudu, dokud nebude načten celý segment dat, který poté zobrazíme v příkazovém řádku.

```
do
{
     outputString = streamReader.ReadLine();

     if( outputString != null )
     {
          Console.WriteLine(outputString);
     }
}
while( outputString != null );
```

Abychom otestovali zabudovanou funkcionalitu, vytvoříme jednoduchý testový soubor s několika řádky:

```
This is line one
This is line two
This is line three
This is line four
```

Výstup ze serveru vypadá takto:

```
Output (Server)
Client connected
Sending This is line one
Sending This is line two
Sending This is line three
Sending This is line four
Disconnecting from client...
Exiting...
```

A zde je výstup klienta:

```
This is line one
This is line two
This is line three
This is line four
```

ATRIBUTY A METADATA

Jednou z významných odlišností mezi jazyky C# a C++ je to, že C# nabízí vývojářům neodmyslitelnou podporu práce s metadaty. Metadata jsou data, která popisují vaše třídy, objekty, metody atd. Atributy jsou dodávány ve dvou příchutích: První tvoří atributy, které tvoří standardní součást společného běhového prostředí (CLR). Do druhé skupiny pak můžeme zařadit uživatelsky definované atributy, které si můžete vytvořit dle vašich momentálních potřeb. Atributy společného běhového prostředí se používají pro podporu serializace, marshalingu a COM interoperability. Budete-li věnovat chvilku pro prozkoumání této skupiny atributů, zjistíte, že jich existuje docela dost. Jak se můžete dozvědět, některé z atributů se aplikují na assembly, zatímco jiné si více rozumí s třídami či rozhraními. Těmto atributům se říká atributy zaměřené na cíl.

Již z názvu je zřejmé, že tato skupina atributů bude aplikována na určitý cíl. Aplikaci atributu na cíl provedete tak, že požadovaný atribut umístíte do hranatých závorek ihned před cíl, na který je daný atribut zaměřen. Atributy lze také kombinovat, a to buď ve formě zásobníku, kdy se jeden atribut nachází nad jiným, nebo pomocí čárek. Níže můžete vidět obě možnosti kombinace atributů.
Kombinace atributů ve formě zásobníku:

```
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile(".\\keyFile.snk")]
```

Kombinace atributů pomocí čárky:

```
[assembly: AssemblyDelaySign(false),
   assembly: AssemblyKeyFile(".\\keyFile.snk")]
```

UŽIVATELSKÉ ATRIBUTY

Nic vám nebrání v tom, abyste vytvářeli své vlastní atributy a používali je za běhu programu. Můžete třeba vytvořit dokumentační atribut, jehož pomocí byste mohli označovat sekce programového kódu s adresou URL, na níž by se nacházela příslušná dokumentace. Nebo byste mohli obohacovat váš kód revizními komentáři, či komentáři s informacemi o opravených chybách.

Představte si, že vaše vývojářská společnost by ráda získala přehled o opravených programových chybách. Ačkoliv je vedena interní databáze softwarových chyb, rádi byste svázali zprávy o chybách se specifickými opravami těchto chyb v programovém kódu. Povězme, že vaše komentáře by mohly mít následující podobu:

```
// Bug 323 fixed by Jesse Liberty 1/1/2005.
```

Takovéto komentáře by se mohly v kódu docela hezky vyjímat, ovšem ještě lepší by bylo, kdybyste mohli informace z komentářů použít při sestavování zpráv, nebo je ukládat do databáze pro pozdější použití. Bylo by příjemné, kdyby aplikované chybové notace používaly ve všech zprávách stejnou syntaxi. S uživatelským atributem můžete vyřešit nastíněnou situaci poměrně snadno. Stávající strukturu komentáře byste poté mohli nahradit tímto ekvivalentem:

```
[BugFix(323,"Jesse Liberty","1/1/2005") Comment="Off by one error"]
```

Atributy, podobně jako většina entit v jazyce C#, jsou ztělesňovány třídami. Abyste vytvořili nový atribut, musíte vytvořit třídu uživatelského atributu, kterou odvodíte od bázové třídy **System.Attribute**.

```
public class BugFixAttribute : System.Attribute
```

Dále musíte sdělit kompilátoru skutečnost, s jakými typy elementů bude moci být váš uživatelský atribut použit (jde tedy o determinaci cíle zaměření atributu). Na zprostředkování této informace nepoužijete nic jiného, než další atribut. Zajímavé, že?

```
[AttributeUsage(AttributeTargets.ClassMembers, AllowMultiple = true)]
```

Atribut **AttributeUsage** je metaatribut, tedy atribut, jenž je aplikován na samotné atributy. Tento atribut poskytuje meta-metadata, jinými slovy, data o metadatech. V tomto případě jsou předány dva argumenty: Prvním je cíl zaměření atributu (v našem případě jde o členy třídy), zatímco druhý představuje příznak, jenž říká, zdali je možné, aby byl jistému elementu přiřazen více než jeden atribut daného typu. Argument **AllowMultiple** byl nastaven na hodnotu **true**, což znamená, že jednotlivým členům třídy bude možné přiřadit více než jeden atribut **BugFixAttribute**.

Chcete-li použít více cílů zaměření atributu, můžete pro jejich sloučení použít logický operátor OR (|).

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface,
   AllowMultiple = true)]
```

Uvedený atribut bude možné připojit ku třídě nebo k rozhraní.

Náš nový uživatelský atribut se jmenuje **BugFixedAttribute**. Konvence říká, že za samotný název atributu byste měli umístit ještě textový identifikátor **Attribute**, abyste vy, nebo vaši kolegové okamžitě věděli, že pracujete s atributem. Kompilátor ovšem umožňuje volání atributu i ve zkrácené formě, která pozůstává pouze ze jména atributu a ne konvenčního sufixu. Takže atribut můžete aplikovat i takto:

```
[BugFix(123, "Jesse Liberty", "01/01/05", Comment="Off by one")]
```

V tomto případě bude kompilátor nejprve hledat atribut se jménem **BugFix**, a když jej nenalezne, vyhledá atribut s názvem **BugFixAttribute**.

Každý atribut musí disponovat přinejmenším jedním konstruktorem. Atributy dokáží pracovat se dvěma typy parametrů: pozičními a pojmenovanými. Kupříkladu, v předcházející ukázce byly informace o identifikačním čísle chyby, jménu programátora a datu představovány pozičními parametry, zatímco komentář byl uložen v pojmenovaném parametru. Poziční parametry jsou předávány prostřednictvím konstruktoru a musejí být předávány přesně v pořadí, v kterém jsou deklarovány.

```
public BugFixAttribute(int bugID, string programmer, string date)
{
    this.bugID = bugID;
    this.programmer = programmer;
    this.date = date;
}
```

Pojmenované parametry jsou implementovány v podobě vlastností.

POUŽITÍ ATRIBUTU

Abychom mohli atribut otestovat, vytvoříme jednoduchou třídu s názvem **MyMath**, do které vložíme dvě funkce. Poté třídě přiřadíme vytvořený uživatelský atribut.

```
[BugFixAttribute(121,"Jesse Liberty","01/03/05")]

[BugFixAttribute(107,"Jesse Liberty","01/04/05",
      Comment="Fixed off by one errors")]
public class MyMath
```

Tyto atributy budou uloženy společně s metadaty. Kompletní výpis zdrojového kódu můžete najít níže v sekci *Uživatelské atributy*.

Uživatelské atributy

```
using System;

// create custom attribute to be assigned to class members
[AttributeUsage(AttributeTargets.Class,
    AllowMultiple = true)]
public class BugFixAttribute : System.Attribute
{
   // attribute constructor for
   // positional parameters
   public BugFixAttribute
        (int bugID,
         string programmer,
         string date)
 {
         this.bugID = bugID;
         this.programmer = programmer;
         this.date = date;
 }

 // accessor
 public int BugID
 {
         get
         {
            return bugID;
         }
 }
```

```
 // property for named parameter
 public string Comment
 {
        get
        {
           return comment;
        }
        set
        {
           comment = value;
        }
 }

 // accessor
 public string Date
 {
        get
        {
           return date;
        }
 }

 // accessor
 public string Programmer
 {
        get
        {
           return programmer;
        }
 }

 // private member data
 private int      bugID;
 private string   comment;
 private string   date;
 private string   programmer;

}

// ********* assign the attributes to the class ********

[BugFixAttribute(121,"Jesse Liberty","01/03/05")] [BugFixAttribute(107,"Jesse
Liberty","01/04/05",
    Comment="Fixed off by one errors")]
public class MyMath
{

   public double DoFunc1(double param1)
   {
       return param1 + DoFunc2(param1);
   }

   public double DoFunc2(double param1)
   {
       return param1 / 3;
   }

}

public class Tester
{
   public static void Main()
   {
       MyMath mm = new MyMath();
       Console.WriteLine("Calling DoFunc(7). Result: {0}",
           mm.DoFunc1(7));
   }
}
```

Výstup je následovní:

```
Calling DoFunc(7). Result: 9.3333333333333339
```

Jak můžete vidět, atributy nemají absolutně žádný vliv na výstup a vytváření atributů zase nijak neovlivňuje výkonnost aplikace. Ve skutečnosti máte pouze mé slovo, že atributy vůbec existují. Ovšem již letmý pohled na metadata zobrazená pomocí aplikace ILDASM odkryje, že atributy jsou na svém místě.

Obr. 9.1: Metadata zobrazená v aplikaci ILDASM

REFLEXE

Pro plné využití naší ukázky budeme muset najít způsob, jak přistupovat k atributům prostřednictvím metadat, nejlépe za běhu aplikace. Programovací jazyk C# nabízí podporu reflexe pro prozkoumávání metadat. Začneme inicializací objektu třídy **MemberInfo** (tato třída je umístěna v jmenném prostoru **System.Reflection**). Objekt třídy **MemberInfo** lze využít pro analýzu atributů datového členu a pro přístup k metadatům.

```
System.Reflection.MemberInfo inf = typeof(MyMath);
```

Výše uvedený programový kód aplikuje operátor **typeof** na instanci třídy **MyMath**. Výsledkem práce operátoru je objekt třídy **System.Type**, jenž je odvozen ze třídy **MemberInfo**.

V následujícím kroku dochází k aktivaci metody **GetCustomAttributes** objektu třídy **MemberInfo**, které předáváme typ atributu, jenž hodláme vyhledat. Po splnění úkolu vrací metoda pole objektů typu **BugFixAttribute**:

```
object[] attributes;
attributes = Attribute.GetCustomAttributes(inf,
     typeof(BugFixAttribute));
```

Nyní můžeme vytisknout vlastnosti objektu **BugFixedAttribute** pomocí cyklu **foreach**, jak je uvedeno v části *Tisknutí vlastností*. Když tento náhradní programový kód umístíte do části *Uživatelské atributy*, budou zobrazena metadata.

Tisknutí vlastností

```
public static void Main()
{
   MyMath mm = new MyMath();
   Console.WriteLine("Calling DoFunc(7). Result: {0}",
      mm.DoFunc1(7));

   // get the member information and use it to
   // retrieve the custom attributes
   System.Reflection.MemberInfo inf = typeof(MyMath);
   object[] attributes;
   attributes =
       Attribute.GetCustomAttributes(inf,typeof(BugFixAttribute));

   // iterate through the attributes, retrieving the
   // properties
   foreach(Object attribute in attributes)
   {
     BugFixAttribute bfa = (BugFixAttribute) attribute;
     Console.WriteLine("\nBugID: {0}", bfa.BugID);
     Console.WriteLine("Programmer: {0}", bfa.Programmer);
     Console.WriteLine("Date: {0}", bfa.Date);
     Console.WriteLine("Comment: {0}", bfa.Comment);
   }
}
```

ODHALOVÁNÍ TYPŮ

Pomocí reflexe můžete prozkoumávat obsah assembly. To se vám může hodit v případě, kdy vyvíjíte software, jenž potřebuje zobrazovat informace o assembly, nebo když chcete dynamicky spouštět metody přítomné v dané assembly. Reflexe vám pomůže i při vývoji skriptovacího stroje, jenž bude uživatelům umožňovat generovat a spouštět skripty uvnitř vaší aplikace.

S reflexí můžete vyhledat datové typy asociované s moduly, metodami, datovými členy, vlastnostmi, a také události asociované s typy. Kromě toho můžete odhalit i signatury metod daného typu, dále rozhraní (která daný typ podporuje) a rovněž i jméno bázové třídy typu.

Naši další ukázku zahájíme dynamickým načtením assembly pomocí statické metody **Assembly.Load**. Signatura této metody je následovní:

```
public static Assembly.Load(AssemblyName)
```

Poté byste měli předat odkaz na hlavní běhovou knihovnu:

```
Assembly a = Assembly.Load("Mscorlib.dll");
```

Je-li jednou assembly načtená, můžete zavolat metodu **GetTypes**, která vrací pole objektů třídy **System.Type**. Objekt této třídy představuje srdce reflexe, protože pomocí něj můžeme získat přístup ke třídám, rozhraním, polím, hodnotovým typům a enumeracím.

```
Type[] types = a.GetTypes();
```

Pole dostupných typů můžeme zobrazit pomocí cyklu **foreach**. Výstupem může být i několik stránek informací. Uveďme si jenom malou ukázku:

```
Type is System.TypeCode
Type is System.Security.Util.StringExpressionSet
Type is System.Text.UTF7Encoding$Encoder
Type is System.ArgIterator
Type is System.Runtime.Remoting.JITLookupTable
1205 types found
```

Obdrželi jsme pole datových typů, které se nacházejí v hlavní běhové knihovně. Je to možná k neuvěření, ovšem knihovna obsahuje přes dvanáct set typů.

OBJEVOVÁNÍ TYPŮ

Kromě kompletního seznamu dostupných typů můžete pracovat i s jednotlivými typy. Provedete to tak, že požadovaný typ vyjmete z assembly pomocí metody **GetType**.

```
public class Tester
{
    public static void Main()
    {
        // examine a single object
        Type theType = Type.GetType("System.Reflection.Assembly");
        Console.WriteLine("\nSingle Type is {0}\n", theType);
    }
}
```

Výstupní data by měla vypadat takto:

```
Single Type is System.Reflection.Assembly
```

VYHLEDÁVÁNÍ ČLENŮ

Obdržíte-li požadovaný typ, může vám poskytnout informace o všech svých členech, tedy o metodách, vlastnostech a datových členech, jak můžete vidět v části *Získávání všech členů*.

Získávání všech členů

```
public class Tester
{
   public static void Main()
   {
      // examine a single object
      Type theType = Type.GetType("System.Reflection.Assembly");
      Console.WriteLine("\nSingle Type is {0}\n", theType);

      // get all the members
      MemberInfo[] mbrInfoArray =
          theType.GetMembers(BindingFlags.LookupAll);
      foreach (MemberInfo mbrInfo in mbrInfoArray )
      {
          Console.WriteLine("{0} is a {1}", mbrInfo,
              mbrInfo.MemberType.Format());
      }
   }
}
```

Ačkoliv je výstup opět poměrně dlouhý, můžete v něm vidět datové členy, metody, konstruktory a vlastnosti:

```
System.String s_localFilePrefix is a Fields
Boolean IsDefined(System.Type) is a Method
Void .ctor() is a Constructor
System.String CodeBase is a Property
System.String CopiedCodeBase is a Property
```

ZÍSKÁVÁNÍ POUZE METOD

V určitých situacích budete chtít vyhledat jenom metody, přičemž vás nebudou zajímat třeba datové členy či vlastnosti. Postupujte tak, že nejprve odstraníte kód, jenž provádí volání metody **GetMembers**.

```
MemberInfo[] mbrInfoArray =
     theType.GetMembers(BindingFlags.LookupAll);
```

Poté zavolejte metodu **GetMetods**.

```
mbrInfoArray = theType.GetMethods();
```

Výsledkem bude zobrazený seznam metod.

```
Výstup (výňatek)
Boolean Equals(System.Object) is a Method
System.String ToString() is a Method
System.String CreateQualifiedName(System.String, System.String)
 is a Method
System.Reflection.MethodInfo get_EntryPoint() is a Method
```

ZÍSKÁVÁNÍ SPECIFICKÝCH ČLENŮ

Vyhledávací proces můžete ještě optimalizovat tím, že budete vyhledávat pouze specifické datové členy požadovaného typu. Kupříkladu je možné, abyste vyhledávali jenom ty metody, jejichž název začíná písmeny "Get". Praktickou aplikaci ilustruje programový kód v části *Vyhledávání specifických členů*.

VYHLEDÁVÁNÍ SPECIFICKÝCH ČLENŮ

```
public class Tester
{
   public static void Main()
   {
      // examine a single object
      Type theType = Type.GetType("System.Reflection.Assembly");

      // just members which are methods beginning with Get
      MemberInfo[] mbrInfoArray
      theType.FindMembers(MemberTypes.Method,
             BindingFlags.Default,
             Type.FilterName, "Get*");
      foreach (MemberInfo mbrInfo in mbrInfoArray )
      {
         Console.WriteLine("{0} is a {1}", mbrInfo,
             mbrInfo.MemberType.Format());
      }
   }
}
```

Výstup je zde:

```
System.Type[] GetTypes() is a Method
System.Type[] GetExportedTypes() is a Method
System.Type GetType(System.String, Boolean) is a Method
System.Type GetType(System.String) is a Method
System.Reflection.AssemblyName GetName(Boolean) is a Method System.Reflection.AssemblyName
GetName() is a Method
Int32 GetHashCode() is a Method
System.Reflection.Assembly GetAssembly(System.Type) is a Method
System.Type GetType(System.String, Boolean, Boolean) is a Method
```

DYNAMICKÉ SPOUŠTĚNÍ METOD

Jakmile jste objevili cílovou metodu, můžete použít reflexi pro její spuštění za běhu programu. Například byste mohli aktivovat metodu **Cos** třídy **System.Math**, která vypočte a navrátí kosinus zadaného úhlu.

Začněme tím, že získáme typové informace o třídě **System.Math**:

```
Type theMathType = Type.GetType("System.Math");
```

Známe-li tyto informace, můžeme provést dynamické zrození instance dané třídy.

```
Object theObj = Activator.CreateInstance(theMathType);
```

Pro bezpečné vytvoření objektu použijeme statickou metodu **CreateInstance** třídy **Activator**.

Ihned poté, co je instance třídy **System.Math** vytvořena, můžeme zavolat metodu **Cos**. Ještě předtím si ovšem musíme připravit pole, které bude popisovat datové typy parametrů metody. Jelikož metoda **Cos** pracuje pouze s jedním parametrem, kterým je úhel, jehož kosinus chceme vypočítat, budeme potřebovat pole pouze s jedním prvkem. Do pole uložíme instanci třídy **System.Type**, která bude schopna přijímat hodnoty datového typu **System.Double**, které se posléze použijí jako vstupní data pro parametr metody **Cos**.

```
Type[] paramTypes = new Type[1];
paramTypes[0]= Type.GetType("System.Double");
```

Nyní můžeme předat jméno metody a pole metodě **GetMetod**.

```
MethodInfo CosineInfo =
    theMathType.GetMethod("Cos",paramTypes);
```

V této chvíli máme k dispozici objekt třídy **MethodInfo**, jenž můžeme použít pro aktivaci metody. Volané metodě musíme předat aktuální hodnoty parametrů opět v podobě pole.

```
Object[] parameters = new Object[1];
parameters[0] = 45;
Object returnVal = CosineInfo.Invoke(theObj,parameters);
```

Všimněte si, že jsme vytvořili dvě pole. První se jmenuje **paramTypes** a obsahuje typ parametrů, zatímco druhé s názvem **parameters** disponuje aktuálními hodnotami. Kdyby metoda přijímala dva argumenty, uvedená pole byste museli deklarovat tak, aby je mohla přijmout. Na druhé straně, pole by bylo zapotřebí vytvořit i v případě, kdyby signaturu metody netvořily žádné parametry. Potom by ovšem pole mělo nulovou velikost.

```
Type[] paramTypes = new Type[0];
```

Přestože tento zápis vypadá poněkud podivně, je zcela správný. Kompletní programovou ukázku můžete najít v části *Používání reflexe*.

Používání reflexe

```
using System;
using System.Reflection;

public class Tester
{
   public static void Main()
   {
      Type theMathType = Type.GetType("System.Math");
      Object theObj = Activator.CreateInstance(theMathType);

      // array with one member
      Type[] paramTypes = new Type[1];
      paramTypes[0]= Type.GetType("System.Double");

      // Get method info for Cos()
      MethodInfo CosineInfo =
          theMathType.GetMethod("Cos",paramTypes);

      // fill an array with the actual parameters
      Object[] parameters = new Object[1];
      parameters[0] = 45;
      Object returnVal = CosineInfo.Invoke(theObj,parameters);
      Console.WriteLine(
         "The cosine of a 45 degree angle {0}", returnVal);
   }
}
```

ZÁVĚR

Syntaxe programovacího jazyka C# není velmi odlišná od jazyka C++, což je atribut, který napomáhá tomu, aby byl přechod pro programátory v C++ do prostředí jazyka C# relativně snadný a bezproblémový. Nedejte se ovšem zmást podobnostmi z hlediska syntaktické skladby: Neopatrní vývojáři v C++ se mohou po provedení prvních krůčků v C# ocitnout v jedné z mnoha na první pohled neviditelných pastí. Jazyk C# přináší mnoho pozitivních programovacích rysů, které můžete ovládnout a využít při psaní vlastních aplikací. Ze všeho nejlepší je ovšem skutečnost, že nyní nejste nuceni psát manuálně ohromné bloky kódu, ale můžete sáhnout po již připraveném kódu, jenž se nachází v bázové knihovně tříd platformy .NET Framework. Je pochopitelné, že cílem této kapitoly nebylo vyčerpávající pojednávání o všech aspektech migračního procesu z C++ do C#. Udělali jste však první krok, což je důležité. Vývojová platforma .NET Framework a její společné běhové prostředí nabízejí hlubokou podporu pro vývoj aplikací pro operační systém Windows a Web.

Příbuzné články:
C# Offers the Power of C++ and Simplicity of Visual Basic
(Jazyk C# nabízí sílu C++ a jednoduchost Visual Basicu)

Další informace o vývojové platformě:

Getting Started with .NET Framework
(Začínáme s vývojovou platformou .NET Framework)

Převzato z magazínu MSDN Magazine, vydání červenec 2001.

O AUTOROVI

Jesse Liberty je autorem tuctu knih, které pojednávají o vývoji softwarových aplikací. Je prezidentem společnosti Liberty Associates Inc. (*www.LibertyAssociates.com*), která poskytuje kursy pro vývojáře na bázi platformy .NET Framework a programování na zakázku. Informace uvedené v této kapitole byly převzaty z Jessiho knihy *Programming C# (Programujeme s jazykem C#)*, kterou vydalo v roce 2001 nakladatelství O´Reilly & Associates, Inc.

10 Nativní a řízené C++: Jazyková interoperabilita

Autor: Kate Gregory, Gregory Consulting Limited

Jednou z otázek, kterou mi vývojáři často pokládají je, jak lze opětovně využít již napsaný programový kód jazyka C++ v projektech řízeného C++. Pravděpodobně až přílišná různorodost cest, kterými se při realizaci tohoto záměru mohou vývojáři vybrat, jim způsobuje bolení hlavy. Mým primárním cílem je zjemnit působení těchto obav. Je pravdou, že žonglování s tolika potenciálními programátorskými volbami může do značné míry zkomplikovat výběr správného rozhodnutí. V této kapitole uvidíte, jak tuto svízelnou problematiku vyřešit logicky, přičemž bude kladen důraz na poukázání na všechny důležité důsledky vybrané alternativy postupu.

Začněme uvažováním o typu C++ kódu, jenž píšete a používáte. Kód může představovat obchodní logiku, patentované algoritmy či jistý druh knihovny třetí strany. Možná, že disponujete zdrojovým kódem, a možná, že ne. V dnešním světe může programový kód vystupovat v několika odlišných podobách:

- Kód v statické knihovně: Pozůstává z .LIB souboru a doprovodných hlavičkových (.h) souborů. Pomocí direktivy **#include** začleníte odkaz na hlavičkové soubory a poté uskutečníte spojení s .LIB souborem.
- Kód v dynamicky linkované knihovně: Kód dynamicky linkované knihovny je uložen v souboru DLL ve společnosti s hlavičkovými (.h) soubory a eventuálně i s .LIB souborem.
- Kód v COM komponentě nebo v ovládacím prvku ActiveX.
- Kód v souboru .CPP a hlavičkové (.h) soubory, které lze zkompilovat do jedné z výše uvedených forem.

Budeme předpokládat, že máte k dispozici skutečný zdrojový kód, se kterým můžete provádět, co budete chtít (kupříkladu jej můžete přetvořit do COM komponenty). Řeč přijde na klady a zápory různých programových přístupů. Pokud nemáte po ruce zdrojový kód, nebo je-li tento kód uložen v programových balíčcích, vaše možnosti výběru budou méně početné. Avšak přinejmenším zjistíte charakteristiky vaší výchozí situace.

Ve světe programování existují miliony řádků otestovaného, přesného a užitečného programového kódu jazyka C++. Tento kód má obrovskou hodnotu. Vývojáři vědí, jak kód pracuje, a tudíž se mohou spolehnout na to, že odvede svoji práci znamenitě. Kód přešel testováním a odlaďováním a programátoři vědí, jak s ním pracovat. Mnohé softwarové společnosti investovali čas, energii a peníze do vývoje a údržby svého programového kódu, a proto je snadno pochopitelné, že se ho nechtějí jen tak vzdát. Na druhé straně by ovšem rády přešly do nového universa řízeného kódu, společného běhového prostředí, webových služeb a všech ostatních lahůdek vývojové platformy .NET Framework. Jsem tu proto, abych vám ukázala, že můžete přejít do nového světa řízeného C++ s vědomím, že váš stávající programový kód nepřijde nazmar.

DĚDICTVÍ

Nejprve si ukážeme několik technik, jejichž pomocí můžeme používat programový kód, jenž jsme „zdědili“ z nativního C++. Naše dědictví bude tvořeno pouze jednoduchou třídou, která obsahuje kód jedné metody.

LegacyArithmetic.h:

```
class ArithmeticClass
{
public:
  double Add( double num1, double num2);
};
```

LegacyArithmetic.cpp:

```
#include "legacyarithmetic.h"
double ArithmeticClass::Add(double num1, double num2)
{
  return num1 + num2;
}
```

Přestože je ukázkový kód docela jednoduchý, techniky, které si předvedeme, budou fungovat i s reálnými fragmenty programového kódu. Když jsem se připravovala na napsaní této kapitoly, opětovně jsem používala kód, jehož jsem se naposledy dotkla v roce 1994. Tento kód provádí přesné aritmetické operace s celými čísly libovolné délky (použitím textových řetězců pro uložení číslic) a se zlomky (pomocí dvou celých čísel typu **long**). Pouze s jedinou části knihovny kódu jsem měla potíže: Šlo o sekce s kódem asembleru pro počítače třídy 286. Bohužel jsem nemohla najít starý asembler, abych se mohla s nimi poprat a nahradit je kódem jazyka C++. Dobrou zprávou je, že procesory urazily za těch deset let vskutku dlouhou cestu, takže jsem již nebyla více nucena psát některé části knihovny v asembleru, abych získala dodatečné výkonností body.

V zájmu kompletnosti jsem napsala malou neřízenou aplikaci, která používá tuto starou knihovnu s programovým kódem. Její existence přijde vhod, když budeme později porovnávat účinky několika aplikovaných technik.

Main.cpp:

```
#include "stdafx.h"
#include <iostream>
using namespace std;

#include "legacyarithmetic.h"

int _tmain(void)
{
   ArithmeticClass arith;
   cout << "1 + 2 is " << arith.Add(1,2) << endl;
   return 0;
}
```

Vzhledem k tomu, že třída **ArithmeticClass** je neřízená a píšeme neřízenou aplikaci, nic nám nebrání v tom, abychom vytvořili instanci třídy na zásobníku a prováděli volání metod pomocí tečkového operátoru (.).

JAKÉ JSOU VAŠE VOLBY A KOMPROMISY?

Povězme, že máte k dispozici modul napsaný pomocí neřízeného C++, jenž je tvořen hromadou tříd s množstvím metod. Nyní se rozhodnete, že vytvoříte řízenou aplikaci typu **Managed C++ Application** (možná, že půjde o aplikaci s grafickým rozhraním – **Windows Forms Application**, jelikož tato technologie je podporována), která bude znovu používat již existující kód nativního C++. Můžete postupovat několika cestami:

- **COM Interop.** Můžete vzít váš kód a zaobalit jej do podoby COM komponenty. Poté můžete nativní kód volat z řízeného programovacího jazyka prostřednictvím technologie COM Interop.

- **P/Invoke.** Neřízený kód můžete přetvořit do dynamicky linkované knihovny (DLL) a aktivovat jej z řízeného programovacího jazyka pomocí technologie P/Invoke.

- **It Just Works.** Je-li váš kód uložen v .LIB souboru, nebo v knihovně DLL, jejíž společníkem je .LIB soubor, můžete se na něj z řízeného C++ jednoduše odkázat.

- **It Just Works II.** Můžete vzít vás starý C++ kód, plný volání do ATL, MFC, STL a bůh ví čeho ještě, a zkompilovat jej do podoby řízeného kódu. Této variantě říkám *Přenos pomocí XCopy* a budete ji muset vidět v akci, abyste uvěřili, že je realizovatelná.

- **Míchání řízeného a nativního C++ v jednom EXE souboru.** Řízený a nativní kód je skutečně možné používat v rámci jednoho spustitelného souboru (EXE). Výsledkem takovéhoto počínání je assembly, která obsahuje řízený kód společně s nativním kódem a kódem jazyka IL.

- **Vytvoření řízeného obalu.** Tak budete moci vystavit svůj kód i pro použití v „méně šťastných“ jazycích, jakými jsou třeba Visual Basic a Visual C#. Ovšem dávejte si pozor na problém smíšených knihoven DLL.

Mám v plánu charakterizovat každý z uvedených přístupů a poukázat na jejich klady a zápory v oblastech, jakými jsou výkonnost, vývojářský komfort, údržba a podobně. K jednomu tématu, a sice bezpečnosti, bych se ráda vyjádřila jako k prvnímu.

BEZPEČNÝ PŘÍSTUP KE KÓDU A OVĚŘITELNÝ KÓD

Pokud jste četli něco o bezpečném přístupu k programovému kódu, víte, že platforma .NET Framework obsahuje poměrně rozsáhlou sadu povolení pro aplikace: Můžete kontrolovat, zdali aplikace může navázat síťové spojení, zdali může přistupovat k SQL serveru, nebo zdali může získávat hodnoty proměnných prostředí atd. Ve všeobecnosti byste měli chtít, aby vaše aplikace byla schopná pracovat s co možná nejpřísnějšími povoleními: Jenom proto, že aplikace funguje dobře při bezpečnostním profilu *Full Trust* (*Plná důvěra*), nemusíte tento profil využívat, když lze aplikaci naprogramovat tak, aby mohla splňovat i náročnější kritéria.

Bezpečný přístup ke kódu zahrnuje povolení pro přístup k neřízenému kódu. Takže byste si mohli myslet, že existují jisté bezpečnostní důsledky, které se odvíjejí od volby pro přístup k nativnímu kódu. Zvláště tehdy, pokud přenášíte kód do jeho řízené podoby, byste mohli nabýt dojmu, že oproštění se od striktnější sady bezpečnostních povolení je skvělým nápadem. Ve skutečnosti ovšem tomu tak není.

Všechny řízené C++ aplikace vyžadují povolení typu *SkipVerification* (*Vynechání verifikace*). A všechny znamená skutečně všechny, tohle pravidlo platí dokonce i pro stoprocentní řízené aplikace v C++, které jsou kompilovány výhradně do kódu jazyka IL a neprovádějí žádná volání do nativního kódu. Povolení *SkipVerification* je velice mocné, protože dovoluje, aby byla assembly nahrána společným běhovým prostředím, aniž by byla verifikována (Abyste měli představu, jak je toto povolení mocné, uvažte toto: I povolení typu *Everything* (*Vše*) nezahrnuje povolení *SkipVerification*. Na to je zapotřebí aktivace povolení *FullTrust* (*Plná důvěra*)).

Verifikační proces vykonává zjišťování, aby se ujistil, že assembly může přistupovat jenom k těm paměťovým lokacím, ke kterým má autorizovaný přístup. Před příchodem Visual C++ .NET 2003 nemohl být kód řízeného C++ kompilován do podoby ověřitelného kódu. Nyní, ve verzi 2003, můžete vytvářet ověřitelný kód, ovšem jde o dosti složitou operaci a jsem ochotna se vsadit, že nikdo nebude vytvářet reálnou aplikaci pomocí řízeného C++, jejíž kód by byl plně ověřitelný. Nicméně, povím vám, co musíte udělat předtím, než se můžete spokojit s větší úrovní bezpečnosti, než jakou zprostředkovávají povolení *SkipVerification* či *FullTrust*.

- Žádné ukazatele na neřízená data, například `char *` nebo `int *`
- Žádné třídy, které nejsou předmětem automatické správy paměti
- Žádná ukazatelová aritmetika
- Žádné použití konverzních operátorů `static_cast<> a reinterpret_cast<>`
- Nepoužívejte It Just Works nebo direktivu **#pragma unmanaged** pro neřízený kód (P/Invoke a COM Interop jsou v pořádku, ovšem assembly bude potřebovat povolení pro přístup k neřízenému kódu)
- Optimalizace musejí být vypnuty
- Nevyvolávejte chybové výjimky u základních typů (kupříkladu vyvolávejte výjimku „Řetězec je příliš dlouhý.“)
- Použijte přepínač **/noentry** pro potlačení přístupu k běhové knihovně jazyka C (C Runtime Library)
- Přidejte alespoň jednu globální proměnnou (verifikátor neoblibuje prázdné sekce)
- Abyste potlačili ověřování verze společného běhového prostředí, vytvořte spojení se souborem **nochkclr.obj**

Dokážete-li provést všechny výše uvedené akce, zůstává vám jenom upravit podobu souboru **AssemblyInfo** vaší aplikace. Do tohoto souboru přidejte následující atribut:

```
[assembly: SecurityPermissionAttribute(SecurityAction::RequestMinimum,
SkipVerification=false)];
```

Na závěr použijeme pomocnou aplikaci s názvem **SetILOnly.exe**, jejíž pomocí provedeme úpravu hlaviček spustitelného souboru. Tímto způsobem naznačíme, že se jedná o ověřitelný kód. Nepředpokládám, že se však někdy dostanete až tak daleko. Vždyť posuďte sami: Musíte vypnout optimalizační nastavení, nesmíte používat funkce běhové knihovny jazyka C, ukazatelovou aritmetiku ani neřízené třídy. Domnívám se, že jazyk C++ používáte z jistého dobře uváženého důvodu, ovšem seznam procedur, které je nutné realizovat pro vytvoření ověřitelného kódu, eliminuje většinu výhod, jimiž se může C++ chlubit.

Pokud budete akceptovat skutečnost, že psaní ověřitelného kódu není pro vás nezbytně prioritou číslo jedna, budete moci zděděný C++ kód opětovně používat podle svých nároků na výkonnost, komfort či údržbu.

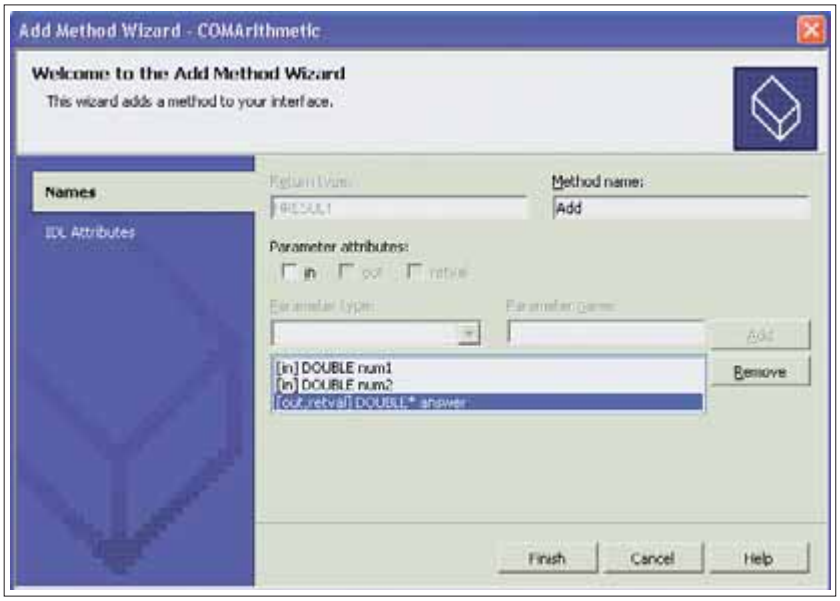
COM INTEROP

COM Interop je nepopiratelně technologií, která vzbudila největší pozornost. V podstatě lze říci, že z pohledu vašeho řízeného kódu se jakákoliv COM komponenta jeví jako řízený objekt, jenž existuje a pracuje na platformě .NET Framework. Dovolte mi, abych vám ukázala, jak provést konverzi běžné neřízené třídy do podoby COM komponenty a jak tuto komponentu používat v prostředí nativního a řízeného programového kódu.

Vytváření COM komponenty

Je nasnadě, že když přemýšlíte o COM v C++, v mysli se vám nejspíše vybaví tří magická písmena ATL. Knihovna Active Template Library (ATL) dělá z psaní COM komponent daleko jednodušší proces, než tomu bylo dříve. Můžete vytvořit ATL projekt, do kterého poté přidáte objekt, rozhraní (jedno nebo více) objektu a programový kód metod. Obdrželi-li jste zděděnou knihovnu s mnoha třídami a funkcemi, nejlepší bude, když začnete s návrhem rozhraní, a poté vytvoříte COM komponentu, která bude zaobalovat skutečné třídy. Prostřednictvím sestrojeného obalu můžete přistupovat k instancím jedné nebo i více skutečných tříd. Rovněž můžete volat metody těchto instancí, pomocí nichž budete poskytovat služby, které nabízejí implementovaná rozhraní. Jelikož já používám pouze jednoduchou třídu (**ArithmeticClass**) s jedinou metodou (**Add**), není nutné, abych se zvlášť zabývala návrhovou fází. Ve skutečnosti hodlám vytvořit tuto metodu ještě jednou, přičemž tělo metody jednoduše vyplním kódem z originální třídy.

V prostředí Visual Studia .NET jsem vytvořila nový projekt typu **ATL Project**, který jsem pojmenovala jako **COMArithmetic**. Přitom jsem použila všechna výchozí nastavení, která mi nabídl průvodce sestrojením projektu. Poté jsem do nově vytvořeného projektu přidala soubor s kódem třídy. Soubor přidáte tak, že klepnete pravým tlačítkem myši na název projektu v okně **Solution Explorer**, ukážete na položku **Add** a zvolíte příkaz **Add Class**. Dále jsem vybrala šablonu **ATL Simple Object**, nacož se spustil průvodce **ATL Simple ObjectWizard**, jehož pomocí jsem třídu pojmenovala jako **ArithmeticClass**. (Je dobré, když použijte název, jenž se liší od skutečné třídy. Tuto techniku můžete aplikovat při vytváření obalů, nebo třeba také při použití techniky kopírovat-a-vložit.) V okně **ClassView** jsem rozvinula uzel s názvem **COMArithmetic** a vyhledala rozhraní **ArithmeticClass**, které bylo automaticky vygenerováno. V dalším kroku jsem přidala novou funkci, a to tak, že jsem klepla na název rozhraní pravým tlačítkem myši, dále jsem ukázala na položku **Add** a konečně zvolila příkaz **Add Method**. Na obr. 10.1 můžete vidět, jak vypadalo dialogové okno **Add Method Wizard** chvíli předtím, než jsem aktivovala tlačítko **Finish**.



Obr. 10.1: Přidání nové metody

Po přidání metody jsem vyplnila její tělo následujícím kódem:

```
STDMETHODIMP CArithmeticClass::Add>
(DOUBLE num1, DOUBLE num2, DOUBLE* answer)
{
   *answer = num1 + num2;
   return S_OK;
}
```

Sestavení projektu vytvoří COM komponentu a provede její registraci v systému. Nyní je komponenta připravena k použití jinými aplikacemi.

Použití COM komponenty z neřízeného kódu

Jedním z důvodů, proč byste mohli převést neřízený kód do podoby COM komponenty je, že takto lze kód používat i z jiných aplikací staršího data, které byly napsány v nativním C++. Na následujících řádcích se nachází kód jednoduché konzolové aplikace, která používá naši novou COM komponentu.

```
#import "..\ComArithmetic\Debug\ComArithmetic.dll" no_namespace
int _tmain(int argc, _TCHAR* argv[])
{
   ::CoInitialize(NULL);
   //braces for scope only
   {
      IArithmeticClassPtr arith("ComArithmetic.ArithmeticClass");
      double answer = arith->Add(1,3);
      cout << "1 + 3 is " << answer << endl;
      cin.get();
   }
   ::CoUninitialize();
   return 0;
}
```

Vzhled kódu je zjednodušen pomocí direktivy **#import**, která připravuje třídu inteligentního ukazatele s názvem **IArtihmeticClassPtr**, která provádí zaobalení komponenty. Dodatečná sada složených závorek nás ujišťuje, že instance třídy inteligentního ukazatele bude podrobena destrukci předtím, než se spustí metoda **CoUninitialize**(). Srdcem uvedeného kódu je volání metody **Add**, které je realizováno prostřednictvím přetíženého operátoru nepřímého adresování (->) inteligentního ukazatele (který, samozřejmě, není skutečným ukazatelem). Ačkoliv je kód plný konstrukcí, které jsou známé z komponentní technologie COM, můžete si všimnout, že vystavené metody COM komponenty jsou přístupné z nativního kódu bez větších potíží.

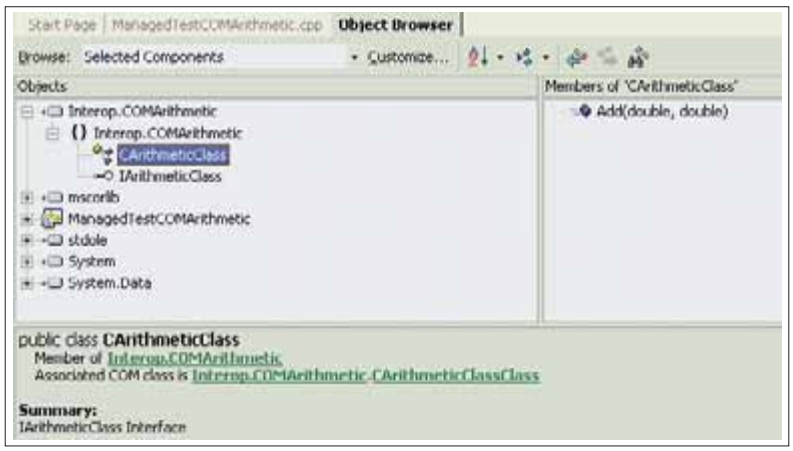
Použití COM komponenty z řízeného kódu

Používání COM komponenty z řízeného C++ je o poznání snazší než v nativním podání. Klíčem k úspěchu je použití tzv. Runtime Callable Wrapper (RCW), což je segment kódu, jenž zvenčí vypadá jako COM komponenta, ovšem ve svých útrobách obsahuje identifikační údaje, jejichž pomocí je schopen přesměrovat volání běhového prostředí a požádat tak skutečnou COM komponentu o zpracování požadavku. RCW si dovede poradit s metodami **QueryInterface**, **AddRef** a **Release**, dále s konverzí typů mezi COM a .NET a problémy ji nečiní ani převádění návratových hodnot typu **HRESULT** do formy řízených výjimek. Vzato kolem a kolem, pomocí RCW je život vývojáře zas o něco snadnější. RCW můžete přidat prostřednictvím záložky **COM** dialogového okna **Add Reference**.

Vytvořila jsem novou .NET konzolovou aplikaci (**Console Application (.NET)**), kterou jsem pojmenovala **ManagedTestCOMArithmetic**. Poté jsem v okně **Solution Explorer** klepla na položku **References** pravým tlačítkem myši a vybrala jsem položku **Add Reference**. (Tento programový rys je ve Visual Studiu .NET 2003 nový.) V zobrazeném dialogovém okně jsem zvolila záložku **COM**, kde jsem vyhledala položku **COMArithmetic**, kterou jsem označila. Jakmile byla položka označena, aktivovala jsem tlačítko **Select** a poté i tlačítko **OK**. Vývojové prostředí následně vytvořilo pro specifikovanou COM komponentu Runtime Callable Wrapper. Jméno výslední řízené třídy je složeno ze jména COM třídy (**CArithmeticClass**), ke kterému je přidáno ještě slovíčko **Class**. Kompletní jméno třídy je tedy **CArithmeticClassClass**. A nyní se podívejme na konkrétní programový kód:

```
double answer;
Interop::COMArithmetic::CArithmeticClassClass* arith =
   new Interop::COMArithmetic::CArithmeticClassClass();
answer = arith->Add(2,3);
Console::Write("2 + 3 is ");
Console::WriteLine(__box(answer));
return 0;
```

V tomto fragmentu kódu se můžete střetnout se dvěma nejasnostmi. Za prvé, jak jsem mohla vědět, že řízená třída, kterou jsem vytvořila pomocí operátoru **new** je nazývaná **Interop::COMArithmetic::CArithmeticClassClass**? Jednoduše jsem na nově přidanou referenci klepla pravým tlačítkem myši a zvolila položku **Open**, nacož se otevřelo okno **Object Browser** (obr. 10.2).



*Obr. 10.2: Okno **Object Browser***

Zde naleznete všechny informace, které potřebujete. Kromě jiného jde o plně kvalifikované jméno, které je složeno z názvu jmenného prostoru a třídy. Rovněž je možné vidět charakteristiku metody **Add**.

Za druhé, metoda **Add** disponuje návratovou hodnotou typu **double**. Abychom mohli hodnotu tohoto primitivního datového typu předat metodě **WriteLine**, musíme použít techniku, jíž se říká boxing. Mimochodem, můžete si být jisti, že při vzájemné interakci mezi nativním a řízených kódem se nevyhnete potížím s datovými typy: Jelikož metoda vrací data v nativní podobě, budeme je muset ještě před jejich použitím v řízeném prostředí podrobit jistým konverzím. Když budete mít štěstí, bude vám postačovat boxování hodnoty. V opačném případě budete muset tomuto procesu věnovat přece jenom více pozornosti.

Je COM Interop pro vás tou správnou volbou?

Jestliže je váš nativní kód již zapouzdřen v COM komponentě, bez jakýchkoliv dlouhých úvah použijte COM Interop. Máte-li k dispozici pouze kolekci tříd, dvakrát si promyslete, zdali má smysl je převádět do COM komponenty jenom proto, abyste se na ně mohli odvolávat z řízené aplikace. Ze všech technik, které si ukážeme a které umožňují opětovné použití nativního kódu, je COM Interop tou nejpomalejší. Vytváření COM komponenty a přidávání nezbytného programového kódu vás může stát mnoho energie, času a dalších prostředků, v závislosti od náročnosti a složitosti realizace konverzního procesu.

Skvělý východiskový bod získáte, když máte hned ze začátku před sebou hotovou COM komponentu, kterou byste rádi používali z řízeného kódu. V tomto případě ovšem rozhodně proveďte výkonnostní testy, jejichž pomocí získáte kýžené informace o rychlosti a stylu práce programového kódu. Pokud se použití COM Interop stává přílišnou zátěží, možná budete muset popřemýšlet o refaktorizaci kódu. Kupříkladu byste mohli přenést většinu kódu do samostatného .LIB souboru a nechat tak COM komponentu, jako i nový řízený kód používat tuto knihovnu kódu. Tak byste mohli získat optimální výkonnost, zatímco byste stále mohli vystavovat COM komponentu pro stávající nativní kód. Nebo byste mohli přenést celou komponentu do řízeného kódu a vystavit ji nativnímu kódu pomocí technologie .NET Interop (pomocí níž by se .NET řízená třída tvářila jako COM komponenta). Vše se samozřejmě odvíjí od vašich požadavků na výkonnost aplikace.

Jak ovšem může řízený kód používat knihovnu? Existují kromě COM Interop ještě jiné cesty pro znovupoužití programového kódu? Čtěte dál.

P/INVOKE

V tuto chvíli jste viděli zděděnou třídu s nativním kódem jazyka C++, která prováděla aritmetické operace a také jste se dozvěděli, jaké alternativy interoperability jsou vám, jako programátorovi v C++, k dispozici. Rovněž jsme si ukázali, jak zapouzdřit nativní třídu v COM komponentě a jak k takto upravené třídě můžeme přistupovat z neřízeného i řízeného programového kódu. Vzhledem k tomu, že technologie COM Interop je charakteristická svou signifikantní výkonnostní penalizací, měla by být implementována jenom v opravdu odůvodněných případech. Dobrou zprávou je, že existují také jiné postupy, jejichž pomocí lze volat funkce nativního C++ z řízeného prostředí.

Ano, uhodli jste, nyní budeme mluvit o technologii Platform Invoke, neboli P/Invoke. Na toto téma existuje docela hodně ukázek, z nichž většina se soustřeďuje na volání užitečných funkcí dynamicky linkovaných knihoven (DLL), které jsou součástí souborové struktury operačního systému Windows. V této části si ukážeme, jak přenést stávající C++ kód do knihovny DLL. Stranou pochopitelně nezůstane ani ozřejmení postupu volání kódu této knihovny z nativního i řízeného C++.

Vytváříme knihovnu DLL

Když se rozhodnete pro vytvoření knihovny DLL v neřízeném C++, můžete sestrojit kolekci globálních funkcí, nebo můžete vytvořit členské metody jedné či několika tříd. Pokud budete funkce knihovny DLL volat z neřízeného C++, napsání třídy a její členských funkcí je skvělým řešením. Nicméně, měli byste mít na paměti, že plně kvalifikovaná jména C++ funkcí jsou dekorována, a tudíž jsou tvořena informacemi o svých parametrech, jako i o třídě, součástí které tyto funkce jsou. Tato skutečnost není viditelná v případě, kdy používáte knihovnu DLL z jiné nativní aplikace. Bohužel, potíže se začínají objevovat v okamžiku, kdy se rozhodnete volat funkce knihovny DLL z řízeného kódu. Obsahuje-li vaše knihovna DLL pouze globální funkce, můžete předejít dekorování jejich jmen pomocí modifikátoru **extern "C"**. Není ovšem nutné, abyste se vzdávali objektově orientovaných přístupů: Napište pro každou třídu jednu knihovnu DLL a členské funkce tříd implementujte jako globální funkce v knihovně DLL.

Přijmete-li tuto koncepci při vytváření rozsáhlého a komplexního systému, pamatujte na to, že nemusíte konvertovat váš starý kód do knihovny DLL: Můžete napsat novou knihovnu DLL, která bude nabízet jisté služby a které bude volat starý kód pro implementaci těchto služeb. V následující ukázce si předvedeme, jak začlenit metodu **Add** do knihovny DLL.

Vytvořila jsem nový Win32 projekt typu DLL. Poté jsem přidala kód funkce **Add**:

```
extern "C" __ declspec (dllexport )double Add(double num1, double num2)
{
   return num1 + num2;
}
```

Jak jsme si již pověděli, modifikátor **extern "C"** zabraňuje tomu, aby bylo jméno globální funkce dekorováno (dekorace jmen je u globálních funkcí implicitní). Modifikátor **__declspec(dllexport)** říká kompilátoru, že tato funkce má být exportována z knihovny DLL. Zbytek kódu funkce je snadno pochopitelný: V složitějším systému by funkce hrála roli obalové funkce, která by volala metody jiných tříd nebo jiné funkce z jiné dynamicky linkované knihovny.

Použití knihovny DLL – Dřívější přístup

Pro potřeby volání funkce **Add** jsem vytvořila běžnou neřízenou konzolovou aplikaci. Programový kód má následující podobu:

```
#include "stdafx.h"
#include <iostream>
using namespace std;
extern "C" __ declspec (dllimport ) double Add(double num1, double num2);
int _tmain (int argc , _TCHAR* argv [])
{
   cout << "1 + 2 is " << Add(1,2) << endl;
   return 0;
}
```

Abych předešla problémům s konflikty knihoven DLL známé jako „Peklo DLL“, dávám přednost vytvoření lokálních kopií jakýchkoliv knihoven DLL, které používám. Takovéto úpravy knihovny DLL neovlivní můj kód, dokud se nerozhodnu kopírovat pozměněnou knihovnu DLL. Soubor dynamicky linkované knihovny legacy.dll jsem zkopírovala do projektové složky konzolové aplikace. Když provedete sestavení projektu knihovny DLL ve Visual C++, získáte .LIB soubor, jenž představuje importovací knihovnu. Tento soubor jsem zkopírovala na stejné místo. Další postup je již jenom otázkou navázání spojení mezi projektem a importovací knihovnou. V okně **Solution Explorer** klepněte pravým tlačítkem na název projektu a zvolte položku **Properties**. Rozviňte sekci **Linker**, vyberte položku **Input** a do pole **Additional Dependencies** vložte odkaz na soubor legacy.lib.

Pro použití knihovny DLL je to vše. Co se týče programovací náročnosti, není uvedený postup o moc odlišnější ve srovnání s navazováním spojení se statickou knihovnou. Skutečným problémem je ovšem to, že jakmile dojde ke změně kódu knihovny DLL, váš kód začne používat tento nový kód. Jedná se o vskutku dvojsečnou zbraň. Ponecháte-li si soukromou kopii, můžete obavy pustit z hlavy. Současně však umožníte starému neřízenému kódu, jakožto i novému řízenému kódu používat shodnou knihovnu.

Použití knihovny DLL – Nový přístup

Jakým způsobem používá řízený kód knihovnu DLL? Přece prostřednictvím technologie Platform Invoke, tedy P/Invoke. Většina programových ukázek v dokumentaci ilustruje, jak lze přistupovat k dynamicky linkovaným knihovnám systému Windows v případě, kdy bázová knihovna tříd nenabízí přímou podporu jistého programového rysu, jenž hodláte použít. Kromě systémových knihoven DLL však můžete samozřejmě používat i vaše vlastní, neboli uživatelské DLL.

Budete-li chtít pracovat s funkcí **Add** v konzolové aplikaci, budete ji muset deklarovat pomocí atributu **DllImport**:

```
using namespace System::Runtime::InteropServices;
extern "C" {
[DllImport("legacy")]
double Add(double num1, double num2);
}
```

Jak si můžete všimnout, parametrem atributu je textový řetězec legacy, jenž specifikuje název knihovny DLL. Zajímavostí je, že můžete vynechat souborovou extenzi (v některých jiných programovacích jazycích platformy .NET je však zahrnutí extenze obligatorní). Pro použití atributu **DllImport** nemusíte navazovat žádné dodatečné reference, protože direktiva **using** provádí vše potřebné.

Soubor knihovny DLL zkopírujte do projektové složky. Nyní můžete aktivovat funkci:

```
System::Console::Write (S"1 + 2 is ");
System::Console::WriteLine (__box( Add(1,2)));
```

Jelikož metoda **WriteLine** neumí pracovat s hodnotami typu **double**, musíte návratovou hodnotu funkce (která je typu **double**) zaboxovat.

Ačkoliv byla uvedená programová ukázka docela názorná, nepředváděla skutečnou sílu technologie P/Invoke. Pomocí P/Invoke můžete totiž nejenom volat funkce z DLL, rozsah vestavěné působnosti je mnohem širší. Kupříkladu, pokud parametr funkce knihovny DLL přijímá ukazatel na typ **char** (**char***), můžete deklarovat .NET verzi této funkce s parametrem typu **System::String*** (ukazatel na typ **System::String**) a společné běhové prostředí zabezpečí nezbytnou automatickou konverzi typů. Kromě toho můžete zapojit do hry rozmanité atributy, jejichž pomocí můžete ovládat třeba konverzi textových řetězců či rozvržení struktur. Dokonce si můžete napsat svůj vlastní kód pro uskutečňování konverzí mezi neřízenými a řízenými datovými typy. Dále v tomto textu prodiskutujeme některé pokročilejší charakteristiky technologie P/Invoke.

Je P/Invoke pro vás tou pravou volbou?

Má vůbec smysl převádět váš stávající neřízený kód do podoby knihovny DLL a poté volat funkce této knihovny z řízeného kódu prostřednictvím P/Invoke? Dobrá, jde o poněkud složitou otázku. V každém případě však P/Invoke představuje výkonnější řešení než COM Interop. Nicméně, když pro vás pracuje standardní konverze typů, může se použití P/Invoke jevit jako malinko přehnaný luxus. Proto si na následujících řádcích ukážeme, jak je možné přistupovat ke stejné knihovně DLL bez použití atributu **DllImport**. Budeme používat jenom čisté C++.

KDO POTŘEBUJE P/INVOKE?

Když se budete toulat po Google a do vyhledávače zadáte textový řetězec „DLL managed code“ (DLL řízený kód), zcela jistě naleznete mnoho stránek pojednávajících o technologii P/Invoke. Po pravdě řečeno, z hlediska jazyků Visual Basic .NET a C# jde o jedinou variantu pro přístup ke knihovně DLL. Ovšem, jak ráda každého upozorňuji, C++ je speciální.

Jedním z pilířů této speciálnosti jazyka C++ je technologie s názvem It Just Works, jejíž pomocí lze provádět volání neřízeného kódu z řízených aplikací, aniž byste byli nuceni aplikovat jakékoliv speciální atributy, příznaky, volby, nebo cokoliv jiného. Zkusme znovu uvažovat o ukázce, kterou již dobře známe: jedná se o třídu, která uskutečňuje aritmetiku. Již jste byli svědky toho, jak jsme tuto třídu převedli do podoby knihovny DLL a volali její metodu prostřednictvím technologie P/Invoke. Metoda pracovala pouze s primitivními typem **double**, tudíž nevznikaly žádné požadavky na konverzi hodnot. Aplikovaný atribut **DllImport** jsme používali jenom proto, abychom kompilátoru dodali informace o lokaci dynamicky linkované knihovny.

Knihovna DLL přichází s importovací knihovnou, jejíž kód je uložen v souboru s extenzí LIB. Odkaz na tuto knihovnu můžete začlenit do své aplikace. Když píšete neřízenou aplikaci a začleníte odkaz na importovací knihovnu, za běhu programu dojde k implicitnímu načtení a spuštění knihovny DLL. Toto chování nařídíte pomocí vlastnosti **Additional Dependencies**, která se nachází na stránce **Input** pod položkou **Linker** v okně s vlastnostmi projektu. Když vytváříte řízenou aplikaci, postup není odlišný. I v tomto případě můžete zahrnout odkaz na importovací knihovnu.

Projekt využívající knihovnu DLL jsem překopírovala do jiné složky. Mějte na paměti, že knihovna DLL již byla zkopírována do projektové složky. Do stejné lokace jsem zkopírovala také importovací knihovnu a dále jsem provedla dvě úpravy. Za prvé, z deklarace funkce **Add** jsem odstranila atribut **DllImport**, takže nyní má deklarace této funkce následující podobu:

```
extern "C" {
double Add(double num1, double num2);
}
```

Pokračovala jsem přidáním odkazu na importovací knihovnu. Tento kód pracuje nádherně: Nyní je vhodná chvíle pro poukázaní na skutečnost, že technologie It Just Works je skutečně hodná svého jména.

V ČEM JE ROZDÍL?

Ukázková aplikace může volat funkci **Add** dynamicky linkované knihovny bez ohledu na to, zdali se rozhodnete pro aplikaci technologie P/Invoke či nikoliv. Co se týče volání funkce, je použit stejný programový kód. Rovněž vynaložené úsilí na začlenění atributu **DllImport** je srovnatelné s nastavením vlastnosti **Additional Dependencies**. Logicky se tedy naskýtá otázka, který z přístupů upřednostnit.

Začněme konstatováním, že ne všechny knihovny DLL jsou dodávány také s příslušnými .LIB soubory. Pokud vytváříte uživatelskou knihovnu DLL pro opětovné použití zděděného kódu, nebo tvoří-li společnost knihovny DLL i importovací knihovna, můžete tuto možnost využít. Na druhé straně, jestliže jste obdrželi knihovnu DLL z jiného zdroje, předcházející výhody můžete postrádat.

Technologie P/Invoke vám umožňuje kontrolu nad realizací marshalingu a konverzních operací včetně hladké konverze textových řetězců. Je-li zapotřebí, aby byly hodnoty parametrů modifikovány při každém volání funkce (kupříkladu konverze mezi **System::String** a **char***), použití P/Invoke se jeví jako dobrá volba.

Budete-li přistupovat k neřízenému kód přes P/Invoke, bude vyžadováno povolení nižšího stupně nežli *FullTrust* (*Plná důvěra*). Jsou i jiné možnosti, ačkoliv vaše aplikace potřebuje zmíněné povolení v každém případě. Když však projdete všemi zatěžkávajícími zkouškami pro vývoj ověřitelného kódu, technologie P/Invoke vaši snahu nezmaří a ponechá kód ověřitelným.

Proč by tedy někdo měl používat technologii It Just Works místo P/Invoke? Existují dva pádné důvody, jeden lepší než druhý. První důvod je psychologický: Nemusíte se totiž učit ničemu novému. Jednoduše dělejte to, co jste dělávali – proveďte spojení s importovací knihovnou, deklarujte funkce (dokonce můžete používat staré hlavičkové soubory s modifikátory **__declspec**, tyto však budou ignorovány) a volejte je. Druhým důvodem je výkonnost. Procházení zásobníku, kontrola bezpečnostních pravidel, marshaling a vlastně vše, co poskytuje P/Invoke, je spojeno s jistou výkonnostní penalizací. Nemáte-li z těchto operací žádné výhody, proč byste měli za jejich realizaci zbytečně platit ztrátou výkonu? IJW je rychlejší než P/Invoke, takže pokud chcete tuto technologii použít, dlouho nepřemýšlejte a vhrněte se na ní.

PORTACE POMOCÍ XCOPY

Technologie It Just Works vám slibuje něco na první pohled neskutečného: Můžete vzít kousek kódu jazyka C++, jenž působil jako neřízený kód a pak jej zkompilovat do podoby kódu jazyka IL. A světe div se, kód bude i nadále fungovat tak, jak má. Není vůbec podstatné, co kód provádí, nebo jaká spojení navazuje. Jednoduše je zaručeno, že když jej zkompilujete v řízeném prostředí, bude odvádět řádnou práci. Fungovat bude skutečně vše, či už jde o staticky, nebo dynamicky linkované knihovny, aplikace MFC, ATL, STL, nebo vaše vlastní knihovny.

Stávající C++ kód se může stát použitelný prostřednictvím řízeného kódu i tak, že jej na řízený kód převedete. Ve skutečnosti vyžaduje tento postup daleko menší úsilí, než byste se mohli domnívat. Uvedený postup ráda nazývám jako „Portace XCOPY“, protože aplikuje stejné koncepty jako rozmísťování aplikací pomocí příkazu XCOPY. Pro ty, kdo o XCOPY ještě v životě neslyšeli, si připomeňme, že se jednalo o jeden z mnoha příkazů operačního systému DOS, jehož pomocí bylo možné kopírovat najednou několik souborů. Termínem XCOPY se označuje také proces distribuce aplikací, v rámci něhož jsou assembly zkopírovány do příslušných složek cílové počítačové stanice, aniž by bylo nutné cokoliv konfigurovat, registrovat či instalovat. XCOPY portace v našem podání ovšem představuje poněkud více, nežli pouhé zkopírování kódu do nového projektu a jeho kompilaci.

Vše, co potřebujete, je sestrojení řízeného projektu (já jsem vytvořila konzolovou aplikaci, takže jsem mohla relativně snadno připravit testovací kód), zkopírování všech souborů s extenzí .cpp a .h z původní složky do složky nového projektu a jejich přidání do projektu. Nacházíte-li se v prostředí Visual Studia .NET, klepněte v okně **Solution Explorer** pravým tlačítkem myši na název projektu, ukažte na položku **Add** a klikněte na položku **Add Existing Item**. Přidejte do projektu odkazy na všechny soubory, které jste právě zkopírovali.

Dále použijte řízený kód tak, jak byste očekávali. Například, níže je uvedena řízená verze funkce **main**, která používá třídu **ArithmeticClass**:

```
#include "stdafx.h"
#using <mscorlib.dll>
#include "LegacyArithmetic.h"
using namespace System;
int _tmain()
{
   ArithmeticClass a;
   Console::Write(S"1 + 2 is ");
   Console::WriteLine(__box( a.Add(1,2)));
   return 0;
}
```

Přestože třída **ArithmeticClass** není řízenou třídou a její instance tudíž nejsou kontrolovány automatickou správou paměti, nic nám nebrání v tom, abychom tuto třídu používali v řízeném kódu. Po sestavení projektu je vybudován spustitelný soubor (EXE), v němž je uložena assembly s kódem jazyka IL. Máte-li chuť, můžete se na IL kód podívat pomocí aplikace ILDASM. Kupříkladu IL kód metody **Add** má tuto podobu:

```
.method public static float64 modopt(
 [mscorlib]System.Runtime.CompilerServices.CallConvThiscall)
   ArithmeticClass.Add( valuetype ArithmeticClass*
 modopt([Microsoft.VisualC]Microsoft.VisualC.IsConstModifier)
 modopt([Microsoft.VisualC]Microsoft.VisualC.IsConstModifier) A_0,
                            float64 num1,
         float64 num2) cil managed
{
  .vtentry 45 : 1
  // Code size   6 (0x6)
  .maxstack 2
  IL_0000:  ldarg.1
  IL_0001:  ldarg.2
  IL_0002:  add
  IL_0003:  br.s  IL_0005
  IL_0005:  ret

} // end of method 'Global Functions'::ArithmeticClass.Add
```

Je tomu skutečně tak, třída, která byla původně definována v nativním kódu, si nyní dobře rozumí i s řízeným prostředím. Ovšem pozor: Třída není řízená, což znamená, že její instance nejsou z paměti počítače automaticky uvolňovány v okamžiku, kdy již nejsou zapotřebí. Jinými slovy, i nadále můžete vytvářet instance třídy na zásobníku, či na hromadě a rovněž musíte vhodně ošetřit uvolňování instancí z paměti. Jak vidíte, tento jednoduchý program v C++ je schopný pracovat i s neřízenými daty, což je vlastnost, kterou jiné řízené programovací jazyky, jako třeba C#, nedokáží zcela realizovat. Pokud vše, co jste chtěli udělat, bylo napsat novou aplikaci v řízeném C++, přesvědčili jste se, že tato technika pracuje báječně. Plánujete-li však vytvořit kód, jenž by byl použitelný i v jiných jazycích, věřte, že i tohle je možné.

ŘÍZENÝ OBAL

Dobrá, dokázali jsme provést kompilaci nativního kódu třídy v C++ v řízeném prostředí. Přestože je kód třídy uložen v podobě IL kódu, pořád se jedná o neřízenou třídu, která je schopná pracovat s neřízenými daty. Třídu můžete použít v jakékoliv nově založené řízené C++ aplikaci, která bude pracovat pod křídly platformy .NET Framework.

Zkuste si ovšem představit, že vaše třída je tak ohromující, že jí budou chtít používat všichni vývojáři ve vaší společnosti, dokonce i ti, kteří pracují s jazyky Visual Basic či C#. Dříve zmiňované techniky, jako je COM Interop nebo P/Invoke, vám umožňují sestrojit knihovnu tříd, která může být použita v kterémkoliv řízeném jazyce. Portace prostřednictvím XCOPY bohužel nefunguje. Jenom C++ může pracovat s neřízenými daty.

Jestliže byste obohatili deklarační příkaz třídy **ArithmeticClass** o klíčové slovo **__gc**, z nativní třídy by se stala třída řízená. Tím pádem by třída podléhala pravidlům řízeného prostředí: kupříkladu by nemohla aplikovat koncepci vícenásobné dědičnosti. Bude-li nutné jakkoliv upravit třídu pro její použití v řízeném prostředí, budete nejspíš muset udržovat dvě verze třídy, jednu v nativní a druhou v řízené podobě. Lepší alternativou bude, když pro vaši neřízenou třídu vytvoříte řízený obal. Třída obalu je zcela řízená a pracuje s řízenými daty. Tudíž ji lze použít z jakéhokoliv jiného řízeného programovacího jazyka. Vaši neřízenou třídu můžete zkompilovat do kódu jazyka IL (díky technologii It Just Works), nebo ji můžete nechat ve stávající (nativní) podobě a aktivovat ji přes řízenou třídu obalu (opět díky It Just Works).

Jak můžete ponechat stávající třídu v nativním kódu? Ve skutečnosti ji může nechat tak, jak je (povězme v .LIB souboru) a jednoduše ji aktivovat z vašeho řízeného kódu (viz *Kdo potřebuje P/Invoke?*). Nebo ji můžete přenést do aplikačního projektu, ovšem v tomto případě deaktivujte volbu **/clr** pro příslušný soubor se zdrojovým kódem. Tuto akci provedete následovně: V okně **Solution Explorer** klepněte pravým tlačítkem myši na název souboru (v našem případě jde o soubor LegacyArithmetic.cpp) a vyberte položku **Properties**. Ve složce **C++ Properties** najděte položku **General** a vyhledejte vlastnost **Compile As Managed**. Vlastnost nastavte na hodnotu **Not Using Managed Extensions**. A je to! Uprostřed assembly bude vyhrazen prostor pro uložení kýženého nativního kódu.

Budete si muset pohrát ještě s dalšími volbami, kupříkladu budete muset vypnout tvorbu předkompilovaných hlaviček pro všechny soubory, které hodláte v řízeném prostředí kompilovat do nativního kódu (důvodem takovéhoto počínání je, že hlavičky by se záhy staly nekompatibilními). Nemusíte mít však obavy, protože na všechny potenciální nesrovnalosti vás výslovně upozorní samotný kompilátor. Pravděpodobně budete muset také odstranit projektové reference (které jsou aplikovány na všechny soubory v projektu) a soubory, které budete chtít kompilovat do IL kódu, budete muset identifikovat pomocí direktivy **#using** (ve stylu verze 2002).

Ne všechen stávající kód však budete chtít kompilovat v jeho nativní podobě. Vaše rozhodnutí bude asi záviset od stanovení jistých mezí pro nativní a řízený kód. Jelikož je konverzní proces vykoupen jistou ztrátou výkonu, nejlépe uděláte, když popřemýšlíte nad vaší současnou situací a vyberete to řešení, které bude v co možná největší míře minimalizovat absenci výkonnostních bodů.

Předpokládejme, že vaše neřízená třída obsahuje pět až deset vlastností a že k nim přistupujete pomocí „upovídaného“ rozhraní: Bude provedeno jedno volání pro každou vlastnost, které je následováno finálním voláním, jenž něco skutečně provádí. Při psaní třídy obalu se rozhodnete, že nabídnete „robustní“ rozhraní: Bude realizováno jedno volání, které vezme pět až deset parametrů a použije je pro nastavení vlastnosti a poté aktivuje požadovanou metodu. Když přenesete starou třídu do řízeného kódu, hranice se posunou, takže všechna upovídaná volání budou uskutečňována v řízeném kódu. Je ovšem možné, že vaše stará třída již implementuje robustní rozhraní a že používá volání nějaké jiné nativní třídy. V této situaci můžete ponechat třídu v neřízené podobě, což znamená, že bude provedena jenom jedna konverze v okamžiku, kdy budete volat třídu přes řízenou obalovou vrstvu. Všechna upovídaná volání budou kompletně realizována v neřízeném kódu. Mez mezi nativním a řízeným světem můžete vhodně stanovit jenom za předpokladu, že víte, jak váš kód pracuje. V tomto směru neexistuje žádné ideální „kuchařské“ řešení.

Napsání třídy obalu je docela jednoduché. Níže můžete vidět programový kód třídy obalu pro nativní třídu **ArithmeticClass**:

```
public __gc class ArithmeticWrapper
{
private:
    ArithmeticClass* ac;
public:
    ArithmeticWrapper() {ac = new ArithmeticClass();}
    double Add(double num1, double num2)
    {return ac->Add(num1,num2);}
    ~ArithmeticWrapper() {delete ac;}
};
```

Jak můžete vidět, ukazatel na instanci neřízené třídy je uchován v členské proměnné. (Musíme pracovat s ukazatelem, protože řízené třídy nedovolují, abychom měli členské proměnné, které jsou instancemi neřízených tříd.) Zatímco konstruktor vytváří instanci třídy, destruktor ji odstraňuje a obalová funkce předává parametry a uskutečňuje volání příslušných funkcí neřízené třídy. Ve skutečnosti bude takřka každá třída obalu vypadat jako uvedený vzor, samozřejmě s tím rozdílem, že skutečné třídy disponují více metodami a ne pouze jednou, jak je tomu v našem případě. Nabízí se vám také příležitost pro vylepšení vašeho rozhraní, kupříkladu můžete nahradit robustní rozhraní upovídanou variantou tak, že vytvoříte obalovací metodu, která bude volat mnoho jiných metod zaobalené třídy.

Jakmile úspěšně napíšete řízenou třídu obalu, která pracuje s řízenými daty a kódem, mohou ji bez váhání použít i vývojáři pracující s jazyky Visual Basic a Visual C#. Skutečné jádro spočívá v originální nativní třídě, ke které přistupuje jednak váš nativní i řízený kód. Jste sice osvobozeni od výkonnostní penalizace, která je typická pro technologie COM Interop nebo P/Invoke, ovšem tato výhoda je na druhou stranu vykoupena tranzitivními náklady na přechod do řízeného prostředí. Tyto náklady však můžete minimalizovat rozhodnutím, které třídy kompilovat do nativního a které do řízeného (IL) kódu. V tomto směru vám napomáhá skutečnost, že volbu **/clr** můžete nastavovat separátně pro jednotlivé projektové soubory.

A jsme u konce. V této kapitole jste viděli, jak mohou programátoři opětovně používat stávající C++ kód v novém a řízeném prostředí. Přestože existují ještě další postupy pro realizaci jazykové interoperability, základní „velká trojka“, jež je tvořena technologiemi COM Interop, P/Invoke a It Just Works, plní svůj účel na jedničku. Poté, co jste viděli tyto techniky v akci, nemáte chuť vyzkoušet si je sami v praxi?

O AUTORCE

Kate Gregory je spoluzakladatelkou společnosti Gregory Consulting Limited (*www.gregcons.com*). V lednu roku 2002 byla vyjmenována za regionální ředitelku sítě MSDN pro Toronto v Kanadě. Své zkušenosti s jazykem C++ nabyla ještě dávno předtím, než světlo světa spatřil vývojový nástroj Visual C++. Kate je ctěným řečníkem a přednášejícím na konferencích společnosti Microsoft, přičemž se soustřeďuje na témata, mezi která patří vývojová platforma .NET Framework, Visual Studio, XML, UML, C++, Java a Internet. Společně se svými kolegy se ve společnosti Gregory Consulting Limited specializují na kombinaci vývoje softwaru a atraktivních webových sídel. Vytvářejí kvalitní zákaznická řešení a hotové softwarové komponenty pro využití ve webových stránkách, nebo jiných počítačových aplikacích. Kate je rovněž autorkou mnoha knih. Ze všech vzpomeňme alespoň *Special Edition Using Visual C++ .NET* a připravovanou *Microsoft Visual C++ .NET 2003 Kick Start for Sams*.

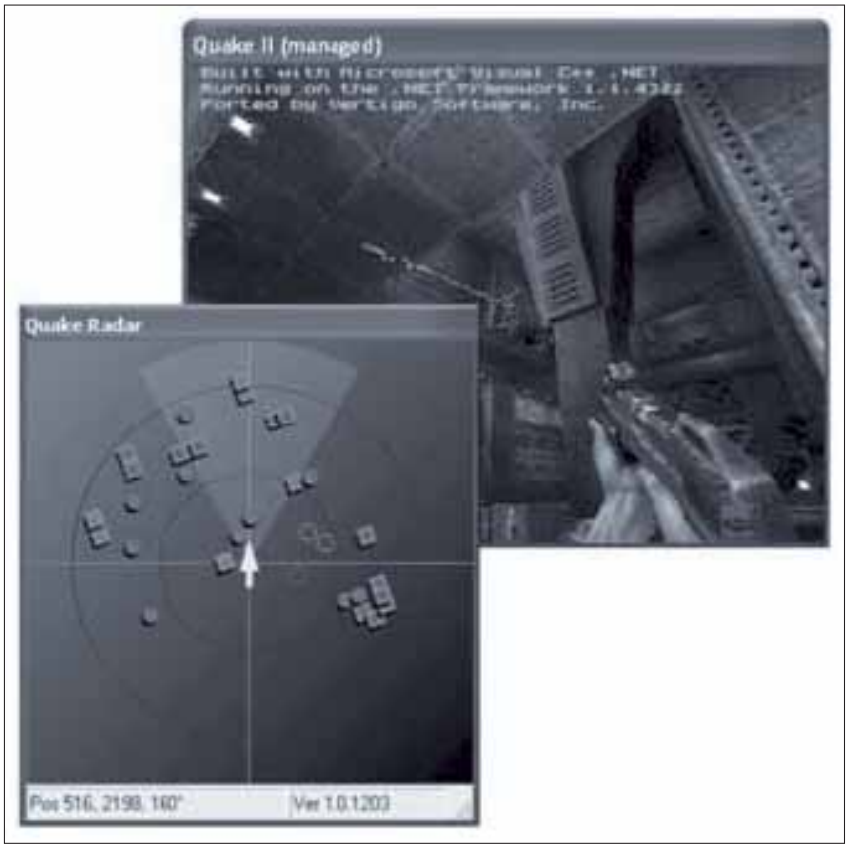
11 Quake II .NET

Autor: Ralph Arveson, Vertigo Software, Inc.

V roce 1997 uvedla společnost id Software na trh revoluční 3D akční střílečku z pohledu první osoby s názvem Quake II. Tato počítačová hra byla vskutku úspěšná, a proto není divu, že se jí prodalo více než milion kopií. Kromě toho hra získala i významná ocenění za přínos v oboru, jako například *Hra roku*. O něco později, v prosinci roku 2001, společnost id Software velkoryse uvolnila zdrojový kód hry a zejména 3D renderovacího enginu pod záštitou licence GNU GPL (General Public License) pro volné použití.

V červnu roku 2003 uvedla společnost Vertigo Software, Inc. .NET verzi počítačové hry Quake II pod názvem Quake II .NET. Quake II .NET je výsledkem portace primárního enginu, jenž byl napsán v jazyce C, do řízeného prostředí Visual C++ .NET. Kromě přenesení kompletního originálního zdrojového kódu ovšem .NET verze nabízí také dodatečnou funkcionalitu ve formě radaru, jenž mapuje prostředí hry. Portací úspěšné počítačové hry na platformu .NET Framework jsme chtěli poukázat na jednu skutečnost: Je skutečně možné převést rozsáhlý kód v C do C++, a poté využít dovednosti řízeného prostředí a společného běhového prostředí (Common Language Runtime , CLR), aniž bychom postřehli citelné snížení výkonu aplikace. Jakmile jsme hru Quake II přenesli do řízené .NET aplikace, přidání nových prvků bylo jen otázkou několika okamžiků. Abych nezapomněl, převod kódu jako i práce v řízeném prostředí skýtá až neobvyklou lehkost a radost z dobře odvedené práce.

V této kapitole se dozvíte, jak probíhala portace a rozšiřování hry Quake II až se z ní stala verze s přídomkem .NET.



Obr. 11.1: Počítačová 3D akční střílečka Quake II .NET v akci (všimněte si radar, který byl nově přidán)

ZDROJOVÝ KÓD A SOUBORY

Kompletní zdrojový kód hry Quake II můžete získat od společnosti id Software na následující adrese: *ftp://ftp.idsoftware.com/idstuff/source/quake2.zip*. Zdrojový kód je dostupný v rámci licence GNU General Public License. Pro více informací ohledně GPL vám doporučuji si přečíst doprovodné textové soubory readme.txt a gnu.txt. Počítačová hra Quake II je tvořena dvěma hlavními částmi: herním enginem a daty.

Herní engine

Engine hry pozůstává z programového kódu, jenž je odpovědný za běh hry. Tento kód je rozložen do následujících souborů:

Tab. 11.1 Přehled základních zdrojových souborů hry

SOUBOR	POPIS
quake2.exe	Hlavní spustitelný soubor hry
ref_soft.dll	Engine pro realizaci softwarového renderování
ref_gl.dll	Engine pro realizaci renderování pomocí knihovny OpenGL
gamex86.dll	Hlavní herní engine

Řízená verze s názvem Quake II .NET obsahuje dodatečný soubor, jenž nabízí podporu rozšíření hry ve formě zabudovaného herního radaru:

Tab. 11.2 Dodatečný soubor řízené verze (Quake II .NET)

SOUBOR	POPIS
Radar.dll	Herní rozšíření napsané v řízeném C++

Mapy, příšery, zbraně a další důležité prvky hry jsou uloženy v datovém souboru (často se jedná o jeden soubor s extenzí PAK) ve složce **baseq2**. Data pro multiplayer (hru více hráčů) jsou uložena v adresáři **base2\players**.

JAK SPUSTIT QUAKE II .NET

Společnost Vertigo Software, Inc. poskytuje pět výše uvedených souborů. Nicméně, pro spuštění hry Quake II .NET budete potřebovat ještě jeden soubor, a sice pak0.pak. Soubor s extenzí PAK obsahuje sbírku trojrozměrných modelů a grafických dat, která je vlastnictvím společnosti id Software a na kterou se vztahují platná autorská práva. Tento soubor však můžete od uvedené společnosti získat.

- **Poznámka**: *„Všechny datové soubory počítačové hry Quake II zůstávají i nadále licencovány a chráněny platnými autorskými právy, což znamená, že nesmíte data z těchto souborů vyjímat a opětovně je distribuovat.“* – John Carmack, id Software (úryvek z textového souboru readme.txt).

Příslušný PAK soubor můžete získat prostřednictvím oficiální demoverze hry Quake II. Co tedy musíte udělat?

1. Nainstalujte hru Quake II z odpovídajícího MSI souboru, který získáte na adrese *www.vertigosoftware.com/quake2*. Tento instalační balíček zabezpečí instalaci souborů pro nativní i řízenou verzi hry.

2. Stáhněte si a otevřete komprimovaný soubor, jenž obsahuje soubory demoverze hry Quake II. Tento soubor můžete najít na adrese *ftp://ftp.idsoftware.com/idstuff/quake2/q2-314-demo-x86.exe*. Po spuštění dekomprimačního procesu budou všechny potřebné soubory rozbaleny a umístěny do výchozí složky (výchozí složkou je c:\windows\desktop\Quake2 Demo).

3. Zkopírujte soubor **pak0.pak** ze složky **Quake2 Demo\Install\Data\baseq2** do složky **%ProgramFiles%\Quake II .NET\managed\baseq2** a také do adresáře **%ProgramFiles%\Quake II .NET\native\baseq2**. Tento soubor je poměrně veliký – jeho kapacita je kolem 48 MB, navíc musíte udělat dvě kopie. Pokud později odinstalujete hru Quake II .NET z vašeho počítače, budete muset manuálně odstranit i právě vytvořené kopie souboru pak0.pak.

4. Nativní, respektive řízenou verzi hry spustíte prostřednictvím příslušného zástupce v nabídce **Start**.

JAK SESTAVIT KÓD

Sestavení kódu je relativně jednoduché, ovšem bude nutno zkopírovat vygenerovaný spustitelný soubor (EXE) a dynamicky linkovanou knihovnu (DLL) do běhového prostředí ještě před spuštěním aplikace. Postupujte podle následujícího algoritmu:

1. Rozpakujte zdrojový ZIP soubor s kódem hry Quake II .NET. Komprimovaný soubor obsahuje kód enginu hry, jenž byl přenesen do prostředí Visual C++® .NET 2003.

2. Otevřete soubor **quake2.sln**.

3. Vyberte cílovou konfiguraci (**Release** nebo **Debug**, **Native** nebo **Managed**) a sestavte kód řešení. Soubory jsou vygenerovány podle specifikované konfigurace (kupříkladu **Release Managed**).

4. Zkopírujte soubory enginu hry ze zdrojové lokace do instalační složky hry Quake II .NET (výchozí složka je %ProgramFiles%\Quake II .NET). Následující tabulka demonstruje, které soubory máte kopírovat:

Tab. 11.3

SOUBOR	KOPÍROVAT DO
quake2.exe	\Program Files\Quake II .NET\
ref_soft.dll	\Program Files\Quake II .NET\
ref_gl.dll	\Program Files\Quake II .NET\
gamex86.dll	\Program Files\Quake II .NET\baseq2
Radar.dll	\Program Files\Quake II .NET\ (soubor je vyžadován jenom pro řízenou verzi hry)

JAK JSME PŘENÁŠELI KÓD

Zdrojový kód jsme získali z výše uvedené adresy FTP serveru společnosti id Software. Tento kód obsahoval soubor pracovního řešení pro Visual Studio 6 s názvem **quake2.dsw**. Když otevřete tento soubor, Visual Studio vás vyzve k aktualizaci projektových souborů a následně vygeneruje soubor řešení s názvem **quake2.sln**. Dále jsme provedli tyto úpravy:

- Kód specifický pro platformu by odstraněn. Byly odstraněny soubory assembly a deaktivovány vložené (inline) rutiny assembly.

- Projektové konfigurace byly upraveny tak, aby zahrnovaly nastavení **Debug Managed**, **Debug Native**, **Release Managed** a **Release Native**.

- Všechny soubory měli aktivovanou volbu **Compile as C code** (**/TC**). Místo toho, abychom opatřovali zdrojové soubory příponou CPP, použili jsme volbu **Compile as C++ code** (**/TP**).

Je-li prostředí náležitě nakonfigurováno, je čas pro zahájení portace zdrojového kódu do C++.

PORTACE KÓDU DO NATIVNÍHO C++

Když jsme přenášeli kód z C do C++, setkali jsme se s některými problémy, které na následujících řádcích blíže popisujeme.

Klíčová slova

Jazyk C++ si vyhrazuje pro interní použití klíčová slova, která nepatří do skupiny vyhrazených klíčových slov jazyka C. Kupříkladu, v kódu hry je používána proměnná s názvem **new**, která byla pro potřebí svého působení v jazyce C++ přejmenována na **new_cpp**.

```
// C
qboolean new;

// C++
qboolean new_cpp;
```

Quake II kód dále definuje svůj vlastní typ **boolean** pomocí hodnot **true** a **false**. Tato klíčová slova jsou však v C++ rezervována, takže jsme pro typ **bool** použili adekvátní typedef.

```
// C
typedef enum {false, true} qboolean;

// C++
typedef bool qboolean;
```

Přetypování

Jazyk C++ pracuje s datovými typy velice svědomitě, což znamená, že při přiřazení hodnot, nebo při provádění operací s funkčními argumenty je vyžadována shoda datových typů, jinak dochází k přetypování. Přetypování sehrálo v procesu přenosu kódu rozhodující část. Ačkoliv jde o poněkud únavnou práci, šla snadno od ruky, nakolik kompilátor je vaším spolehlivým průvodcem a upozorňuje vás na přesný výskyt té-které chyby (identifikován je nejenom soubor s chybou, ale i konkrétní řádek a požadovaná konverzní operace).

```
// C
pmenuhnd_t* hnd = malloc(sizeof(*hnd));

// C++
pmenuhnd_t* hnd = (pmenuhnd_t*)malloc(sizeof(*hnd));
```

Kód hry používá funkci **GetProcAddress** pro dynamické získávání adres funkcí v jiných dynamicky linkovaných knihovnách. Při všech voláních bylo nutné provádět přetypování, a věžte, že oněch volání bylo opravdu dost. Vytvořili jsme speciální skript, který četl protokol sestavování a opatřoval zdrojový kód příslušnými konverzními příkazy. Ukázku přetypování můžete vidět níže.

```
// C
qwglSwapBuffers = GetProcAddress (
  glw_state.hinstOpenGL, "wglSwapBuffers" );

// C++
  qwglSwapBuffers = (BOOL (__stdcall *)(HDC)) GetProcAddress (
  glw_state.hinstOpenGL, "wglSwapBuffers" );
```

Vzhledem k tomu, že jazyk C nevyžaduje, aby deklarace striktně odpovídaly definicím, nebo deklaracím v jiných zdrojových souborech, kompilátor hlásil chyby: **C2371** (opětovná definice, odlišné základní typy) a **C2556** (přetížené funkce se liší jenom typem návratové hodnoty). Abychom vše uvedli na pravou míru, upravili jsme deklarace a definice funkcí. Kupříkladu, níže uvedená deklarace funkce byla modifikována tak, aby vracela hodnotu typu **rserr_t** místo hodnoty typu **int**.

```
// C
int GLimp_SetMode( int *pwidth, int *pheight,
  int mode, qboolean fullscreen );

// C++
rserr_t GLimp_SetMode( int *pwidth, int *pheight,
  int mode, qboolean fullscreen );
```

Jestliže bylo v deklaraci funkce, která byla definována v jiném souboru, použito klíčové slovo **extern**, tato chyba nebyla zachycena dřív, než v okamžiku propojování a jevila se jako nevyřešená externalita.

Použití COM objektů

Volací konvence pro COM rozhraní jsou mezi jazyky C a C++ rozdílné, což je způsobeno skutečností, že virtuální funkční tabulky (neboli vtables) jsou podporovány v jazyce C++. Z hlediska jazyka C je virtuální funkční tabulka COM rozhraní explicitně přístupná a jako první argument je předáván „ukazatel this“. Následuje ukázka, která provádí volání metody **Unlock** objektu COM.

```
// C
sww_state.lpddsOffScreenBuffer->lpVtbl->Unlock(
  sww_state.lpddsOffScreenBuffer, vid.buffer );

// C++
sww_state.lpddsOffScreenBuffer->Unlock(
  vid.buffer );
```

PORTACE KÓDU DO ŘÍZENÉHO C++

Běh řízeného kódu je pod správou společného běhového prostředí vývojové platformy .NET Framework. Ačkoliv použití řízeného kódu není povinné, existuje mnoho důvodů, proč byste mu měli dát šanci. Počítačový program, jenž využívá řízený kód prostřednictvím řízených rozšíření pro C++ (Managed Extensions for C++), může vzájemně spolupracovat se společným běhovým prostředím a využívat služby, mezi něž patří například paměťový management, jazyková interoperabilita, bezpečný přístup ke kódu, či automatická kontrola životních cyklů objektů.

Při portaci nativního C++ kódu do řízeného C++ je nutné aktivovat přepínač kompilátoru s názvem /**CLR**, čímž se ihned zapne také podpora programování s řízenými rozšířeními pro C++. V závislosti od charakteru a složitosti vašeho projektu, může být aktivace přepínače /**CLR** to jediné, co budete muset provést. Při přenosu kódu hry Quake II jsme se však střetli s následujícími chybami.

Nekompatibilní přepínače

Přepínače /**clr** a /**YX** (Automatické používání předkompilovaných hlaviček) nejsou vzájemně kompatibilní, a proto jsme přepínač /**XY** v řízeném prostředí deaktivovali.

Problém s natahováním smíšených knihoven DLL

Kód byl přeložen, ovšem všechny projekty, které generovaly knihovny DLL byly opatřeny následujícím varováním:

```
LINK : warning LNK4243: DLL containing objects compiled with /clr is not linked with /
NOENTRY; image may not run correctly
```

Toto varování se objevuje v situaci, kdy řízená knihovna DLL, napsaná v C++, definuje svůj vstupní bod a když linker hlásí, že během natahování knihovny do paměti by mohlo dojít ke vzniku mrtvého bodu (deadlock). Chybu lze ošetřit tímto způsobem:

- Přidáním přepínače /**NOENTRY**
- Přepojením s knihovnou **msvcrt.lib**
- Zahrnutím symbolu reference **DllMainCRTStartup@12**
- Voláním **__crt_dll_initialize** a **__crt_dll_terminate** v knihovně DLL

Podrobnější informace o popsaných tématech můžete nalézt v následujících článcích:
- *PRB: Linker Warnings When You Build Managed Extensions for C++ DLL Projects*
  http://support.microsoft.com/?id=814472
- *Mixed DLL Loading Problem*
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vcconMixedDLLLoadingProblem.asp

Potíže s předem stanovenou deklarací

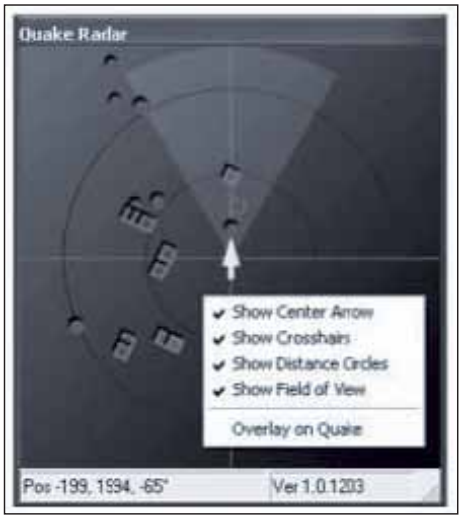
Spustitelný soubor hry Quake II obsahuje předem stanovené deklarace struktur, které jsou definovány v jiných knihovnách DLL. Kompilátor Visual C++ ovšem při pokusu o vysílání nevyhnutných metadat pro této struktury selže a vzápětí je (za běhu programu) generována chybová výjimka **System.TypeLoadException**. Tato výjimka říká, že cílová struktura nemohla být v dané assembly nalezena.

Popsaná situaci se objevuje u struktur **image_s** a **model_s** a můžeme ji vyřešit tak, že budeme struktury definovat v hlavní spustitelné assembly. Více informací naleznete na adrese www.winterdom.com/mcppfaq/archives/000262.htm.

```
// in cl_parse.c
// empty definitions for structs that are forward declared
// this causes the compiler to emit the proper metadata
// and not throw a System.TypeLoadException exception
struct image_s {};
struct model_s {};
```

Rozšiřování hry Quake II

Nyní, když nám Quake II běží uvnitř společného běhového prostředí platformy .NET Framework, můžeme pomýšlet o rozšíření stávajícího kódu. Přidáme tedy skutečně nový programový rys, jenž připravíme výhradně v řízeném prostředí. Při pohledu na současné počítačové hry (jako je třeba Halo), jsme se rozhodli přidat pohotový herní radar, který bude zobrazovat nepřítele, artefakty a jiné zajímavé objekty z ptačí perspektivy (obr. 11.2).



Obr. 11.2: Novinka v Quake II .NET – herní radar

Radarové rozšíření bylo vytvořeno v řízeném C++. Přirozeně, srdcem implementované technologie je řízená třída, která využívá vlastnosti grafického subsystému GDI+ platformy .NET Framework. Tak můžeme používat šipky, grafické štětce s gradientními vzory, antialiasing či průhlednost a neprůhlednost oken. Položky radaru jsou rotovány kolem středu okna pomocí metody **RotateAt** třídy **Matrix**. Použití této metody je daleko snazší, než propočítávání pozic všech položek prostřednictvím trigonometrických funkcí. Rovněž jsme začlenili i podporu kontextové nabídky, která umožňuje zapnout, respektive vypnout některé programové rysy (kupříkladu jde o přítomnost vizuálních položek a zaměřovacího kříže či zorného pole).

Poslední položka kontextové nabídky (**Overlay on Quake**) umísťuje radar přímo do hry. Radar tak překrývá obrazovku, je zbaven stavového řádku a rámu okna a disponuje průhledným pozadím, což způsobuje, že je vhodně zakomponován do samotného herního prostředí. Abychom nezapomněli, jsou také zvětšeny rozměry radaru. Podobu herního radaru v tomto režimu můžete vidět na obr. 11.3.

Obr. 11.3: Integrace herního radaru do hry

Položky, které se objevují na radaru, jsou uchovávány v seznamu třídy **vector** z knihovny STL. Následující výpis kódu ukazuje, jak lze použít iterátor (typedef **iterator**) pro procházení seznamem položek a pro jejich vykreslování na radaru.

```
// draw each item in the list
ItemVector::iterator i;
for (i = m_items->begin(); i != m_items->end(); i++)
{
  // calculate location on radar
  rc.X = (int)center.X +
    ((*i).x/Const::Scale) - (Const::MonsterSize/2);

  rc.Y = (int)center.Y –
    ((*i).y/Const::Scale) - (Const::MonsterSize/2);

  switch ((*i).type)
  {
    case RadarTypeHealth:
    g->FillRectangle(Brushes::Green, rc);
    break;

  case RadarTypeMonster:
   g->FillEllipse(Brushes::Firebrick, rc);
 break;

   . . .
  }
}
```

Když jsme však použili třídu **vector** z knihovny STL, objevilo se několik chyb při sestavení projektu.

Chyba kompilátoru C3633

Prvním zásadním problémem byla chyba kompilátoru, která se objevila při přidávání datového členu třídy **std::vector** do řízené třídy.

```
private __gc class RadarForm : public System::Windows::Forms::Form
{
. . .
private:
  std::vector<RadarItem> m_items;
. . .
};

\quake2-3.21\Radar\RadarForm.h(92): error C3633: cannot define
'm_items' as a member of managed 'Radar::RadarForm'
```

Tato chyba dává najevo, že není možné definovat proměnnou **m_items** jako člena řízené třídy **RadarForm**, protože třída **std::vector** obsahuje kopírovací konstruktor. Situaci jsme vyřešili pomocí ukazatele na seznam položek.

```
private __gc class RadarForm : public System::Windows::Forms::Form
{
. . .
private:
  std::vector<RadarItem>* m_items;
. . .
};
```

Chyba kompilátoru C3377 a C3635

Další nejasnosti vznikly při předávání neřízeného typu v řízeném kódu. Kód hry předává řízenému kódu ukazatel na instanci třídy **std::vector** pro aktualizaci radaru. Jak můžete vidět, oheň byl rychle na střeše.

```
// update method in the radar extension class
static void Update(int x, int y, float angle,
  std::vector<RadarItem>* items)
{
. . .
}

\quake2-3.21\client\cl_ents.c(1612): error C3377:
'Radar::Console::Update' : cannot import method - a parameter type or
the return type is inaccessible

\quake2-3.21\client\cl_main.c(305): error C3635:
'std::vector<RadarItem,std::allocator<RadarItem> >': undefined native
type used in 'Radar::Console'; imported native types must be defined in
the importing source code
```

Problém byl vyřešen definicí prázdné třídy, která byla odvozena od třídy std::vector.

```
__nogc class ItemVector : public std::vector<RadarItem>
{
};

// pass an ItemVector instead of std::vector
static void Update(int x, int y, float angle,
  ItemVector* items)
{
}
```

ZAČLENĚNÍ HERNÍHO RADARU DO HRY QUAKE II

Proces začleňování herního radaru je tvořen třemi klíčovými body: zobrazení radaru, aktualizace radaru a uvědomení radaru při změně pozice okna.

Zobrazení herního radaru

Do slovní zásoby kódu hry Quake II byl přidán nový příkaz s názvem **radar**. Následující fragment programového kódu zabezpečuje zobrazení radaru, jakmile je zadán uvedený příkaz.

```
// check for our new radar command
if (Q_stricmp(cmd, “radar”) == 0)
{
  // toggle the visible state of the radar
  cl_radarvisible = !cl_radarvisible;
  Radar::Console::Display(cl_radarvisible, cl_hwnd);
  return;
}
```

Aktualizace herního radaru

Vyprší-li stanovený časový interval (500 milisekund), je provedena automatická aktualizace herního radaru. Další ukázka kódu předvádí vytvoření seznamu třídy **vector** z knihovny STL, jeho naplnění radarovými položkami a posléze i předání inicializovaného seznamu řízenému rozšíření.

```
void UpdateRadar(frame_t *frame)
{
  // see if enough time has elapsed to update the radar
  static int oldTime;
  int newTime = timeGetTime();
  if (newTime - oldTime < UPDATE_RADAR_MS)
     return;

  // update time so can detect next interval
  oldTime = newTime;

  // store radar items in an STL vector list
  ItemVector* items = new ItemVector();
  RadarItem item;

  // get the players info
  int playernum = cl.playernum+1;
  entity_state_t* player = &cl_entities[playernum].current;

  // loop through list and add items to the radar list
  entity_state_t* s;
  int pnum, num;
  for (pnum = 0 ; pnum<frame->num_entities ; pnum++)
  {
    // get item entity_state
    num = (frame->parse_entities + pnum)&(MAX_PARSE_ENTITIES-1);
    s = &cl_parse_entities[num];

    // make sure this is not the player
    if (s->number != player->number)
    {
```

```
      // add item to the radar list
      item.x = s->origin[0] - player->origin[0];
      item.y = s->origin[1] - player->origin[1];
      item.type = GetRadarType(s);
      items->push_back(item);
    }
  }

  // pass to the radar extension so it can update the display
  Radar::Console::Update(
    player->origin[0], player->origin[1],
    player->angles[1], items);

  // clean up list
  delete items;
}
```

Změna pozice okna

Když je radar zobrazen v překrytém módu, musí vědět, zdali došlo ke změně pozice nebo velikosti okna hry či nikoliv. Kód zpracovává zprávu **WM_WINDOWPOSCHANGED** a předává událost kódu s radarovým rozšířením.

```
case WM_WINDOWPOSCHANGED:
  // pass along to the radar
  Radar::Console::WindowPosChanged(hWnd);
  return DefWindowProc (hWnd, uMsg, wParam, lParam);
```

Makro _MANAGED

C++ kompilátor Visual Studia obsahuje předdefinované specifické makro společnosti Microsoft s názvem **_MANAGED**. Pokud je specifikován přepínač **/clr**, je tohle makro nastaveno na hodnotu 1. Makro se využívá pro zaobalení specifického řízeného kódu.

```
// setting the title of the window
#if _MANAGED
  "Quake II (managed)",
#else
  "Quake II (native)",
#endif
```

VÝKONNOST

Portace existujících projektů do řízeného kódu je určitě smysluplná, protože nabízí mnoho programátorské svobody, kupříkladu:

- Paměťový management můžete ovládat sami, nebo jeho kontrolu přenecháte automatické správě paměti.
- Můžete používat metody bázové knihovny třídy platformy .NET Framework, nebo můžete provádět volání nativního aplikačního programového rozhraní systému Windows.
- Můžete používat třídy bázové knihovny tříd platformy .NET Framework, nebo můžete pracovat s existujícími knihovnami (jako je třeba knihovna STL).

Zcela jistě se shodneme na tom, že o všech uvedených užitečných možnostech můžeme hovořit jenom v případě, když je řízená aplikace tak výkonná, jak má být. V případě počítačové hry Quake II .NET je situace poměrně příznivá: Realizované testy prokázaly, že řízená verze běží jenom o 15 procent pomaleji než její nativní protějšek. Výkonnost řízené verze je tedy přijatelná a ani testeři nezaznamenali výrazné výkonnostní rozdíly mezi oběma verzemi. Pokud budete chtít spustit test v podobě běhu ukázkového módu (dema), proveďte následující kroky:

1. Stisknutím klávesy se znakem **tilda** (~) vyvolejte příkazovou konzoli.

2. Jestliže máte právě rozehranou hru, zadejte do konzole příkaz **disconnect**. Nachází-li se hra v ukázkovém módu, zadání tohoto příkazu není nezbytné.

3. Zadejte příkaz **timedemo 1** a stiskněte klávesu ENTER.

4. Opětovným stisknutím klávesy se znakem **tilda** (~) zavřete příkazovou konzoli. Záhy bude zahájen test, v rámci kterého bude měřen průměrný počet vykreslených snímků za vteřinu.

5. Pro zastavení testu stiskněte klávesu se znakem **tilda** (~). Zjištěný výsledek testu se objeví v příkazové konzoli.

6. Test ukončíte zadáním příkazu **timedemo 0**.

ZÁVĚR

Přenos kódu jazyka C do C++ zabral čtyři dny a portace C++ kódu do řízeného prostředí trvala další den. Implementace řízeného rozšíření si vyžádala přibližně dva dny, přičemž stejnou dobu jsme strávili analýzou kódu hry a pochopením integrujících bodů. Celková zkušenost však byla velmi dobrá. Kromě toho jsme cítili, že naše práce je značně produktivní a poměrně hladká. Je skutečně působivé míchat nativní a řízený kód, ovládat paměťový management a používat stávající knihovny stejně jako třídy platformy .NET Framework. A to vše v rámci jedné a té samé aplikace.

12 Přenos Java aplikací na platformu .NET Framework

Microsoft Corporation

Platí pro:
Microsoft .NET

SHRNUTÍ

Tato kapitola prozkoumává pozitiva a výhody, které vývojářům v jazyce Java přináší vývojová platforma Microsoft .NET. Rovněž uvidíte, co obnáší proces portace stávajících Java aplikací do nového prostředí umožňujícího vývoj a běh aplikací .NET.

Z internetové adresy
http://msdn.microsoft.com/vjsharp/using/techinfo/default.aspx?pull=/library/en-us/dndotnet/html/dotnet_movingjavaapps.asp si můžete stáhnout programové ukázky **MigratingJavaApplicationsCSharpDemo.msi** a **MigratingJavaApplicationsJSharpDemo.msi**.

CO ZNAMENÁ .NET?

Microsoft .NET (vyslovováno jako „dot net“) představuje na jedné straně vizi tvorby softwarových aplikací a na straně druhé sadu vývojových nástrojů pro realizaci této vize. Abychom ilustrovali hnací silu vize .NET, podívejme se na běžné problémy obchodních společností, které se úzce dotýkají konektivity a interoperability: Ačkoliv většina obchodních firem spolupracuje s jinými firmami, jejich informační systémy pracují odděleně. Dodavatelské řetězce produktů nejsou napříč prodejci integrovány, a tudíž je komunikační proces mezi dodavatelem a zákazníkem často omezen pouze na výměnu faxů a jednoduchých textových zpráv. Tato skutečnost ovšem představuje výraznou bariéru, která brání maximalizaci produktivity. Když se obchodní operace a činnosti společností stanou více propojenými, mohou zvýšit svou efektivitu a výkonnost. Když budou mezi jednotlivými prodejci v rámci dodavatelského řetězce vybudovány intenzivní komunikační vztahy, budou dosaženy mnohé cíle, jako například minimalizace nákladů pro práci se zásobami, výroba na požádání, koordinace činností a celkově vyšší efektivita.

S problémy se vzájemnou konektivitou úzce souvisí i otázka interoperability. Když totiž i obchodní společnosti přistoupí k zlepšení vzájemných vztahů v oblasti komunikace, záhy musí čelit složitým technickým úkolům, které se vztahují k návrhu a implementaci plánovaného sblížení. Navíc, informační systémy, na jejichž bázi se má spolupráce realizovat, se často nacházejí v různých stadiích vývoje, nebo jsou poznačeny neustálými změnami. Jako by toho již tak nebylo dost, společnosti se potýkají také s bezpečnostními potížemi. Prioritou číslo jedna je bezpečnost a ochrana dat, která putují mezi firemními firewally. Dalším potenciálním nebezpečenstvím je neoprávněný přístup k vybraným datům, a také skutečnost, že data se po takovémto přístupu mohou dostat do nepovolaných rukou konkurenčních subjektů.

Hlavní náplní vize .NET je povznést úroveň vzájemné komunikace prostřednictvím lepšího navrhování a vytváření bezpečně propojitelných a spolupracujících systémů kdykoliv, kdekoliv a na jakémkoliv počítačovém zařízení. Stěžejní technologii, pomocí níž je tato vize realizována, představují XML webové služby. Tato technologie vytváří metodologickou i transportní vrstvu pro předávání informací mezi komponenty různých počítačových stanic, rozličných typů sítí a rozmanitých operačních systémů.

Podpora XML webových služeb je široká a stále vzrůstá. Mnoho společností právě v těchto chvílích úspěšně používá XML webové služby pro dosažení efektivní spolupráce s jinými organizacemi. Společnost Microsoft přidává podporu pro XML webové služby do všech svých produktových řad. Z pohledu vývojáře představuje vize .NET výrazné usnadnění při vytváření a propojování informačních systémů pomocí vývojové platformy .NET Framework, softwarové sady Visual Studio .NET a, samozřejmě, XML webových služeb.

.NET FRAMEWORK

Vize .NET ovšem nejsou jenom XML webové služby. Srdcem této myšlenky je vývojový rámec s názvem .NET Framework, jenž pozůstává ze společného běhového prostředí (Common Language Runtime, CLR) a bohaté knihovny tříd (.NET Framework Class Library, FCL). Tyto dvě komponenty zabezpečují běh aplikací a aplikační programová rozhraní pro vývoj aplikací .NET.

Aplikace kompilované pro platformu .NET nejsou překládány přímo do nativního kódu. Místo toho jsou kompilovány do kódu jazyka Microsoft Intermediate Language (MSIL). Když dojde k prvnímu spuštění aplikace .NET, společné běhové prostředí aktivuje Just-In-Time (JIT) překladač, který provede na požádání kompilaci MSIL kódu aplikace do nativního kódu, jenž je poté přímo vykonáván. Nicméně, společné běhové prostředí znamená více, než jenom JIT kompilátor. Toto prostředí je rovněž odpovědné za realizaci exekučních služeb nízké úrovně, mezi něž patří třeba automatická správa paměti, manipulace s výjimkami, bezpečnostní služby a kontrola typové bezpečnosti za běhu programů. Jelikož role společného běhového prostředí je v rámci řízení aplikací více než důležitá, cílové aplikace, které běží na platformě .NET Framework, jsou někdy označovány jako řízené aplikace.

.NET Framework obsahuje kolekci tříd pro vytváření aplikací, jejichž běh spravuje společné běhové prostředí. Všechny třídy jsou seskupeny v hierarchicky uspořádané knihovně tříd, která nabízí podporu pro realizaci široké škály úkolů: přístup k datům, bezpečnost, vstupně-výstupné operace při práci se soubory, manipulace s XML, zpracování zpráv, reflexe, XML webové služby, ASP.NET aplikace a služby Windows.

Pravděpodobně nejbáječnějším atributem vize .NET je podpora interoperability více programovacích jazyků. Společnost Microsoft připravila nabídku čtyř komerčních programovacích jazyků, směrujících k platformě .NET Framework. Jedná se o Visual Basic .NET, Visual C# .NET, řízená rozšíření pro C++ (Managed Extensions for C++) a Visual J#™ .NET. Existují ovšem také další .NET-kompatibilní programovací jazyky, jako třeba Perl, Python a COBOL.

Aby mohly programovací jazyky .NET společně pracovat ve vývojovém prostředí .NET Frameworku, musejí vyhovovat jistým standardům. Tyto standardy byly vytvořeny společností Microsoft a jsou soustředěny v takzvané společné jazykové specifikaci (Common Language Specification, CLS). CLS deklarativně říká, jakým kriteriím musí každý programovací jazyk vyhovovat, aby mohl být použit na platformě .NET Framework společným běhovým prostředím a aby mohl spolupracovat se softwarovými komponentami, které byly vytvořeny v jiných programovacích jazycích. Pokud jazyk implementuje nezbytnou funkcionalitu, je označován jako .NET-kompatibilní. Každý .NET-kompatibilní programovací jazyk pracuje se shodnými datovými typy, používá stejné třídy platformy .NET Framework, jeho aplikace jsou kompilovány do stejného MSIL kódu a jsou řízeny společným běhovým prostředím. Z tohoto důvodu je každý .NET-kompatibilní programovací jazyk prvotřídním obyvatelem městečka .NET. Vývojáři si nyní mohou vybrat programovací jazyk, jenž nejlépe vyhovuje jejich požadavkům pro vývoj softwarových komponent, aniž by byli jakkoliv ochuzeni v oblasti výkonu či dostupnosti rysů vývojové platformy. Nejlepší ovšem je, že komponenty napsané v jednom jazyce mohou snadno spolupracovat s komponentou, která byla připravena v jiném programovacím jazyce .NET. Kupříkladu můžete napsat třídu v jazyce C#, která bude odvozena od mateřské třídy připravené ve Visual Basicu. Aby mohli i další vývojáři programovacích jazyků připravit .NET-kompatibilní verze svých produktů, byla koncepce společné jazykové specifikace předložena společností Microsoft standardizační komisi ECMA. V době psaní tohoto textu bylo ve vývoji přes dvacet .NET-kompatibilních programovacích jazyků.

Tab. 12.1 zobrazuje rozličné komponenty platformy .NET Framework, které vytvářejí softwarovou vrstvu nad operačním systémem. Jde o komponenty, které jsou psány *kurzivou* (ASP.NET, Windows Forms, ADO.NET, XML a další subkomponenty).

Tab. 12.1 Architektura vývojové platformy .NET Framework

Visual Basic	Visual C++	Visual C#	Visual JScript	Visual J#
Společné jazyková specifikace (Common Language Specification)				
ASP.NET *Web Forms / Webové služby*			*Windows Forms*	
ADO.NET a XML				
Bázová knihovna tříd (Base Class Library, BCL)				
Společné běhové prostředí (Common Language Runtime, CLR)				
Operační systém				

VISUAL STUDIO .NET

Abyste mohli vyvíjet své aplikace pro platformu .NET, společnost Microsoft přepracovala své vývojářské nástroje a poskládala je do jednotné sady, která se nazývá Visual Studio .NET. Pomocí Visual Studia .NET můžete vytvářet mnoho typů aplikací v jednom, nebo třeba i v několika programovacích jazycích. Visual Studio .NET disponuje integrovaným vývojovým prostředím, které slouží pro návrh, vývoj, ladění a rozmísťování aplikací, a které všechny .NET-programovací jazyky společně sdílejí.

Spektrum potenciálních aplikací je vskutku široké: Můžete vyvíjet konzolové aplikace, standardní aplikace pro systém Windows, dynamicky linkované knihovny (DLL), webové aplikace, XML webové služby a aplikace pro počítače do dlaně. Visual Studio .NET nabízí několik jedinečných rysů, které v notné míře zvyšují produktivitu vaší práce: Jde o technologii IntelliSense, vizuální návrháře pro Web Forms a Windows Forms, schémata XML, datová schémata, výkonný odlaďovač aplikací (jenž je schopen pracovat s kódem vícerých programovacích jazyků), dále je to těsná integrace s platformou .NET Framework, dynamická nápověda (která vám nabízí témata podle stylu vaší práce), okno **Task List** (zobrazuje chyby kompilátoru a úkoly, které je zapotřebí provést), integrace s návrhářským softwarem jako je Visio, okno **Server Explorer** pro vizuální přístup k databázím, služby Windows, měřiče výkonnosti a vývoj komponent, které běží na straně serveru.

Kolekce vývojářských nástrojů Visual Studio .NET je natolik propracovaná, že není divu, že se záhy po svém představení dočkala několika významných ocenění. Za všechna vzpomeňme alespoň dvě ocenění za přínos v oblasti produktivity od *2002 SD Magazine* a cenu *Technical Excellence od PC Magazine 2001* (toto ocenění bylo uděleno ještě betaverzi produktu).

POROVNÁNÍ .NET A JAVY

Zkušeným Java vývojářům se může platforma .NET Framework jevit jako velmi podobná platformě, na níž pracuje samotná Java. Obě uvedené platformy poskytují sofistikovaný přístup k vývoji aplikací, obě disponují jazyky, jejichž kód je kompilován do jisté jazykové mezivrstvy a obě nabízejí rozsáhlou knihovnu aplikačních programových rozhraní pro vývoj softwaru. Nicméně, vize .NET ve svém jádru ukrývá odlišnou sadu cílů, než je tomu u Javy.

Z pojmového hlediska je Java tvořena dvěma nosnými pilíři: Java platformou (běhové prostředí a API) a programovacím jazykem Java. Hlavním účelem Java platformy je veškerá softwarová podpora aplikací napsaných v jazyce Java a kompilovaných do bajtového kódu Javy. I když existovaly pokusy, jejichž cílem bylo kompilovat také výstupy jiných programovacích jazyků do bajtového kódu Javy, jednalo se veskrze o akademická cvičení. Ideálem pro Javu byl vždy jeden programovací jazyk, jenž mohl působit na mnoha různých platformách.

Pokud ještě na chvíli zůstaneme ve světě pojmů, můžeme říct, že koncepce .NET je rovněž složena ze dvou primárních komponent: Jde o .NET Framework (běhové prostředí a API) a široká množina podporovaných programovacích jazyků. Cílem platformy .NET Framework je podporovat aplikace napsané v jakémkoliv .NET-kompatibilním programovacím jazyce, jehož výstupy jsou kompilovány do kódu jazyka MSIL. Ideálem pro koncepci .NET je tedy jediná platforma, kterou lze sdílet mnoha různými programovacími jazyky.

VÝHODY PLATFORMY .NET FRAMEWORK OPROTI JAVĚ

Vedle svobody při výběru vhodného programovacího jazyka nabízí platforma .NET Framework ještě další výhodné rysy, které u Javy nenajdete. Některé nejvýznamnější jsou dále charakterizovány.

Třídy platformy .NET Framework

Vzhledem k tomu, že Java je multiplatformní, kolekce použitelných tříd byla vždy limitována společnou množinou programových rysů, jimiž disponovaly všechny podporované platformy. Tato skutečnost ovlivňovala jednak rámec použitelnosti tříd a také různorodost těchto tříd. Rámec použitelnosti tříd byl omezen společným jmenovatelem, jenž byl tvořen kolekcí dostupných počítačových platforem a operačních systémů. Tím trpěla také různorodost tříd, která nebyla tak vyspělá, jak by bylo zapotřebí. Vestavěné třídy jsou totiž velice jednoduché, což nutilo vývojáře velice často vyvíjet své vlastní třídy a řešit tak funkcionalitu, která nebyla implicitně implementována. Další možností bylo zakoupení kolekcí tříd u firem třetích stran, což ovšem také nebylo to pravé ořechové. Zářným příkladem může být práce s XML. Jestliže jste chtěli blíže prozkoumávat XML dokumenty před vydáním JDK 1.4, museli jste buď použít knihovnu třetí strany, nebo si vyvinout vlastní řešení, které nezřídka vyžadovalo více než sto řádků programového kódu.

.NET Framework je založen na platformě systému Windows. Speciální verze společného běhového prostředí a platformy .NET Framework jsou rovněž přístupné i pro systém FreeBSD. Zabudované třídy nabízejí plnou podporu všech rysů, které jsou na dané platformě dostupné. Navíc, zdejší třídy jsou bohatší než ty, se kterými pracujete v Javě. Jednoduše řečeno, platforma .NET Framework nabízí vše, co budete potřebovat, a tudíž nejste nuceni vyvíjet svá vlastní řešení pro realizaci běžných programátorských úkolů.

Následující fragment programového kódu jazyka C# předvádí, jak lze pracovat s XML dokumenty v prostředí platformy .NET Framework. Všimněte si, že všechny pracovní třídy jsou poskytovány samotnou platformou. Proto je psaní kódu až neuvěřitelně snadné:

```
XmlTextReader myXmlTextReader = new XmlTextReader (“textToValidate.xml”);
XmlValidatingReader myReader = new XmlValidatingReader(myXmlTextReader);
myReader.Schemas.Add(myXmlSchemaCollection);
myReader.ValidationEventHandler += new ValidationEventHandler (this.ValidationFailureHandler;
// Read XML data.
while (myReader.Read()){}
```

Věrnost platformy .NET Framework je vyšší, než je tomu na Java platformě. Všechny aplikace .NET mohou využívat plnou sílu systému Windows a nejsou tedy omezovány společnou množinou „průnikových“ funkcí všech podporovaných platforem, což je přesně případ Javy.

„... opravdovým rozhodovacím faktorem pro práci na platformě .NET Framework je skutečnost, že společnost Microsoft poskytuje všechny standardizované a kompatibilní nástroje, které bude potřebovat 90 procent všech vývojářů, zatímco programování v Javě ještě stále znamená vytváření řešení z různých kousků, které musíte získat z mnoha zdrojů.“ [1]

Napiš jednou, odlaďuj všude

Ačkoliv je rčení Javy „napiš jednou, spusť všude“ zcela jistě atraktivní, drtivá většina dnešních aplikací se zaměřuje pouze na jeden operační systém. Pro takovéto počínání existují dva technické důvody: Za prvé, Java platforma je omezena ve směru své použitelnosti i různorodosti, což má za následek, že vývojáři často používají pro přístup k rysům cílové platformy soukromé knihovny. Za druhé, přetrvávající prvky nekompatibility mezi různými implementacemi Javy přispívají ke vzniku potíží při snaze o vývoj skutečného multiplatformního řešení. Vývojářům proto nezbývá nic jiného, než testovat svůj vlastní kód na každé platformě, kterou hodlají používat. Zejména uvedené důvody jsou příčinou toho, že někteří programátoři začali původní rčení Javy „napiš jednou, spusť všude“ obměňovat ekvivalentem „napiš jednou, odlaďuj všude“.

Bezpečnost

Podpora bezpečnosti na platformě .NET Framework je bohatší. Vedle aplikačních programových rozhraní pro podporu kryptografie se v tomto prostředí můžete setkat také s bezpečnými cookies či autentikací. Dokonce můžete využít konfigurovatelný mechanizmus pro přiřazování bezpečnostních povolení různých úrovní různým skupinám uživatelů na základě původu programového kódu. Kupříkladu, veškerý kód, jenž pochází od společnosti Microsoft, může být považován za bezpečný. Na druhou stranu, programový kód z jistého URL může, případně nemusí, získat povolení pro přístup ke specifickým zdrojům, jako je třeba složka pro ukládání dočasných souborů či obrazovka počítače. Navíc, aplikace mohou být „označkovány“ tak, že budou vyžadovat jistá povolení (například pro přístup k souborovému systému). Jestliže nejsou stanovená povolení k dispozici, aplikace nebude kompletně zavedena do paměti počítače. Vzhledem k tomu, že kontrola bezpečnostních povolení je uskutečňována ihned po spuštění aplikace, vývojáři nemusí trávit čas psaním programové logiky pro vyhledávání bezpečnostních selhání. Tato činnost je řízena prostřednictvím nové technologie s názvem Bezpečný přístup ke kódu, která pracuje s kódem načteným buď z Internetu, nebo z lokální počítačové stanice.

1 Bornstein, Niel. May 2002. Pull Parsing in C# and Java, *www.xml.com/pub/a/2002/05/22/parsing.html?page=1*.

Práce s verzemi

.NET Framework disponuje robustnějším systémem správy verzí než Java. Pokud jsou na Java platformě instalovány dvě verze stejné třídy, běhové prostředí Javy jednoduše načte jako první tu třídu, kterou jako první objeví. Verze třídy je přitom zcela ignorována. Platforma .NET Framework se však správě verzí věnuje velice pečlivě. Pomocí paralelní exekuce (side-by-side execution) je možné, aby byly dvě nebo i víceré verze stejné třídy načteny a zpracovány ve stejný okamžik. Každá aplikace si přitom může vybrat svoji variantu propojovacího mechanizmu: Aplikace může navázat spojení se specifickou verzí knihovny tříd, s nejaktuálnější verzí knihovny, nebo s verzí knihovny, která se nachází v určité složce. Navíc, politika správy verzí může být aplikována na bázi konkrétního počítače, a tak můžete ještě víc vypilovat kontrolu verzí. Pozornost věnovaná správě verzí na platformě se automaticky přenáší i do programovacích jazyků. Kupříkladu v jazyce C# jsou metody standardně deklarovány jako nevirtuální a pokud tedy vývojáři chtějí pracovat s virtuálními metodami, musejí je explicitně deklarovat. To tedy znamená, že komponenty nemohou nechtěně narušit chování bázových tříd. Jazyk C# nevyžaduje po vývojáři explicitní deklaraci výjimek, které může metoda generovat. Jestliže se změní seznam výjimek, kód na straně klienta bude i nadále pracovat správně.

Výkonnost

Platforma .NET jednoznačně zastiňuje Javu ve schopnosti ovlivňování chování a výkonnosti aplikace. Hlavní stavební bloky aplikace jsou navrženy s ohledem na škálovatelnost a vysokou výkonnost. Příkladem je třeba nepravidelný přístup k datům pomocí ADO.NET, jehož pomocí může být více operací s daty prováděných v paměti, což v prostředí Java Database Connectivity (JDBC) není realizovatelné a samotná databáze musí být pořád kontaktována. Již samotný návrh architektury aplikací .NET je předurčuje k tomu, aby byl jejich běh rychlejší. Kupříkladu, ASP.NET webové stránky jsou na tom z hlediska výkonnosti mnohem lépe, než řešení vytvořená pomocí technologií ASP (Active Server Pages), JSP (Java Server Pages) a skriptovacích jazyků, které tvoří součást HTML kódu webové stránky. Porovnáváme-li obě platformy (.NET a Java), musíme také říct, že aplikace .NET si vedou lépe nejenom při práci na jednom počítači, nýbrž i v síťovém prostředí pracujícím podle modelu klient/server.

„Aplikace .NET podávají zřetelně lepší výkon než Java aplikace při vyřizování požadavků bez použití vyrovnávací paměti, jsou víc než desetkrát rychlejší při obsluhování 5000 virtuálních uživatelů a při použití vyrovnávací paměti běží více než dvakrát svižněji ve srovnání se svými Java protějšky.“ [2]

JAK PŘENÉST JAVA APLIKACE DO .NET

Pokud se rozhodnete přenést své stávající aplikace napsané v jazyce Java, můžete uvažovat o následujících alternativách: Můžete aplikace inovovat v prostředí programovacího jazyka Visual J# .NET, nebo je můžete portovat do prostředí jazyka Visual C# .NET. Každá alternativa si vyžaduje jiný přístup a každá má své výhody, jak uvidíte dále. Obě varianty však znamenají přenos vaší aplikace do prostředí operačního systému Windows, nakolik plná verze NDP je k dispozici pouze na této platformě. Ačkoliv NDP obsahuje kompilátor příkazového řádku pro Visual C# .NET, Visual Basic .NET a Visual J# .NET, v dalším textu budeme předpokládat, že vývojáři přenášejí své aplikace pomocí Visual Studia .NET. Jak již bylo zmíněno, Microsoft Visual Studio .NET je nejproduktivnější sada nástrojů pro vývoj aplikací .NET.

Inovace na Visual J# .NET

Visual J# .NET je implementací jazyka Java pro vývojovou platformu .NET Framework. Tento jazyk je pro předplatitele MSDN k dispozici na internetové adrese *http://msdn.microsoft.com/vjsharp/default.aspx*.

Programovací jazyk J# byl vyvinut nezávisle společností Microsoft pro potřeby platformy .NET Framework, a tudíž jsou výstupy tohoto jazyka kompilovány do kódu jazyka MSIL. Visual J# .NET je tvořen jazykovým kompilátorem, knihovnami tříd JDK verze 1.1.4 a pomůckou, která si poradí s inovací bajtového kódu stávajících Java aplikací do podoby MSIL kódu (což je užitečné zejména při inovování rozsáhlých knihoven do formy, která může být použita aplikacemi .NET). Kromě toho, že vývojáři mohou využívat možností knihoven tříd JDK verze 1.1.4, mohou, podobně jako programátoři v jiných .NET-kompatibilních jazycích, beze všeho pracovat i s bázovou knihovnou tříd platformy .NET Framework. Rovněž mohou používat i vizuální návrháře aplikací pro Windows a web.

2 VeriTest, May 2002. VeriTest® Benchmark and Performance Testing Microsoft .NET Pet Shop, *www.godotnet.com/team/compare/veritest.aspx*.

Inovace na Visual J# .NET představuje nejjednodušší a nejrychlejší cestu pro přenos stávajících Java aplikací na platformu .NET Framework. Výborné je, že vývojáři mohou ihned pracovat v .NET prostředí, k čemuž jim napomáhají nejenom stávající zkušenosti s programovacím jazykem Java, ale i známá syntax a kolekce knihoven tříd. Inovací aplikací pomocí Visual J# .NET dosáhnete efektu okamžitého účinku: Inovace je opravdu rychlá a ihned po její realizaci mohou vývojáři přikročit k programování dalších vlastností, či k využití jazykových rozšíření, které provedla společnost Microsoft. Když ve Visual Studiu .NET otevřete projekt jazyka Visual J++® 6.0, spustí se průvodce inovací s názvem **Java to Visual J# .NET Upgrade Wizard**.

Inovace na Visual C# .NET

Konverzní pomůcku s názvem **Java Language Conversion Assistant** si můžete stáhnout ze stejnojmenného webového sídla na adrese *http://msdn.microsoft.com/vstudio/downloads/tools/jlca/*. Tato aplikace vám pomůže s konverzí Java aplikací do prostředí programovacího jazyka Visual C# .NET. Programový kód jazyka Java, jenž provádí volání do Java API je konvertován do podoby ekvivalentního kódu jazyka C#, který pro svou práci používá .NET Framework. Jelikož je konverzní proces poněkud komplikovaný, ne všechna aplikační programová rozhraní jsou konvertována. Pomůcka **Java Language Conversion Assistant** uskuteční převod devadesáti procent kódu, jenž provádí volání do knihoven tříd JDK verze 1.1.4. Oněch 10 procent pak může představovat kód, který si bude vyžadovat pečlivější pozornost. Pokud se objeví potíže související s převodem, zobrazí se hypertextové odkazy, které vás navedou na témata, v nichž se dozvíte, jak úspěšně dokončit konverzní proces.

Aplikace konvertované do C# se vyznačují značnou flexibilitou: Ačkoliv je převod do tohoto jazyka, ve srovnání s inovací na Visual J# .NET, o něco pomalejší, nabízí více příležitostí, protože přeložené aplikace jsou okamžitě schopné využívat API .NET Frameworku. Po uskutečnění nezbytných úprav je aplikace ihned obdařena mnoha výhodami, které plynou z lepší výkonnosti, škálovatelnosti, zabezpečení a správy verzí (podobně jako tomu je i ve Visual J# .NET).

Na rozdíl od Visual J# .NET, pomůcka **Java Language Conversion Assistant** podporuje konverzi projektů Visual J++ 6.0, konverzi samostatných souborů a konverzi projektů uložených ve složkách. Průvodce konverzí spustíte tak, že otevřete nabídku **File**, ukážete na položku **Open** a klepnete na položku **Convert Project**.

VOLBA MEZI PROGRAMOVACÍMI JAZYKY J# A C#

Rozhodnutí, který z uvedených .NET-kompatibilních jazyků si zvolit při migraci Java aplikací, se odvíjí zejména od osobních preferencí vývojáře. Jak bylo řečeno, výstupy obou programovacích jazyků (J# a C#) jsou překládány do MSIL kódu, oba jazyky zprostředkovávají stejný přístup k bázové knihovně tříd platformy .NET Framework a oba disponují relativně podobnými dovednostmi.

Abychom vám pomohli učinit rozhodnutí, uvádíme tři hlavní faktory, které byste měli mít na paměti při volbě mezi Visual J# .NET a Visual C# .NET:

Čas potřebný pro migraci	Když je Java aplikace inovována v prostředí Visual J# .NET, ze syntaktického hlediska se pořád jedná o Javu, přičemž všechna volání funkcí API knihovny JDK verze 1.1.4 jsou ponechány tak, jak jsou. Je-li stejná aplikace konvertována do prostředí jazyka C#, kód je převeden do syntaxe jazyka C# a volání do Java API jsou převedeny na volání do bázových tříd platformy .NET Framework. Kvalita takovéto konverze je velmi vysoká (nezřídka přes 90 procent), ovšem je zapotřebí provést jisté modifikace zdrojového kódu. Pokud je hlavním kriteriem minimalizace času potřebného pro migraci, Visual J# .NET je nejlepší volbou.
Framework	Jakmile jsou Java aplikace konvertovány do prostředí jazyka C#, vývojáři okamžitě získávají všechny výhody platformy .NET Framework, což ve stručnosti znamená lepší výkonnost, škálovatelnost, zabezpečení a správu verzí. Aplikace, které byly inovovány pomocí Visual J# .NET mohou i nadále používat datové typy a knihovny jazyka Java, i když i ony mohou implementovat novou funkcionalitu platformy .NET Framework. Pokud chcete vaši aplikaci přenést z důvodů využití výhod platformy .NET Framework, poté je jazyk C# pro vás tou pravou volbou.
Jazyk	Pro mnoho lidí je nejdůležitějším faktorem rozhodování jednoduše samotný programovací jazyk: Nyní tedy řešíme otázku, zdali byste raději své aplikace vyvíjeli pomocí Javy nebo jazyka C#. V tomto směru vám nemůžeme nijak pomoci, protože vy sami nejlépe víte, který z uvedených programovacích jazyků je vaším favoritem. Nicméně, tak Visual J# .NET jako i C# jsou odvozeny od C++ a disponují podobnou syntaxí a programovými rysy. Visual J# .NET bude nejspíše vhodnou variantou pro programátory, kteří investovali hodně prostředků do jazyka Java a kteří hodlají tento jazyk používat i nadále. Na druhou stranu, s jazykem C# budou zcela jistě spokojeni ti vývojáři, kteří by rádi vsadili na "nativní" .NET-kompatibilní programovací jazyk určený výhradně pro vývoj aplikací .NET.

Pro lepší porozumění každé z předestřených voleb si v následující části ukážeme, co obnáší přenos aplikace na platformu .NET Framework.

MAPOVÁNÍ TECHNOLOGIE

Každá migrační cesta vás dovede k poněkud jiným výsledkům. Následující dvě tabulky ukazují, co se stane s Java projekty při jejich přenosu na platformu .NET Framework.

Tab. 12.2: Migrace Java komponent do .NET

JAVA TECHNOLOGIE	INOVACE NA VISUAL J# .NET	KONVERZE POMOCÍ C#
Programovací jazyk Java	Programovací jazyk Java	Programovací jazyk C#
Applet	Nekonvertováno	Ovládací prvek Windows Forms
JavaBean	JavaBean	Třída jazyka C# (BeanInfo a ClassInfo nejsou konvertovány)
Ovládací prvek ActiveX	Nekonvertováno	Ovládací prvek ActiveX
Rámec AWT	Rámec AWT	Formulář Windows
Formulář WFC	Formulář WFC	Formulář Windows
Kompilovaná knihovna	Kompilovaná knihovna	Nekonvertováno
Soubor zdrojů	Soubor ResX	Soubor ResX

Tab. 12.3: Migrace pro individuální programové balíky

PROGRAMOVÝ BALÍK	INOVACE NA VISUAL J# .NET	KONVERZE POMOCÍ C#
com.ms.awt java.awt	Java.awt	System.Windows.Forms
com.ms.com	com.ms.com	System.Runtime.InteropServices
com.ms.dll	com.ms.dll	System.Runtime.InteropServices System.ComponentModel
com.ms.dxmedia	Neinovováno	DirectAnimation
com.ms.fx	Neinovováno	System.Windows.Forms.
com.ms.io	com.ms.vjsharp.io	System.IO
com.ms.lang	com.ms.lang	System Microsoft.Win32.RegistryKey
com.ms.object	com.ms.vjsharp.object	System
com.ms.ui	Neinovováno	System.Windows.Forms
com.ms.util	com.ms.util	System System.Collections System.Diagnostics
com.ms.wfc.app	com.ms.wfc.app	System.Windows.Forms System.Globalization.CultureInfo Microsoft.Win32 System.Environment.SpecialFolder System.Threading System.DateTime
com.ms.wfc.core	com.ms.wfc.core	System System.ComponentModel System.Windows.Forms.Design System.ComponentModel.Design System.Resources
com.ms.wfc.data	com.ms.wfc.data	ADODB System.Runtime.InteropServices RDS System Globalization System System.ComponentModel MSDASC System.Resources
com.ms.wfc.data.adodb	com.ms.wfc.data.adodb	ADODB
com.ms.wfc.data.ui	com.ms.wfc.data.ui	System.Windows.Forms System.Data System
com.ms.wfc.io	com.ms.wfc.io	System.IO System.Globalization
com.ms.wfc.ole32	com.ms.wfc.ole32	
com.ms.wfc.ui	com.ms.wfc.ui	System.Windows.Forms

Tab. 12.3: Migrace pro individuální programové balíky

PROGRAMOVÝ BALÍK	INOVACE NA VISUAL J# .NET	KONVERZE POMOCÍ C#
com.ms.wfc.util	com.ms.wfc.util	System System.Diagnostics System.Collections System.Runtime.InteropServices System.Resources System.Globalization
com.ms.wfc.win32	com.ms.wfc.win32 com.ms.win32	Konvertováno na volání P/Invoke
java.io	java.io	System.IO
java.lang	java.lang	System System.Threading
java.lang.reflect	java.lang.reflect	System.Reflection System
java.math	java.math	System.Decimal
java.net	java.net	System.Net System System.IO
java.security	java.security	Nekonvertováno
java.sql	java.sql	System.Data.OleDb System.DateTime
java.text	java.text	System System.Globalization System.Resources
java.text.resources	java.text.resources	System.Resources
java.util	java.util	System System.Collections System.Globalization System.Resources System.Configuration

MIGRAČNÍ MIX

Komponenty napsané v různých .NET-kompatibilních programovacích jazycích mohou bez jakýkoliv potíží mezi sebou spolupracovat. Je tedy možné využít i migrační mix, v rámci kterého provedete inovaci jedné komponenty do prostředí jazyka Visual J# .NET, zatímco jinou komponentu převedete do C#. Praktickým příkladem může být třeba inovace komponenty střední vrstvy do Visual J# .NET, a poté můžete k této komponentě přistupovat prostřednictvím klienta, který byl konvertován do jazyka C#. Pomocí binárního konvertoru jazyka Visual J# .NET (**Visual J# .NET Binary Converter**) můžete vytvářet přeložené verze knihoven, které budou použitelné v prostředí kteréhokoliv .NET-kompatibilního programovacího jazyka.

Technologie bez automatické migrace

Programovací jazyk Visual J# .NET a konverzní průvodce **Java Language Conversion Assistant** se zaměřují na jazyk Java, knihovnu JDK verze 1.1.4 a knihovny společnosti Microsoft, které byly vydány společně s produktem Visual J++ 6.0. Některé aplikace mohou ovšem používat technologie aktuálnějších verzí Javy, jako je třeba J2EE a J2SE. Pokud se rozhodnete provést portaci takovýchto aplikací do prostředí .NET, budete muset provést jisté dodatečné úpravy. Tento proces je normálně docela jednoduchý: Po ukončení hlavní části migračního procesu by měla být převedena většina programového kódu aplikace (jedná se zejména o obchodní logiku a třídy knihovny JDK verze 1.1.4). Programový kód nepodporovaných technologií bude v převedené aplikaci ponechán v původní verzi. Abyste završili migrační proces, budete muset vybrané partie „nepodporovaného“ kódu nahradit ekvivalentním .NET-kompatibilním kódem. To by ovšem vzhledem k tomu, že platforma .NET Framework obsahuje bohatou nabídku tříd, neměl být větší problém. Navíc můžete uskutečnit i některá technologická vylepšení.

Proces migrace si předvedeme na následující ukázkové aplikaci.

Migrace ukázkové aplikace

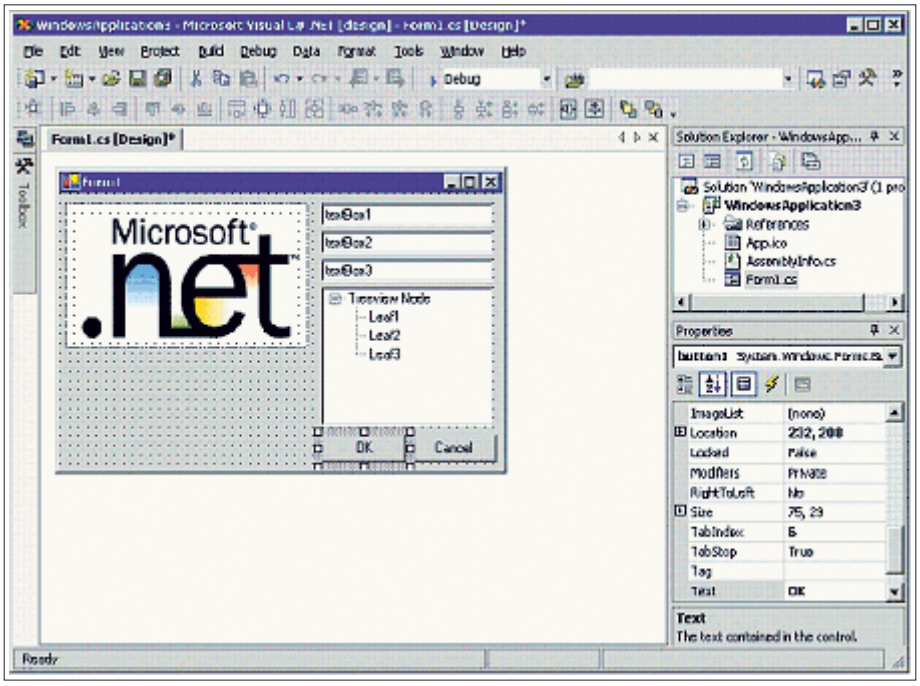
Java obsahuje dva protichůdné balíčky formulářů: Abstract Windowing Toolkit (AWT) a Swing. AWT je dostupné v knihovně JDK verze 1.1.4, je podporované jazykem Visual J# .NET a může být konvertováno konverzním asistentem **Java Language Conversion Assistant** do podoby formulářů Windows (Windows Forms). Přídavná knihovna Swing byla k dispozici v knihovně JDK verze 1.1.4 a je rovněž součástí knihoven tříd J2SE a J2EE. Formuláře knihovny Swing nejsou do prostředí platformy .NET Framework automaticky převedeny. Během migračního procesu jsou třídy **Javax.Swing** ponechány nezměněny ve zdrojovém kódu a měly by být nahrazeny třídami Windows Forms. Tento cíl je možné realizovat pomocí následujícího postupu, který můžete vykonat pro každý formulář knihovny Swing, jenž se nachází ve vaší aplikaci:

1. Do převáděné aplikace přidejte nový formulář.

2. Na formulář umístěte instance ovládacích prvků tak, aby byl upravený vzhled formuláře podobný původnímu Swing formuláři.

3. Zkopírujte kód zpracovatelů událostí z původního formuláře do těl příslušných zpracovatelů událostí nového formuláře.

4. Odstraňte Swing formulář z vaší aplikace.



Obr. 12.1: Opětovný návrh původního Swing formuláře

Knihovna Windows Forms nabízí daleko širší prostředí pro vývoj formulářů než AWT nebo Swing a ve všech ohledech převyšuje funkcionalitu Java platformy. Kupříkladu, pomocí Windows Forms lze snadněji ovládat rozvržení formulářů. Správci rozvržení formulářů v Javě bývají častým zdrojem nesnází. Abyste dosáhli kýženého vzhledu a stylu chování formuláře, musíte nejenom dobře porozumět správcům rozvržení, ale také napsat množství kódu (dokonce i pro realizaci základních operací). Knihovna Windows Forms usnadňuje rozvržení instancí ovládacích prvků pomocí absolutního umísťování. Rovněž můžete aplikovat pokročilé techniky ukotvení či zarovnání instancí ovládacích prvků. Pomocí zarovnání nařídíte, aby se instance automaticky umísťovaly podle vybrané oblasti formuláře. Instance můžete také ukotvit: V tomto případě bude jejich vzdálenost od stran formuláře konstantní, což je užitečné zejména v okamžiku, kdy uživatel začne experimentovat s velikostí formuláře. Sečteno a podtrženo, pomocí zarovnávání a ukotvování instancí ovládacích prvků si můžete ušetřit psaní dodatečného programového kódu.

Z uvedeného je zřejmé, že migrace Swing formulářů na platformu .NET je nejenom reálná, ovšem skýtá řadu dalších potenciálních vylepšení, kterými můžete nové formuláře opatřit.

Migrace Java Server Pages

Java Server Pages (JSP) a servlety představují technologie, které napomáhají vytváření webových stránek pomocí jazyka Java. JSP poskytuje soustavu skriptů, které jsou vkládány do HTML kódu webových stránek, a její použití je podobné technologii ASP. Servlety nabízejí mechanizmus pro běh kompilovaných JSP stránek bez HTML kódu. Obě technologie zaměstnávají kód jazyka Java a obě nabízejí použitelný model programování pomocí událostí, jenž lze využít při stavbě aktivních webových stránek.

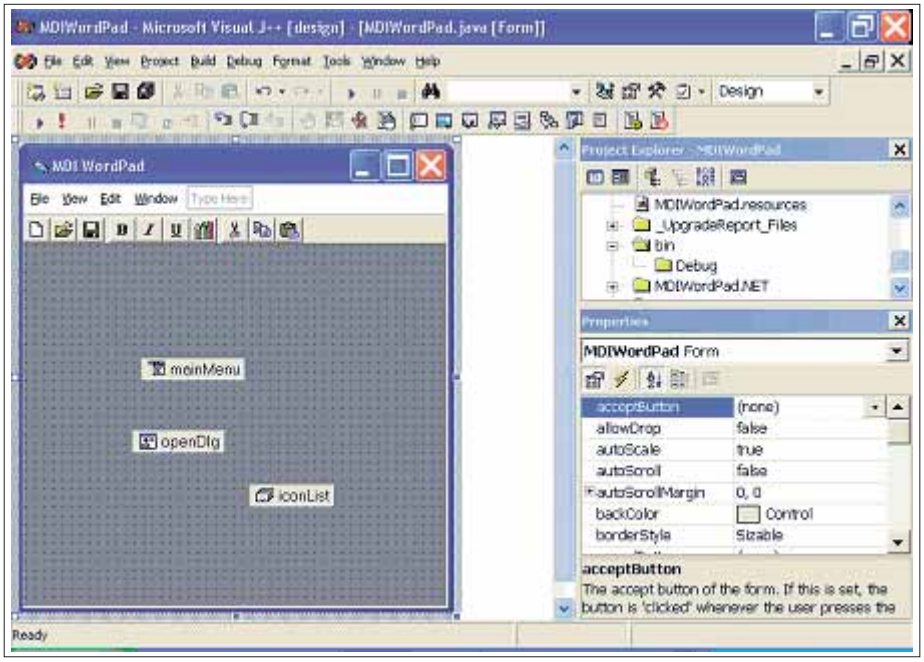
JSP webové stránky i servlety můžete konvertovat do prostředí ASP.NET, dokonce existuje několik pádných důvodů, proč byste tak měli udělat. Technologie ASP.NET je robustnější, je obdařena vyšší výkonností a lepší škálovatelností než JSP, PHP (PHP Hypertext Preprocessor), ASP a jiné technologie pro správu interpretovaných skriptů, které tvoří součást HTML kódu. Měli byste ovšem vědět, že v době psaní tohoto textu prozatím nebyl vytvořen žádný automatizovaný systém, který by dovedl usnadnit přenos JSP stránek do ASP.NET. Nicméně, společnost Microsoft oznámila, že vytvoří rozšíření pomůcky **Java Language Convert Assistant**, které by si mělo poradit i s konverzí JSP stránek do prostředí jazyka C# a ASP.NET (pomůcka měla být k dispozici ve druhé polovině roku 2002). Implementace konverzního průvodce pro migraci JSP webových stránek do Visual J# .NET by se měla objevit v budoucí verzi sady programátorských nástrojů Visual Studio .NET.

UKÁZKY MIGRAČNÍCH SCÉNÁŘŮ

V této podkapitole prozkoumáme, jak uskutečnit inovaci aplikací z Visual J++ 6.0 do Visual J# .NET a konverzi z Javy do C#. Hlavním cílem je tedy představení obou migračních technik. Ještě předtím, než začnete číst následující řádky, byste ovšem měli vědět, že možnosti každého z analyzovaných migračních scénářů jsou ve skutečnosti o něco širší, než jaké jsou zde prezentovány. Ať tak či onak, migraci kterékoliv z uvedených ukázkových aplikací můžete uskutečnit pomocí jakékoliv zde vysvětlované migrační techniky.

Jak provést inovaci projektu Visual J++ 6.0 do prostředí Visual J# .NET

Naše ukázková aplikace, kterou budeme inovovat, se nazývá **MDIWordPad**. Tato aplikace byla dodávána společně s produktem Microsoft Visual J++ 6.0 a můžete jí získat také na stránce produktových ukázek jazyka Visual J++ 6.0, jejíž adresa je *http://msdn.microsoft.com/vjsharp/productinfo/visualj/visualj6/downloads/samples/default.aspx*. Aplikace MDIWordPad představuje textový procesor a pracuje stejně jako aplikace WordPad, kterou můžete znát z operačního systému Windows. Aplikace umí otevírat a editovat textové dokumenty, používat základní formátování a je opatřena základní nabídkou a panelem nástrojů. Grafické uživatelské prostředí ukázkové aplikace MDIWordPad vytvořené ve Visual J++ je znázorněno na obr. 12.2.



Obr. 12.2: Ukázková aplikace MDIWordPad v prostředí Visual J++ 6.0

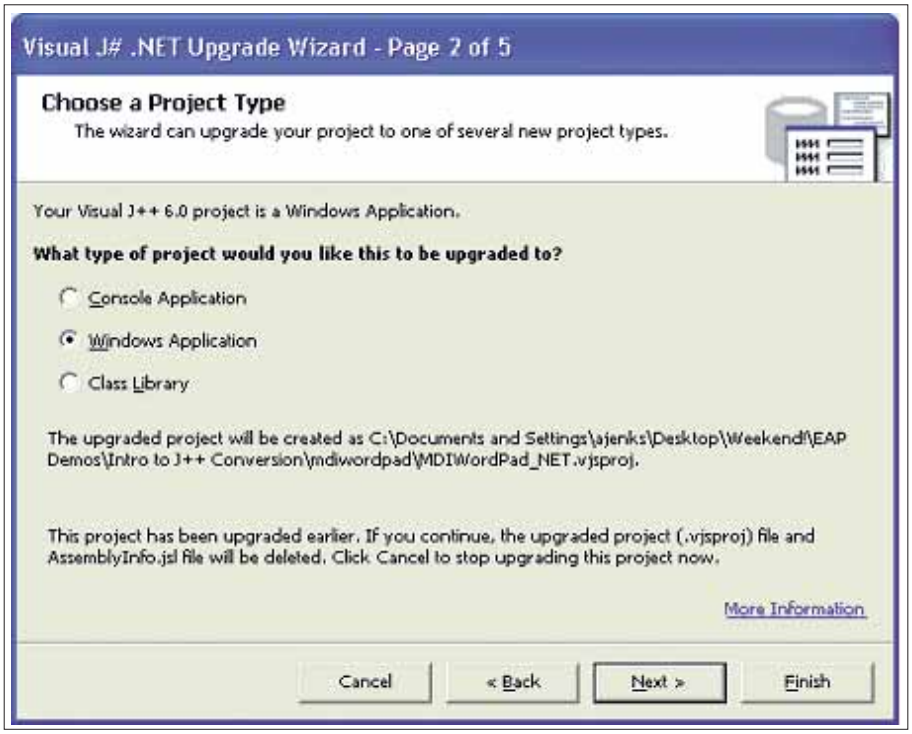
Abyste získali nejlepší možné výsledky, doporučuje se projekty Visual J++ 6.0 inovovat pomocí Průvodce inovací (**Upgrade Wizard**) nástroje Visual J# .NET. Postupujte podle následujícího návodu:

1. V prostředí Visual Studia .NET otevřete nabídku **File**, ukažte na položku **Open** a klepněte na položku **Project**.

2. Vyhledejte projektový soubor **MDIWordPad.vjp** a stiskněte tlačítko **Open**.

   Visual Studio .NET zjistí, že projekt, jenž hodláte otevřít, je projektem jazyka Visual J++ 6.0, a proto iniciuje spuštění Průvodce inovací (**Upgrade Wizard**).

3. Když si přečtete úvodní textové informace, můžete klepnout na tlačítko **Next**, čímž se dostanete na druhou stránku průvodce. Na této stránce budete muset zadat informace o typu projektu, který chcete inovovat. Jelikož aplikace MDIWordPad je určena pro systém Windows, můžete ponechat výchozí nastavení. Kdyby ovšem byla inovovanou aplikací třeba konzolová aplikace nebo knihovna tříd, museli byste zvolit příslušnou volbu. Jakmile jste ujištěni, že je vybrána možnost **Windows Application**, můžete aktivovat tlačítko **Next**, nacož se přesunete na třetí stránku Průvodce inovací (**Upgrade Wizard**).



Obr. 12.3: Průvodce inovací produktu Visual J# .NET sbírá informace o inovované aplikaci

1. V případě aplikací, které používají ovládací prvky ActiveX nebo odkazy na jiné COM komponenty, budete moci na této stránce přidat příslušné reference na externí součásti. Pomocí těchto informací bude Průvodce inovací schopen lokalizovat další datové typy, s nimiž v aplikaci pracujete. Přestože tyto reference budou nalezeny a přidány i pokud je nebudete výslovně specifikovat, jejich přidání v této chvíli vám umožní rychleji projít inovačním procesem. Naše ukázková aplikace nepoužívá žádné externí komponenty, a tudíž můžete směle klepnout na tlačítko **Next**, aniž byste museli provádět jakékoliv další změny.

2. Průvodce inovací začne zpracovávat váš aplikační projekt, přidávat nezbytné odkazy a postupně jej inovovat pro použití v jazyce Visual J# .NET. O pokroku těchto činností vás informují textové zprávy a stavový pruh. Je-li inovační proces u konce, Průvodce inovací vás automaticky přenese na finální stránku.

3. Na poslední stránce můžete určit, zdali si přejete prohlédnout Výkaz o inovaci vašeho projektu. Pro zobrazení tohoto výkazu klikněte na tlačítko **Finish**.

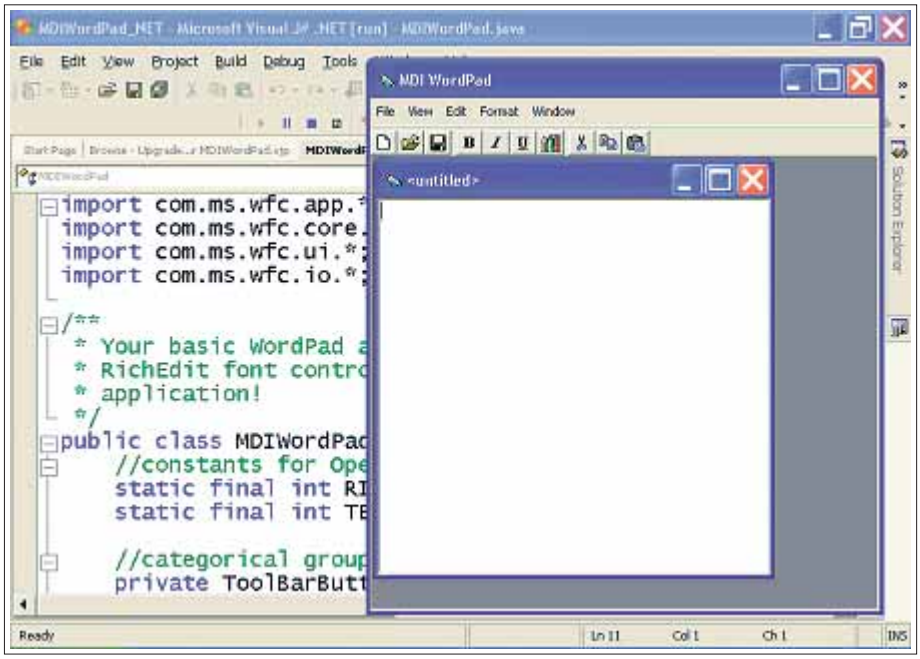
   Výkaz o inovaci obsahuje informace o vaší aplikaci a procesu její inovace. Pokud byla v tomto procesu nalezena nějaká problémová místa nebo chybějící odkazy, budou vám tyto skutečnosti sděleny. Ve výkazu uvidíte také informace o nově přidaných souborech, které byly do stávajícího aplikačního projektu začleněny Průvodcem inovací.

4. V okně **Solution Explorer** poklepejte na soubor **MDIWordPad.java**, což způsobí otevření editoru pro zápis programového kódu.

   Přehlédnete-li si otevřený soubor, uvidíte, že zdrojový kód vypadá přesně tak jako dříve. Inovace projektu do prostředí Visual J# .NET nevyžaduje ve skutečnosti žádné úpravy zdrojového kódu. Jediné, co je nutné upravit, je řešení, projekt a soubory zdrojů. Programový kód ale zůstává beze změny.

5. Inovovanou aplikaci spustíte stisknutím klávesy **F5**.

   Chování aplikace se nezměnilo, o čem se můžete přesvědčit sami tak, že zapíšete několik řádků textu. Vskutku, aplikace běží a pracuje stejně jako předtím, ovšem nyní již v prostředí platformy .NET Framework.



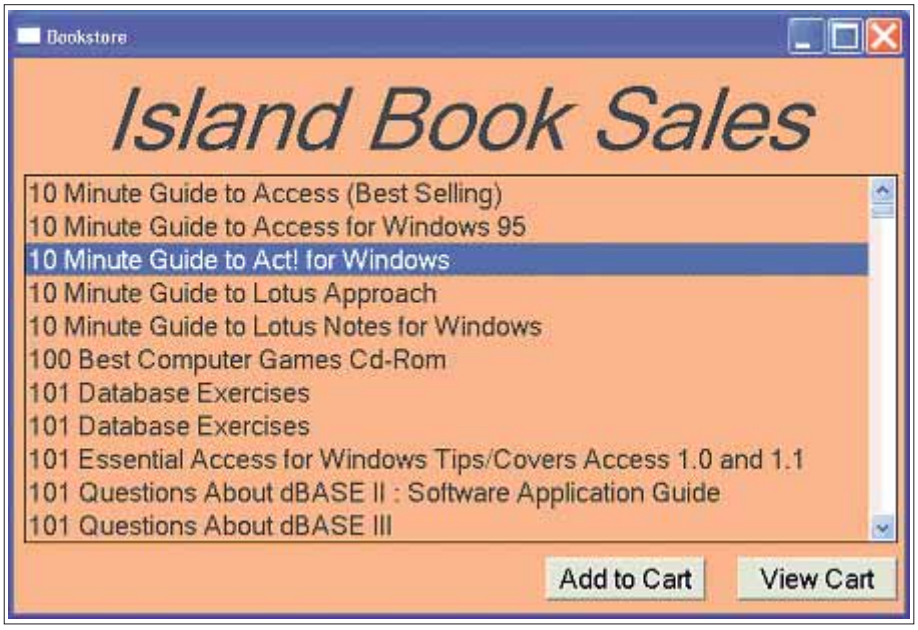
*Obr. 12.4: Inovovaná verze aplikace **MDIWordPad**, připravená ve Visual Studiu .NET a běžící na platformě .NET Framework*

Jak můžete vidět, inovace aplikace do prostředí jazyka Visual J# .NET je docela snadná. Se všemi nezbytnými úkoly vám pomůže speciální průvodce a dokonce ani není zapotřebí upravovat již napsaný programový kód. Vzhledem k tomu, že Visual J# .NET je jazyk takřka shodný s jazykem Java, použití tohoto nástroje je nejvhodnější v případě, kdy potřebujete rychle a snadno portovat aplikační projekty připravené pomocí Visual J++.

Jak konvertovat Java projekt do prostředí jazyka C#

V této sekci uvidíte, jak provést konverzi aplikace napsané v jazyce Visual J++ 6.0 do jazyka C# stylem krok za krokem. Ukázková aplikace, která bude podrobena konverzi, představuje jednoduchou nákupní aplikaci a jmenuje se Island Book Sales. Na příkladu této aplikace si vysvětlíme různé aspekty konverze (konverze AWT do Windows Forms, konverze JDBC do ADO.NET a konverze Javy do C#). Nezapomeneme však ani na některá problémová místa, která bude zapotřebí opravit po úspěšném dokončení konverzního procesu. Naše ukázková aplikace pozůstává ze dvou formulářů. Na prvním formuláři mohou uživatelé najít seznam knížek, které si mohou zakoupit. Informace o všech dostupných knižních titulech jsou uloženy v databázi programu Access. Potřebné informace jsou získávány pomocí JDBC. Uživatel si může ze seznamu vybrat knižní položku a přidat ji do nákupního košíku. Nákupní košík uživatele je znázorněn druhým formulářem. Tento formulář umožňuje uživateli odstraňovat knižní položky z nákupního košíku. Obr. 12.5 přibližuje rozhraní běžící aplikace.

- **Poznámka**: Abyste mohli spustit ukázkovou aplikaci, musíte zaregistrovat databázi bookstore.mdb jako ODBC DSN.



Obr. 12.5: Ukázková aplikace Island Book Sales

Konverze ukázkové aplikace do prostředí programovacího jazyka C# probíhá podle následujících kroků:

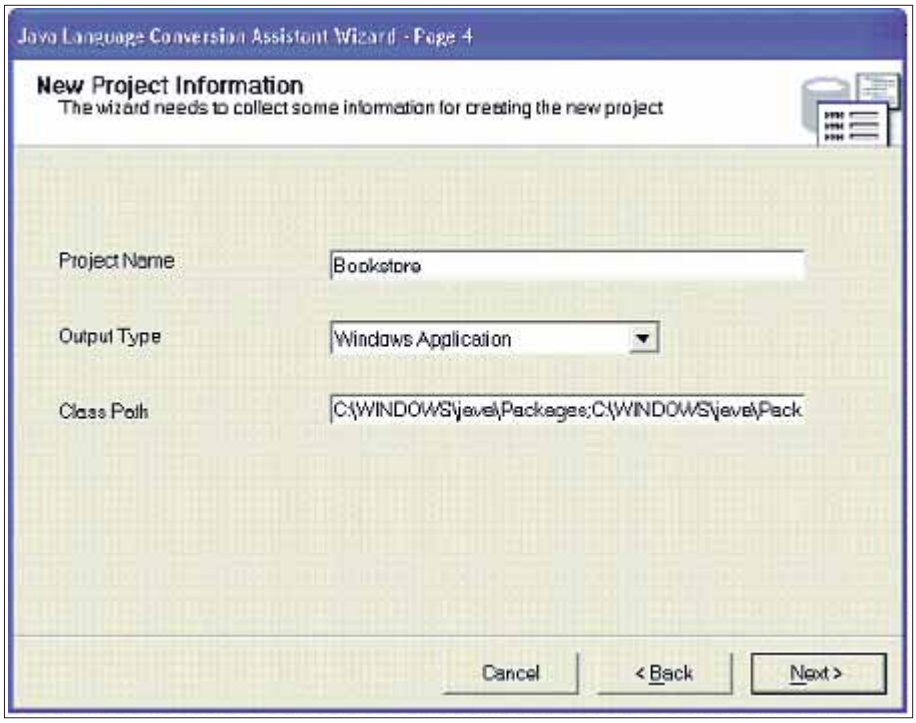
1. Ve Visual Studiu .NET otevřete nabídku **File**, ukažte na položku **Open** a poté klepněte na položku **Convert**. Objeví se dialogové okno **Convert**.

2. Vyberte ikonu, která reprezentuje pomůcku **Java Language Conversion Assistant** a klepněte na tlačítko **OK**. Tím spustíte konverzního průvodce.

3. Ačkoliv byla původní aplikace napsána ve Visual J++, budeme používat konverzi založenou na složkách. Na druhé stránce průvodce vyberte možnost **A directory containing the project´s files** a aktivujte tlačítko **Next**.

4. Na třetí stránce průvodce vyberte složku, ve které jsou uloženy projektové soubory aplikace, a poté klepněte na tlačítko **Next**.

5. Na čtvrté stránce vás průvodce požádá o zadání relevantních informací o převáděné aplikaci. Aplikaci pojmenujte jako **Bookstore**, přičemž ostatní volby můžete ponechat ve výchozím stavu. Jste-li hotovi, klikněte na tlačítko **Next**.

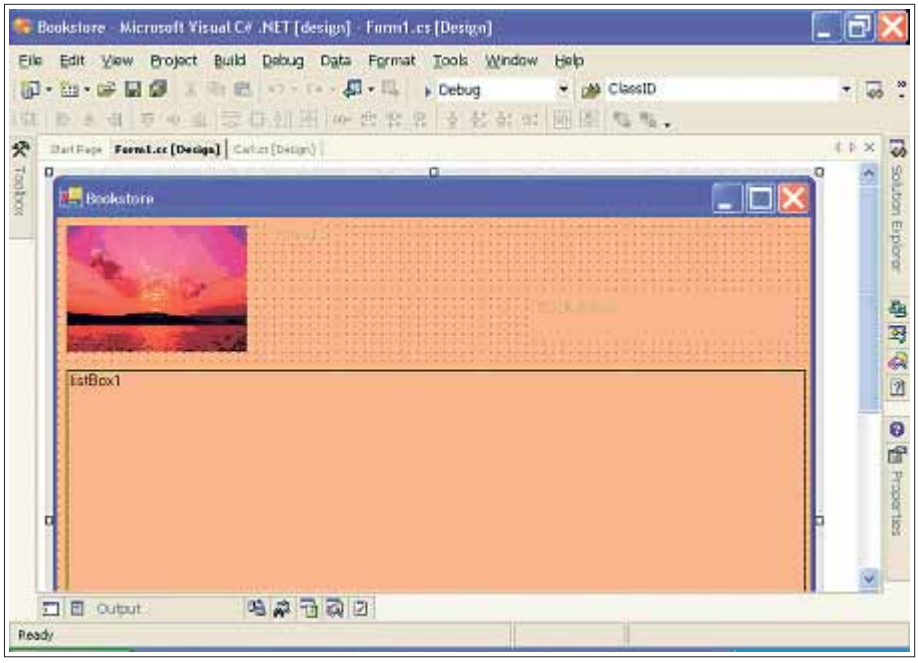


Obr. 12.6: Zadávání informací o projektu

1. Průvodce **Java Language Conversion Assistant** se vás nyní zeptá, kde by měly být uloženy konvertované projektové soubory. Určete vhodný adresář a klepněte na tlačítko **Next**. Průvodce uloží soubory a případně také vytvoří novou složku, bude-li to zapotřebí.

2. Pro zahájení konverze stačí, když ještě jednou klepnete na tlačítko **Next**. Po ukončení konverzního procesu uzavřete také okno průvodce.

3. Pomocí klávesové zkratky **SHIFT+F7** otevřete okno vizuálního návrháře formulářů. Všimněte si, že formulář a všechny jeho součásti jsou v návrháři viditelné. Průvodce odvedl skvělou práci, neboť převedl AWT komponenty na komponenty knihovny Windows Forms. Konvertovaný formulář si můžete prohlížet pomocí vizuálního návrháře.



Obr. 12.7: Vzhled konvertovaného formuláře

1. Zobrazte okno **Task List** (příkaz na zobrazení okna se nachází v nabídce **View**). Uvidíte, že okno **Task List** obsahuje několik přidaných komentářů, které souvisejí s konvertovaným souborem. Tyto komentáře poukazují na problémová místa, která musejí být opravena ještě předtím, než budete moci aplikaci řádně spustit. (Pokud provedete sestavení a spuštění aplikace v této chvíli, uvidíte sice formulář, no nebudou k dispozici žádná data).

2. Poklepejte na první komentář v okně **Task List**. IDE přenese kurzor na místo prvního výskytu chyby, přesněji na metodu **retrieveData**. Tato metoda má na starosti dotazování databáze a získávání informací o dostupných knižních titulech, které budou posléze zobrazeny v aplikaci. Po zběžném prohlédnutí kódu metody zjistíte, že volání JDBC bylo nahrazeno voláními ASP.NET. Pro zdárný průběh metody budeme muset provést pár úprav.

3. První komentář **Upgrade_ToDo** v metodě **retrieveData** říká, že formáty parametrů pro metodu **Class.forName** byly změněny. Řádek kódu pod komentářem byl původně použit pro načtení ovladače JDBC, což ovšem není nutné, když pracujeme s ASP.NET. Z tohoto důvodu odstraňte komentář a také řádek s kódem, jenž se nachází pod ním.

4. Další komentář vás informuje o tom, že formát řetězce spojení se v ASP.NET změnil. Kdybyste znali nový formát řetězce spojení, mohli byste jej jednoduše zapsat na příslušné místo. Místo toho, abyste si museli pamatovat formát řetězce spojení, můžete použít nástroje Visual Studia .NET, které vám s tímto úkolem pomohou a vytvoří vhodný řetězec spojení pro vás.

5. Pomocí klávesové zkratky **SHIFT+F7** znovu otevřete okno vizuálního návrháře. Na formulář přetáhněte instanci komponenty **OleDbConnection**, kterou můžete najít na kartě **Data** soupravy nástrojů (**Toolbox**). V okně **Properties** vyhledejte vlastnost **ConnectionString**, klepněte na tlačítko s šipkou a vyberte příkaz **New Connection**. Téměř okamžitě se zobrazí dialogové okno **Data Link Properties**, které vám pomůže specifikovat databázové spojení.

6. Databáze obsahující informace o knížkách byla vytvořena v aplikaci Microsoft Access. Pro čtení dat z databáze Accessu v prostředí ADO.NET se používá zprostředkovatel Microsoft Jet 4.0 OLE DB. V dialogovém okně **Data Link Properties** klepněte na záložku **Provider**, vyberte zmíněného zprostředkovatele a aktivujte tlačítko **Next**. Vyhledejte soubor s databází (má příponu .mdb) a klepněte na tlačítko **OK**.

7. Vlastnost **ConnectionString** nyní obsahuje správně formátovaný řetězec spojení pro vybranou databázi. Zkopírujte tento řetězec a vložte ho do deklaračního příkazu proměnné **URL**, která se nachází v těle metody **retrieveData**. Před řetězec přidejte znak @, což jazyku C# řekne, aby s řetězcem zacházel jako s literálem. Po provedených úpravách by měl deklarační příkaz proměnné **URL** vypadat následovně:

```
String URL = @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source =..\..\db.mdb;";
```

8. Problém je vyřešen, a proto můžete odstranit příslušný konverzní komentář.

9. Následující dva komentáře neovlivňují aplikaci. Tyto komentáře můžete buď ignorovat, nebo je odstranit.

10. Spusťte aplikaci stisknutím klávesy **F5**. Aplikace se rozběhne a vy se můžete přesvědčit, že funguje správně, docela tak jako její původní verze.

Jak jste mohli vidět, dvojvrstvá aplikace byla konvertována z Javy do C# velice hladce a bez citelných problémů. Přestože naše ukázková aplikace byla z těch jednodušších, prezentované techniky a postupy můžete použít při konverzi jakékoliv jiné počítačové aplikace.

ZÁVĚR

Přenos Java aplikací do jazyků Visual J# .NET a C# je snadný, což je způsobeno vestavěnými inteligentními nástroji, které tento proces ve velké míře automatizují. Úspěšně přenesené aplikace mohou okamžitě využívat všech výhod, které skýtá platforma .NET Framework. Tak mohou vývojáři přidávat nové a vzrušující programové rysy, zatímco uživatelé se na druhé straně mohou těšit na výkonnější a spolehlivější aplikace vyhovující .NET standardům.

Vývojáři se mohou rozhodnout, zdali chtějí své stávající aplikace inovovat pomocí Visual J# .NET, nebo je raději konvertovat do C#. Každý přístup disponuje jistým souborem výhod: Inovace na Visual J# .NET je nejrychlejší a nejsnadnější volbou, která nabízí Java aplikacím okamžitý užitek na nové platformě. Konverze do jazyka C# je pomalejší, nicméně nabízí nejvyšší úroveň flexibility. Konečné rozhodnutí leží samozřejmě na vás, ovšem pokud uvážíte zkoumané faktory, bude se vám lépe rozhodovat.

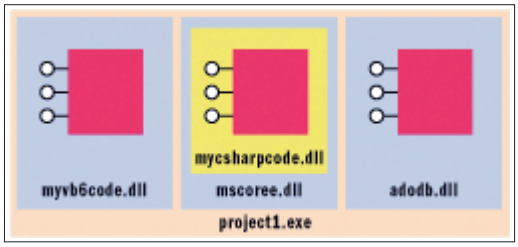
Bez ohledu na to, kterou migrační techniku použijete, můžete si být jisti, že vašim stávajícím aplikacím vdechnete nový život. Sada vývojářských nástrojů Visual Studio .NET a platforma .NET Framework společně představují novou etapu vývoje počítačového softwaru.

13 Přenos nativního kódu do řízeného prostředí platformy .NET Framework

Autor: Don Box, Microsoft Corporation

Navzdory nadějím managementu, akcionářů a vysoce ambiciózních vývojářů není jednoduše možné, aby byl každý blok, každý kousek softwaru napsán prostřednictvím řízeného kódu a spouštěn pomocí společného běhového prostředí platformy .NET Framework (Common Language Runtime, CLR). Vzhledem k nedostatku programátorského talentu není pravděpodobné, že by většina společností kompletně přepisovala svůj programový kód založený na Win32® platformě, jazycích C, C++ a Visual Basicu 6.0 či COM komponentách, a to bez ohledu na to, jak neodolatelně se může nová vývojová platforma .NET Framework jevit. To znamená, že svět nativního kódu, jenž byl vytvořen v „před .NET éře“, musí nerušeně koexistovat s novými řízenými komponentami, které byly vytvořeny .NET-kompatibilními programovacími jazyky. Naštěstí, společné běhové prostředí umožňuje přemostění obou zmíněných světů, aniž by tím nějak výrazně utrpěli samotní vývojáři a programátoři.

Pro zabezpečení soužití nativního a řízeného kódu bylo zapotřebí vyvinout mechanizmus, který by dovoloval na jedné straně běh řízeného kódu uvnitř společného běhového prostředí (MSCOREE.DLL) a na straně druhé exekuci kódu mimo tohoto prostředí. To tedy znamená, že řízené programy by měly být schopny běžet v „neřízeném módu“ tak dlouho, aby dovedly vykonávat programový kód klasických COM komponent nebo Win32 součástí. Tento přístup funguje i naopak: Neřízená programová vlákna by měla disponovat dovedností navázat během jisté doby spojení s knihovnou MSCOREE.DLL pro vyvolání metody specifického řízeného typu.



Obr. 13.1: Integrace řízeného a neřízeného programového kódu

Dobrou zprávou je, že společné běhové prostředí podporuje oba typy přechodů, což umožňuje vzájemnou koexistenci řízených i nativních knihoven DLL v jednom procesu. Rovněž je možné, aby jedna knihovna DLL prováděla volání funkcí jiné knihovny, bez ohledu na to, zdali jsou tyto knihovny hostovány společným běhovým prostředím či nikoliv. Příkladem může být proces, jenž je znázorněn na obr. 13.1. Všimněte si, že jistý programový kód běží uvnitř knihovny MSCOREE.DLL, zatímco jiný ne. Dokud nebude operační systém kompletně přepsán pomocí řízeného kódu, s takovými situacemi se budeme setkávat opravdu často. Důvodem je, že systémy Windows pracují na nativní bázi, která pracuje s neřízenými dynamicky linkovanými knihovnami a ne s jejich řízenými protějšky.

P/INVOKE

Volání funkcí nativní knihovny DLL založené na aplikačním programovém rozhraní Win32 je docela snadné pomocí technologie P/Invoke (P znamená Platform). P/Invoke představuje technologii, která vám umožňuje mapovat deklaraci statické metody na vstupní bod programu formátu PE/COFF, který lze získat voláním funkce **LoadLibrary**/**GetProcAddress**. Podobně jako Java Native Interface (JNI) a J/Direct®, také technologie P/Invoke používá řízenou deklaraci metody pro popsání rámce zásobníku, ovšem předpokládá, že tělo metody bude poskytnuto externí, nativní knihovnou DLL. Na rozdíl od JNI, P/Invoke je užitečná v případě importu „zděděných“ knihoven DLL, které nebyly vytvářeny se zřetelem na existenci společného běhového prostředí.

Skutečnost, že metoda je definována v externí nativní knihovně DLL, dáte na známost tak, že tuto metodu označíte pomocí klíčových slov **static** a **extern** a použijete atribut **System.Runtime.InteropServices.DllImport**. Atribut **DllImport** říká společnému běhovému prostředí, které argumenty je nutné předat funkcím **LoadLibrary** a **GetProcAddress** v okamžiku, když je tato metoda aktivována. Vestavěný atribut **DllImport** jazyka C# je ve skutečnosti pouhým odkazem (aliasem) pro **System.Runtime.InteropServices.DllImport**. Atribut **DllImport** pracuje s různou plejádou parametrů. Jak předvádí následující výpis zdrojového kódu, atribut **DllImport** vyžaduje, aby bylo předáno alespoň jméno cílového souboru, v němž je uložen kód knihovny DLL. Toto jméno je použito běhovým prostředím pro zavolání funkce **LoadLibrary** dříve než dojde k odeslání požadavku na aktivaci metody. Za předpokladu, že atributu **DllImport** nebyl předán parametr **EntryPoint**, bude textový řetězec, který používá funkce **GetProcAddress**, představovat symbolické jméno metody.

```
namespace System.Runtime.InteropServices {
  public class DllImportAttribute : Attribute {
    public DllImportAttribute(String dllname);
// EntryPoint overrides method name for GetProcAddress
 public String EntryPoint;
// ExactSpelling disables W/A _@ mangling
 public bool ExactSpelling;
// cdecl, stdcall, etc
 public CallingConvention Calling Convention;
// unicode/ansi/auto
 public CharSet CharSet;
// function calls SetLastError
 public bool SetLastError;
// treat native result as an HRESULT a la [retval]
 public bool TransformSig;
 }
}
```

Další fragment programového kódu demonstruje dva způsoby volání metody **Sleep** z knihovny kernel32.dll. První ukázka se spoléhá na jméno funkce jazyka C#, které odpovídá symbolickému jménu metody knihovny DLL. Druhá ukázka využívá parametr **EntryPoint**.

```
using System.Runtime.InteropServices;

public class K32Wrapper {
  [ DllImport ("kernel32.dll") ]
  public extern static void Sleep(uint msec);
  [ DllImport ("kernel32.dll", EntryPoint = "Sleep") ]
  public extern static void Doze(uint msec);
  [ DllImport ("user32.dll") ]
  public extern static uint MessageBox(int hwnd, String m,
                                       String c, uint flags);
  [
    DllImport("user32.dll", EntryPoint="MessageBoxW",
                ExactSpelling=true, CharSet=CharSet.Unicode)
  ]
  public extern static uint UniBox(int hwnd, String m,
                                   String c, uint flags);
}
```

Když voláte metody, které pracují s textovými řetězci, budete muset provést nastavení znakové sady Unicode/ANSI buď u samotné metody, nebo u typu, jenž tuto metodu zapouzdřuje. Zvolení vhodného nastavení je důležité proto, aby mohly být textové řetězce náležitě převedeny a následně použity neřízeným kódem. Parametr **CharSet** atributu **DllImport** vám umožňuje určit styl práce s řetězci. Můžete použít znakovou sadu Unicode (**CharSet.Unicode**), nebo ANSI (**CharSet.Ansi**), anebo můžete nařídit, aby se o vhodné nastavení způsobu práce s textovými řetězci postaral operační systém řady Windows (přesněji jde o systémy Windows NT®, Windows 2000, Windows XP, Windows 9x a Windows Millenium Edition (Me)). V posledním případě je parametr **CharSet** nastaven na hodnotu **CharSet.Auto**. Použití **CharSet.Auto** je podobné psaní kódu v jazyce C pro Win32 pomocí datového typu **TCHAR**, vyjma toho, že typ znakové sady a API je určován až za běhu aplikace, a ne v režimu kompilace. Tak je možné, aby aplikace pracovala efektivně na všech verzích systému Windows.

Na platformě Windows existuje varieta dekorovaných jmenných schémat, která označují volací konvence a znakové sady. Když je parametr **CharSet** nastaven na hodnotu **CharSet.Auto**, symbolické jméno funkce bude opatřeno sufixem W nebo A, podle toho, zdali je běhovým prostředím používána znaková sada Unicode nebo ANSI. Navíc, když nebude nalezeno symbolické jméno funkce, běhové prostředí aplikuje na jméno funkce volací konvenci **stdcall** (kupříkladu, název funkce **Sleep** se změní na **_Sleep@4**). Symbolické dekorování jmen můžete potlačit, když použijete parametr **ExactSpelling** atributu **DllImport**.

Nakonec, když voláte funkce Win32, které používají návratové hodnoty **HRESULT** ve stylu COM, máte dvě možnosti. Standardně technologie P/Invoke nahlíží na hodnoty **HRESULT** jako na 32bitová celá čísla, která funkce vrací, a která musejí být manuálně zpracována programátorem. Lepším způsobem, jak volat takovéto funkce, je předat atributu **DllImport** parametr **TransformSig=true**. Poté bude P/Invoke pracovat s uvedenými 32bitovými čísly jako s hodnotami **COM HRESULT**, což v případě neúspěšného volání funkce způsobí vygenerování výjimky **COMException**.

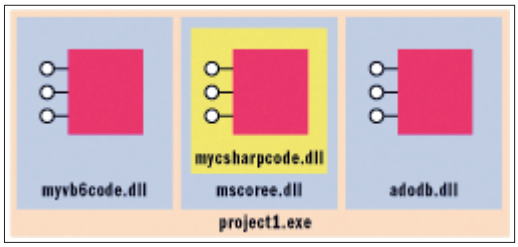
- Poznámka: Parametr **TransformSig** byl v ostré verzi Visual Studia .NET přejmenován na **PreserveSig**.

Následující výpis programového kódu ukazuje, že volání funkce **OLE32Wrapper.CoImpersonateClient1** vyžaduje, aby programátor manuálně zkontroloval výsledek práce funkce, a tedy aby pracoval přímo s hodnotou **HRESULT**.
Na druhou stranu, funkce **OLE32Wrapper.CoImpersonateClient2** používá parametr **TransformSig**, což znamená, že společné běhové prostředí převede neúspěšné hodnoty **HRESULT** do podoby výjimek **COMException**, aniž by byl nutný jakýkoliv zásah programátora. V případě funkce **OLE32Wrapper.CoImpersonateClient2** je návratovou hodnotou **void**, což znamená, že není očekávána žádná návratová hodnota, kterou by bylo možné použít. Kdyby ovšem funkce byla uzpůsobena pro navracení hodnot jistého datového typu (povězme typu **double**), P/Invoke by předpokládala, že níže umístěná nativní funkce přijala dodatečný parametr odkazem (podobně pracuje i COM atribut **[retval]**). K takovémuto mapování dochází, jenom když je parametr **TransformSig** nastaven na hodnotu **true**.

```
using System.Runtime.InteropServices;
public class OLE32Wrapper {
  [
    DllImport("ole32.dll",
              EntryPoint="CoImpersonateClient")
  ]
  public extern static int CoImpersonateClient1();
  [
    DllImport("ole32.dll",
              EntryPoint="CoImpersonateClient",
              TransformSig=true)
  ]
  public extern static void CoImpersonateClietn2();
}
```

KONVERZE TYPŮ

Když je realizováno volání metody uvnitř nebo vně běhového prostředí, parametry jsou neustále ukládány na volací zásobník. Tyto parametry představují instance typů, které mohou existovat v řízeném i neřízeném světě. Klíčem k porozumění interoperability je poznání, že jakákoliv „hodnota“ disponuje dvěma typy: řízeným a neřízeným. Mnohem důležitější je ovšem skutečnost, že některé řízené typy jsou izomorfní, což znamená, že v případě, kdy je nutné předat instanci takovéhoto typu mimo běhové prostředí, není nutno provádět žádné specifické konverze. Na druhou stranu, mnoho typů není izomorfních. Aby se i jejich instance mohli domluvit s nativním kódem, musejí se podrobit jistým konverzním procesům.

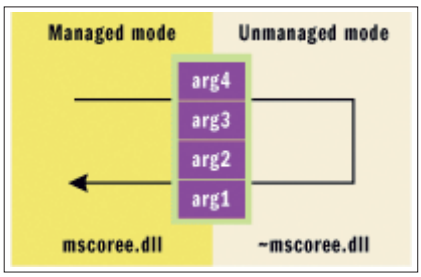


Obr. 13.2: Neizomorfní parametry

Tab. 13.1 přináší seznam základních izomorfních a neizomorfních typů. Jak již bylo zmíněno, při volání externí rutiny, která pracuje pouze s parametry izomorfních typů, nejsou vyžadovány žádný konverze. Obě metody (volající i volaná) mohou sdílet rámec zásobníku, a to i navzdory faktu, že na jedné straně běží neřízený programový kód. Pokud ale alespoň jeden parametr je neizomorfního typu, rámec zásobníku musí být zorganizován do formátu, jenž je kompatibilní s nativním světem. Konverze daného typu se může uskutečnit i v opačném směru, pro hodnotu parametru, která je předávána z volané funkce zpětně do funkce volající. Naštěstí, kompilátor jazyka C# dovede rozeznat směr proudění parametrů, jenž je založen na použití klíčových slov **ref** a **out**.

Tab. 13.1: Přehled izomorfních a neizomorfních typů

JMÉNO TYPU	TYP	POPIS
Single	Izomorfní	Organizován jako nativní typ
Double	Izomorfní	Organizován jako nativní typ
SByte	Izomorfní	Organizován jako nativní typ
Byte	Izomorfní	Organizován jako nativní typ
Int16	Izomorfní	Organizován jako nativní typ
Uint16	Izomorfní	Organizován jako nativní typ
Int32	Izomorfní	Organizován jako nativní typ
Uint32	Izomorfní	Organizován jako nativní typ
Int64	Izomorfní	Organizován jako nativní typ
Uint64	Izomorfní	Organizován jako nativní typ
Jednorozměrná pole izomorfních typů	Izomorfní	Organizován jako nativní typ
Všechna ostatní pole	Neizomorfní	Organizován jako rozhraní nebo safearray
Boolean	Neizomorfní	VARIANT_BOOL nebo Win32 BOOL
Char	Neizomorfní	Win32 WCHAR nebo CHAR
String	Neizomorfní	Win32 LPWSTR/LPSTR nebo BSTR
Object	Neizomorfní	VARIANT (pouze COM Interop)



Obr. 13.2: Neizomorfní parametry

Způsob organizování parametru (nebo datového členu či struktury) můžete ovlivňovat pomocí atributu **MarshalAs**. Tento atribut říká, který neřízený typ by měl být prezentován světu mimo MSCOREE. Atribut **MarshalAs** můžete přinejmenším použít pro zjištění odpovídajícího nativního typu pro daný parametr nebo datový člen. Společné běhové prostředí zvolí pro většinu typů standardní nastavení. Předem určená nastavení však můžete překrýt právě pomocí atributu **MarshalAs**. Kupříkladu, následující výpis programového kódu znázorňuje, jak lze použít atribut **MarshalAs** pro mapování typu **System.String** na jeden ze čtyř běžných formátů, které používá Win32 API. Jenom pro zajímavost: Operace se všemi parametry (kromě parametru **UnmanagedType.LPWStr**) vyústí do kopírování vytvořeného řetězce, čímž bude vyhověno potřebám determinovaného nativního formátu.

```
using System.Runtime.InteropServices;

public class FooBarWrapper {
// this routine wraps a native function declared as
// void _stdcall DoIt(LPCWSTR s1, LPCSTR s2, LPTST s3, BSTR s4);

  [DllImport("foobar.dll") ]
  public static extern void DoIt(
    [MarshalAs(UnmanagedType.LPWStr)] String s1,
    [MarshalAs(UnmanagedType.LPStr)] String s2,
    [MarshalAs(UnmanagedType.LPTStr)] String s3,
    [MarshalAs(UnmanagedType.BStr)] String s4
  );
}
```

Atribut **MarshalAs** vám dovoluje kontrolovat mapování typů, které je založeno na modelech „datový člen vs. datový člen“ a „parametr vs. parametr“. Kromě toho však můžete ovlivňovat také reprezentaci struktur a tříd na nižší úrovni. Exaktní paměťové rozvržení struktur a tříd můžete kontrolovat prostřednictvím atributů **StructLayout** a **FieldOffset**, což je užitečné zejména pro struktury, které jsou předávány vně běhového prostředí. Na následujících řádcích můžete vidět strukturu vytvořenou v jazyce C#.

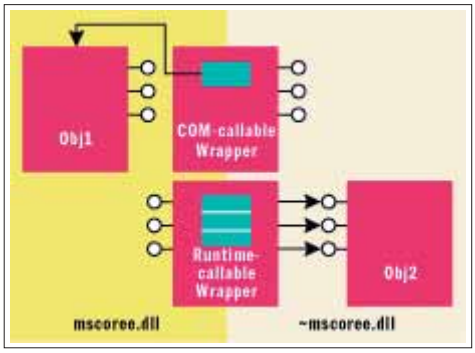
```
using System.Runtime.InteropServices;
[ StructLayout(LayoutKind.Sequential) ]
public struct PERSON_REP {
  [ MarshalAs(UnmanagedType.BStr) ]
  public String name;
  public double age;
  [ MarshalAs(UnmanagedType.VariantBool) ]
  bool dead;
}
```

Tento kód je porovnatelný s definicí v COM IDL.

```
struct PERSON_REP {
  BSTR name;
    public double age;
    VARIANT_BOOL dead;
};
```

RCW A CCW

Když předáváte jiné objektové reference než **System.String** a **System.Object**, standardně je prováděna konverze mezi objektovými referencemi společného běhového prostředí (CLR) a COM objektovými referencemi. Jak demonstruje obr. 13.3, když dochází ke konverzi reference na CLR objekt mimo hranic MSCOREE, je vytvořen tzv. COM **Callable Wrapper** (**CCW**), jenž se chová jako prostředník pro práci s CLR objektem. Podobně, když dojde k předání COM reference do společného běhového prostředí, je vytvořen tzv. **Runtime Callable Wrapper** (**RCW**), jenž je zase zástupcem COM objektu. V obou případech bude zrozený zástupce implementovat všechna rozhraní objektu, jenž se nachází na „druhé straně". Navíc, prostředník se bude snažit provádět mapování COM a CLR idiomů, mezi než patří rozhraní **IDispatch**, stálost objektů a události pro příslušné programové konstrukce v jiných technologiích.

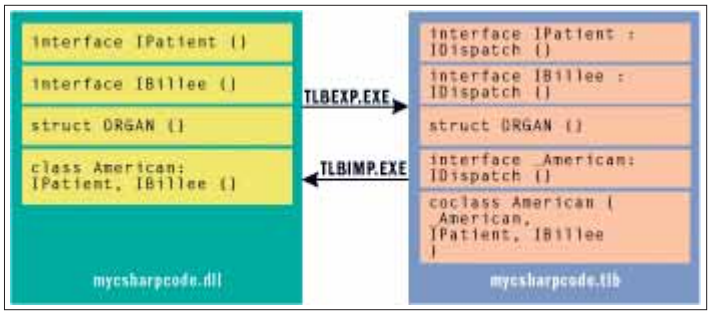


*Obr. 13.3: Architektura **RCW** a **CCW***

U rozhraní, která překlenují hranice MSCOREE přes RCW nebo CCW, se společné běhové prostředí spoléhá na kolekci anotací, které přidává k definici řízených rozhraní. Tyto anotace slouží pro potřeby níže umístěné konverzní vrstvy, která může pomocí nich provádět přetypovávání hodnot různých datových typů. Další aspekty, které musejí být definovány, zahrnují UUID, mapování virtuálních funkčních tabulek, zacházení s rozhraním **IDispatch** a převádění polí. Tyto aspekty jsou přidány k definici řízeného rozhraní použitím atributů z jmenného prostoru **System.Runtime.InteropServices**. V případě absence uvedených atributů bude společné běhové prostředí používat standardní nastavení pro určité rozhraní nebo metodu. Plánujete-li používat nově vytvořená řízená rozhraní vně společného běhového prostředí, měli byste všechny relevantní atributy u všech rozhraní explicitně deklarovat.

TLBIMP A TLBEXP

Převod nativních definicí typů COM (jako jsou struktury, rozhraní atd.) do společného běhového prostředí může být realizován manuálně, což je v některých případech nevyhnutné, zejména, když není k dispozici odpovídající typová knihovna TLB. Převod typových definicí v opačném směru je jednodušší, za což můžeme vděčit všudypřítomné reflexi, ovšem opět platí, že lepší produktivity dosáhnete pomocí specializovaného nástroje než ruční prací. Společné běhové prostředí je doprovázeno programovým kódem, který za předpokladu přítomnosti příslušných COM typových knihoven, odvádí v procesu konverze skvělou práci. Převod mezi typovými knihovnami a assembly společného běhového prostředí může být uskutečňován prostřednictvím třídy **System.Runtime.InteropServices.TypeLibConverter**. Metoda **ConvertAssemblyToTypeLib** provádí čtení assembly a na základě získaných informací sestaví typovou knihovnu, která obsahuje odpovídající COM typové definice. Jakékoliv pomůcky konverzního procesu (jako třeba použití atributu **MarshalAs**) se musejí objevit v podobě uživatelských atributů u rozhraní, metod, datových členů a parametrů ve zdrojových typech. Metoda **ConvertTypeLibToAssembly** zabezpečuje čtení typové knihovny COM a generování assembly, která obsahuje příslušné definice řízených datových typů. SDK (Software Development Kit) je dodáván se dvěma nástroji (TLBIMP.EXE a TLBEXP.EXE), které zapouzdřují volání uvedených metod do rozhraní příkazového řádku, které je vhodné pro použití s NMAKE. Vzájemný vztah mezi těmito nástroji přibližuje obr. 13.4.



*Obr. 13.4: Vztah mezi nástroji **TLBIMP** a **TLBEXP***

Ve všeobecnosti lze říci, že je jednodušší nejprve definovat typy v řízeném programovacím jazyce, a až poté sestavit typovou knihovnu. Uvažujme třeba o následujícím výpisu zdrojového kódu.

```
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

[assembly: Guid("4c5025ef-3ae4-4128-ba7b-db4fb6e0c532") ]
[assembly: AssemblyVersion("2.1") ]

namespace AcmeCorp.MathTypes {
  [
    Guid("ddc244a4-c8b3-4c20-8416-1e7d0398462a"),
    InterfaceType(ComInterfaceType.InterfaceIsIUnknown)
  ]
  public interface ICalculator
  {
    double CurrentValue { get; }
    void Clear();
    void Add(double x);
    void Subtract(double x);
    void Multiply(double x);
    void Divide(double x);
  }
}
```

Kdybyste chtěli tento kód použít v pseudo-IDL souboru, mohli byste jej přeložit pomocí kompilátoru csc.exe a prostřednictvím nástroje tlbexp.exe byste mohli vyprodukovat typovou knihovnu, která by byla co do funkčnosti srovnatelná s knihovnou, jejíž kód by byl uložen ve skutečném IDL souboru.

```
[
  uuid(4C5025EF-3AE4-4128-BA7B-DB4FB6E0C532),
  version(2.1)
]
library AcmeCorp_MathTypes
{
  importlib("stdole2.tlb");
  [
    object,
    uuid(DDC244A4-C8B3-4C20-8416-1E7D0398462A),
    oleautomation,
    custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
           "AcmeCorp.MathTypes.ICalculator")
  ]
  interface ICalculator : IUnknown {
    [propget]
    HRESULT CurrentValue([out, retval] double* pRetVal);
    HRESULT Clear();
    HRESULT Add([in] double x);
    HRESULT Subtract([in] double x);
    HRESULT Multiply([in] double x);
    HRESULT Divide([in] double x);
  }
}
```

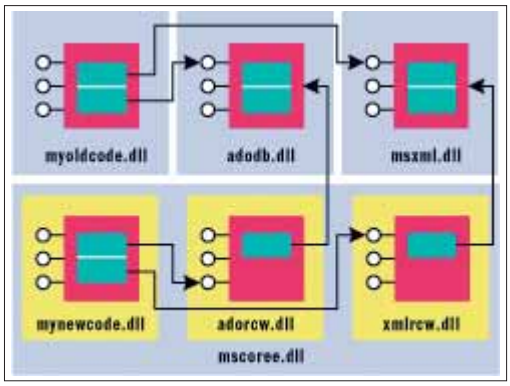
REGASM

Někteří vývojáři chtějí, aby byly jejich řízené třídy přístupné přes COM funkci **CoCreateInstance**. Pro fungování takovéhoto scénáře je nutné vložit do systémového registru některé údaje, které uskutečňují spojení COM CLSID a assembly. A právě k tomuto účelu slouží nástroj REGASM.EXE, jenž přijímá assembly jako argument a pro každou veřejnou třídu uloží do registru specifické informace. Údaj **InprocServer32** ukazuje na soubor MSCOREE.DLL, který vystupuje jako COM průčelí pro vaše řízené objekty. Měly byste vědět, že použití nástroje REGASM.EXE není nevyhnutelně nutné, protože můžete docela snadno hostovat společné běhové prostředí z nativního kódu a používat standardní aplikační doménu (AppDomain) pro načtení typů a založení instancí jakékoliv třídy. Níže uvedená programová ukázka předvádí použití třídy **System.Collections.Stack** z jazyka Visual Basic 6.0. Tato činnost může být dále automatizována pomocí tzv. monikeru. Informační zdroje na adrese *http://discuss.develop.com/archives/wa.exe?A2=ind0008&L=DOTNET&P=R63059* charakterizují „.NET moniker“, jenž dovoluje vytvářet instance řízených typů pomocí klasické COM metody **CoGetObject** nebo metody **GetObject** z Visual Basicu 6.0.

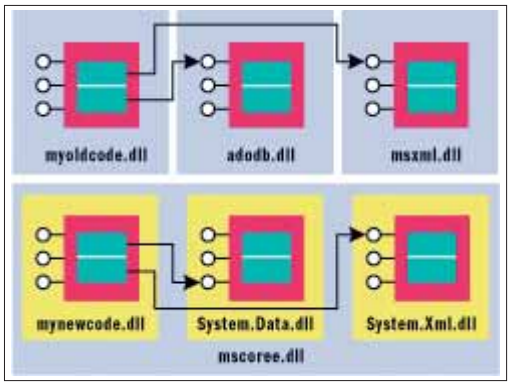
```
' needs to reference mscorlib.tlb and mscoree.tlb
Dim rt As mscoree.CorRuntimeHost
Dim unk As IUnknown
Dim ad As ComRuntimeLibrary.AppDomain
Dim s As ComRuntimeLibrary.Stack

Private Sub Form_Load()
  Set rt = New mscoree.CorRuntimeHost
  rt.Start
  rt.GetDefaultDomain unk
  Set ad = unk
  Set s = ad.CreateInstance("mscorlib", _
                            "System.Collections.Stack").Unwrap
  s.Push "Hello"
  s.Push "Goodbye"
  s.Push 42
  MsgBox s.Pop()
  MsgBox s.Pop()
  MsgBox s.Pop()
End Sub
```

STRATEGIE

Co se týče portace nativního kódu do světa společného běhového prostředí, můžeme hovořit o třech základních strategiích. Obr. 13.5 ilustruje částečnou portaci, v rámci které je zdrojový kód mechanicky převeden z neřízeného programovacího jazyka (jakým je třeba Visual Basic 6.0) do prostředí řízeného programovacího jazyka (kupříkladu Visual Basic .NET nebo C#).



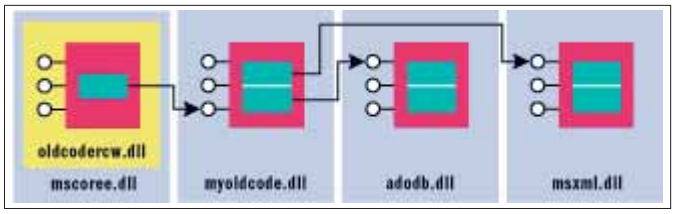
Obr. 13.5: Částeční portace do řízeného jazyka platformy .NET



Obr. 13.6: Úplná portace zdrojového kódu do prostředí programovacího jazyka platformy .NET

Obr. 13.6 přibližuje úplnou portaci, v níž je zdrojový kód kompletně přepsán v řízeném programovacím jazyce, aniž by bylo nutné se spoléhat na jakékoliv nativní subkomponenty. Tento přístup je založen na existenci řízených verzí všech podřízených nativních knihoven. Důležitá je však také vaše ochota provést přizpůsobení zdrojového kódu pro potřeby řízeného programovacího jazyka a příslušných řízených knihoven. Přestože tato cesta postupu vyžaduje nejvíc úsilí, má také své výhody, mezi něž můžeme zařadit realizaci menšího množství konverzních operací a také bohatší a lépe navržené knihovny.

Poslední varianta je znázorněna na obr. 13.7. Jedná se o migrační scénář, podle něhož není nativní zdrojový kód nijak upravován, ale je vytvořeno spojení mezi ním a řízeným prostředím pomocí technik interoperability. Tento přístup nevyžaduje existenci řízených verzí podřízených nativních knihoven a je jistě zřejmé, že ani vývojáři nemusejí provádět modifikace stávajícího programového kódu. Tato varianta je nejmíň náročná a dokonce může skýtat také výhodu menšího množství konverzí, které jsou ovlivněny poměrem volání funkcí podřízených knihoven jednotlivými metodami.



Obr. 13.7: Spolupráce nativního a řízeného kódu

Ačkoliv byste mohli vždy aplikovat technologie jako XML nebo SOAP, společné běhové prostředí nabízí vývojářům rozmanité spektrum možností pro opětovné použití programového kódu ze světů COM a Win32 v novém prostředí jazyka C# a webových služeb.

O AUTOROVI

Don Box je spoluzakladatelem společnosti DevelopMentor, která se zabývá realizací počítačových školení a podporou počítačového průmyslu. Don je autorem knížky Essentials COM a spoluautorem publikace *Effective COM and Essential XML* (obě vydalo nakladatelství Addison-Wesley). Je rovněž spoluautorem specifikace W3C SOAP. Text této kapitoly byl přebrán z první kapitoly knihy *.NET programming with C# (Programujeme .NET aplikace s jazykem C#)*, jejímiž autory jsou Don Box a Ted Patison.

Dodatek: Obsah CD "Přecházíme na .NET"

.NET F/X

- Microsoft Programming Language
  (Programovací jazyky společnosti Microsoft)
- Migrating Native Code to the .NET CLR
  (Migrace nativního kódu do společného běhového prostředí platformy .NET)
- Building an N-Tier Application in .NET
  (Budování N-vrstvých aplikací v .NET)
- Calling a .NET Component from a COM Component
  (Volání .NET komponenty z COM komponenty)
- Calling COM Components from .NET Clients
  (Volání COM komponent z .NET aplikací)
- Common .NET Libraries for Developer
  (Společné .NET knihovny pro vývojáře)
- Comparing System.Xml in Visual Studio .NET to Microsoft.XMLDOM in Visual Studio 6.0
  (Porovnávání System.Xml ve Visual Studiu .NET a Microsoft.XMLDOM ve Visual Studiu 6.0)
- Designing for Web or Desktop?
  (Navrhujete pro Web nebo systém Windows?)
- Determining When to Use Windows Installer Versus XCOPY
  (Jak určit, kdy použít rozmísťování aplikací pomocí služby Windows Installer nebo pomocí příkazu XCOPY)
- Inheritance from a Base Class in Microsoft .NET
  (Dědění z bázové třídy v Microsoft NET)
- Interacting with Message Queues
  (Pracujeme s frontami zpráv)
- Introduction to .NET
  (Úvod do .NET)
- Migrating from the SOAP Toolkit to Web Services
  (Přechod od SOAP Toolkit k webovým službám)
- Structuring a .NET Application For Easy Deployment
  (Strukturování .NET aplikace pro snadné rozmísťování)
- Understanding and Using Assemblies and Namespaces in .NET
  (Porozumění pojmům assembly a jmenný prostor a jejich praktická aplikace)
- Using ADO.NET
  (Použití ADO.NET)
- Using COM+ Services in .NET
  (Použití COM+ služeb v .NET)
- Using Web Services Instead of DCOM
  (Použití webových služeb místo DCOM)

ASP.NET

- Why ASP.NET?
  (Proč ASP.NET?)
- Converting ASP to ASP.NET
  (Přenos ASP aplikací do prostředí ASP.NET)
- Migrating to ASP.NET: Key Considerations
  (Přechod k ASP.NET: Klíčové faktory)
- Jump Start Your Web Site Development with the ASP.NET Starter Kits
  (Rychlý začátek vývoje vašeho webového sídla s ASP.NET Starter Kits)
- HTML for Beginners
  (HTML pro začátečníky)
- JSP to ASP.NET Migration Guide
  (Migrační příručka pro přechod z JSP na ASP.NET)
- The ASP.NET HTTP Runtime
  (Běhové prostředí ASP.NET HTTP)
- Building and Configuring More Secure Web Sites
  (Vytváření a konfigurace bezpečnějších webových sídel)
- Deploying an ASP.NET App Using Visual Studio .NET
  (Rozmísťování ASP.NET aplikace pomocí Visual Studia .NET)
- Deciding When to Use the DataGrid, DataList or Repeater
  (Rozhodování, kdy použít prvky DataGrid, DataList nebo Repeater)
- PHP Migration Guide
  (Migrační příručka jazyka PHP)
- How to Share Session State Between Classic ASP and ASP.NET
  (Jak sdílet stav relace aplikace mezi klasickým ASP a ASP.NET)
- Migrating from ColdFusion to ASP.NET
  (Migrace z ColdFusion na ASP.NET)
- Web Applications Technology Map
  (Technologická mapa webových aplikací)
- Page Object Model Paper
  (Model objektu Page)
- Introduction to ASP.NET and Web Forms
  (Úvod do ASP.NET a Web Forms)

C#

- C++ → C#: What You Need to Know to Move from C++ to C#
  (Co potřebujete vědět pro přechod z C++ k C#)

C++

- Interoperability between Managed and Native C++
  (Interoperabilita mezi řízeným a nativním C++)
- Quake II .NET
  (.NET verze počítačové hry Quake II)
- UNIX Code Migration Guide
  (Migrační příručka kódu systému UNIX)
- Managed Extensions for C++: Migration Guide
  (Řízená rozšíření pro C++: Migrační příručka)

J#

- Moving Java Applications to .NET
  (Přenášíme Java aplikace do .NET prostředí)
- Moving from WFC to the .NET Framework
  (Přenášíme aplikace z WFC na platformu .NET Framework)

Visual Basic

- The Best Gets Better
  (Vylepšování dokonalosti)
- Object-Oriented Programming in Visual Basic .NET
  (Objektově orientované programování v jazyce Visual Basic .NET)
- Debugging in Visual Basic .NET
  (Jak na ladění aplikací v jazyce Visual Basic .NET)
- Windows Forms Data Binding and Objects
  (Datové vazby a objekty pro přístup k datům v knihovně Windows Forms)
- Using Inheritance in Windows Forms Application
  (Použití dědičnosti v standardních aplikacích pro systém Windows)
- Deployment Changes in Visual Basic .NET
  (Změny při realizování distribuce aplikací v jazyce Visual Basic .NET)
- Creating Classes in Visual Basic .NET
  (Vytváření tříd v jazyce Visual Basic .NET)
- Creating Components in .NET
  (Vytváření komponent v jazyce Visual Basic .NET)
- Creating a Windows Form User Control
  (Vytváření uživatelských ovládacích prvků)
- Data Binding with Windows Forms and ADO.NET
  (Vytváření datových vazeb s Windows Forms a ADO.NET)
- Differences Between Visual Basic 6.0 and .NET Controls
  (Rozdíly mezi ovládacími prvky jazyka Visual Basic 6.0 a .NET ovládacími prvky)
- Distributed Transactions in Visual Basic .NET
  (Distribuované transakce v jazyce Visual Basic .NET)

- Error Handling in Visual Basic .NET
  (Ošetřování chyb v jazyce Visual Basic .NET)
- Getting Started with Windows Forms
  (Začínáme s Windows Forms)
- Inheritance and Interfaces
  (Dědičnost a rozhraní)
- Managing Versions of an Application
  (Řízení verzí aplikace)
- Overloading Methods in Visual Basic .NET
  (Přetěžování metod v jazyce Visual Basic .NET)
- Performing Drag-and-Drop Operations
  (Uskutečňování operací typu Drag-and-Drop („táhni a pusť“))
- Raising Events and Responding to Events
  (Vyvolávání událostí a odpovídání na ně)
- Replacing API Calls with .NET Framework Classes
  (Nahrazení volání API použitím tříd platformy .NET Framework)
- Using ActiveX Controls with Windows Forms in Visual Studio .NET
  (Použití ovládacích prvků ActiveX s Windows Forms ve Visual Studiu .NET)
- Variable and Method Scope in Microsoft .NET
  (Obor proměnné a metody v Microsoft .NET)
- Working with MDI Applications and Creating Menus
  (Pracujeme s MDI aplikacemi a vytváříme nabídky)
- Preparing Your Visual Basic 6.0 Applications for the Upgrade to Visual Basic .NET
  (Příprava vašich aplikací napsaných v jazyce Visual Basic 6.0 pro inovaci na Visual Basic .NET)
- Upgrading Applications Created in Previous Versions of Visual Basic
  (Inovace aplikací, které byly vytvořeny v předcházejících verzích jazyka Visual Basic)
- The Transition from Visual Basic 6.0 to Visual Basic.NET
  (Přechod z jazyka Visual Basic 6.0 na jazyk Visual Basic .NET)
- Getting Back Your Visual Basic 6.0 Goodies
  (Opětovné získání lahůdek jazyka Visual Basic 6.0)
- Visual Basic .NET Coding Standards
  (Standardy pro psaní kódu v jazyce Visual Basic .NET)
- Visual Basic .NET Internal
  (Vnitřní znaky jazyka Visual Basic .NET)
- A Framework for Upgrading Your Enterprise Application to .NET
  (Systém pro inovaci vašich podnikových aplikací do prostředí platformy .NET)
- ArtinSoft

Webcasts

- Introduction to ASP.NET
  (Úvod do ASP.NET)

- Visual Basic Series Part 1-Moving to Visual Basic.NET
  (Visual Basic Seriál: Část 1. – Přecházíme na Visual Basic .NET)

- Visual Basic Series Part 2-Upgrading Windows Forms Applications to Visual Basic.NET
  (Visual Basic Seriál: Část 2. – Inovace aplikací pro systém Windows pomocí jazyka Visual Basic .NET)

- Visual Basic Series Part 5-Migrating to Visual Basic.NET Part I
  (Visual Basic Seriál: Část 5. – Přecházíme na Visual Basic .NET Část I.)

- Visual Basic Series Part 6-Migrating to Visual Basic.NET Part II
  (Visual Basic Seriál: Část 6. – Přecházíme na Visual Basic .NET Část II.)

Autor překladu: ***Ján Hanák***

Autor překladu této brožury pro vývojáře se zabývá programováním a vývojem aplikací pro vývojově-exekuční platformu .NET Framework společnosti Microsoft použitím programovacích jazyků Visual Basic .NET, Visual C# .NET a Visual C++ .NET. Svá důmyslná řešení doposud publikoval v odborných počítačových magazínech CHIP a PC REVUE, kde rovněž vedl několik seriálů a rubrik pro programátory, vývojáře a softwarové architekty. Rovněž napsal publikaci Visual Basic .NET 2003 - Začínáme programovat, kterou na svět v roce 2004 přivedlo vydavatelství Grada Publishing.

Kromě programování se zaměřuje také na plánování, návrh, vývoj a implementaci progresivních elektronických dokumentačních systémů, které pracují na bázi technologií Microsoft WinHelp, Microsoft HTML Help a Microsoft Help 2.0. Ve volných chvílích se zabývá programováním umělé inteligence počítačových asistentů pracujících pod křídly technologie Microsoft Agent 2.0, studuje počítačovou grafiku a aplikační programové rozhraní DirectX 9 API.

Microsoft®