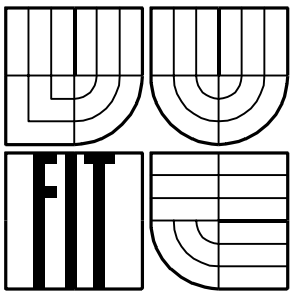


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií

.NET Framework

Semestrální práce z předmětu Vybrané partie
objektově orientovaného modelování v perzistentních systémech

Oto Škrkal

Léto 2003

Obsah

1 Úvod

Nedostatečná schopnost komunikace mezi stávajícími aplikacemi vedla firmu Microsoft v polovině devadesátých let k uvedení technologie COM (Component Object Model). Cílem této technologie bylo umožnit užší spolupráci mezi aplikacemi. Výhodou komponentové technologie byla jazyková nezávislost v binární podobě, jíž bylo dosaženo zavedením tzv. rozhraní, která zajišťovala komunikaci mezi komponentou a jejím klientem. Rozhraní však zcela zakrývají vnitřní implementaci, což znemožňuje dědičnost na úrovni zdrojových kódů. V COM technologii musela být dědičnost nahrazena jinými technikami, v nichž byla základní komponenta interně obsažena v zaobalující komponentě a ta buď musela reimplementovat všechna rozhraní vnořené komponenty (většinou tak, že pouze volala metody odpovídajícího rozhraní vnořené komponenty; technika *Containment*), nebo přímo vystavit rozhraní vnořené komponenty (technika agregace). Rozhraní navíc představuje další článek na komunikační cestě mezi klientskou aplikací a COM komponentou.

Nová platforma .NET pokračuje v trendu podpory modulárního vývoje softwaru a současně se snaží odstranit neduhy předchozích technologií. .NET Framework, srdce platformy .NET, umožňuje zjednodušit a zrychlit aplikační vývoj a poskytuje robustní a bezpečné prostředí pro běh aplikací. Jeho jádro je založeno na objektově orientovaných principech a plně je podporuje u všech programovacích jazyků určených pro platformu .NET. Každá třída nebo rozhraní definované komponentou jsou vždy dostupné v klientské aplikaci na úrovni zdrojového kódu. Díky tomu probíhá komunikace mezi klientem a komponentou přímo, bez jakýchkoliv prostředníků.

Nemalé problémy byly a jsou spojeny se správou a navracením systémových zdrojů. Mezi citlivá místa patří operační paměť, soubory, síťová spojení, datové zdroje a různé prvky grafického uživatelského rozhraní. .NET Framework se automaticky stará o správu všech systémových zdrojů. Všechny zdroje monitoruje a přestanou-li být používány, garantuje jejich navrácení.

Další pokrok přinesla platforma .NET v oblasti signalizace chyb. .NET Framework sjednotil ve všech knihovnách signalizaci chyb formou výjimek. Výjimky jsou dokonce podporovány i interně na úrovni prostředí pro běh aplikací, čímž bylo dosaženo jazykové nezávislosti generování a zpracování výjimek. Není problém vyvolat výjimku v komponentě naprogramované v jazyce C# a zpracovat ji v klientovi naprogramovaném v jazyce Visual Basic.

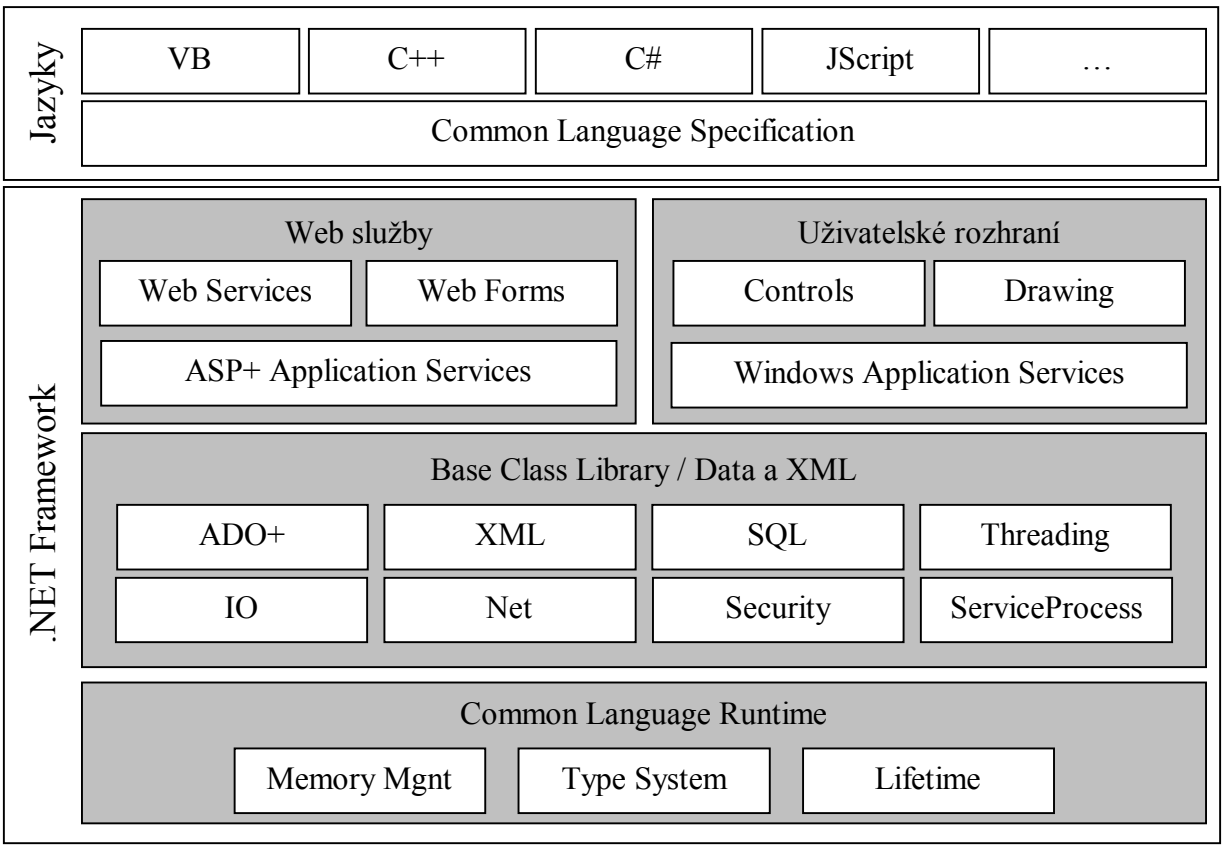
.NET Framework také realizuje společný typový systém, který je interně využíván všemi programovacími jazyky. Tím byly odstraněny problémy s předáváním parametrů libovolného typu, včetně uživatelsky definovaných, mezi komponentami napsanými v různých jazycích. Jádro .NET Framework je také schopno detekovat nepovolené manipulace s typy, které by mohly vést k havárii aplikace.

Další oblasti, ve kterých došlo ke zlepšení, jsou řízení bezpečnosti a instalace a správa softwarového vybavení. V reakci na nové bezpečnostní požadavky v souvislosti s rozmachem internetu implementuje .NET Framework nový objektový model .NET Code Access Security umožňující jemnější řízení bezpečnosti, které není založeno jen na identitě uživatele přihlášeného do systému, ale bere v úvahu také např. původ aplikace. Instalace a deinstalace softwarového vybavení je na platformě .NET zjednodušena na prosté kopírování a mazání.

Tato práce je zaměřena na jádro platformy .NET, .NET Framework. Jejím cílem je stručně popsat architekturu tohoto runtime prostředí a základy, na kterých funguje.

2 Struktura .NET Framework

.NET Framework je tvořen několika vrstvami. Na nejnižší úrovni se nachází *Common Language Runtime* (CLR) realizující základní infrastrukturu, na které je celý .NET Framework vybudován. Nad CLR se nachází knihovna *Base Class Library* a knihovna pro podporu přístupu k datům a práci s XML. Na nejvyšší úrovni .NET Framework jsou dvě knihovny usnadňující tvorbu webových aplikací a tvorbu klasických aplikací s uživatelským rozhraním.



Obr. 1 Struktura .NET Framework a navazující jazyky

Platforma .NET se snaží o jazykovou nezávislost a proto .NET Framework také obsahuje část nazvanou *Common Language Specification* (CLS), která popisuje základní vlastnosti očekávané od všech programovacích jazyků na platformě .NET. V současné době jsou podporované jazyky Visual Basic, C#, C++ a JScript.

3 Common Language Runtime

CLR je nové runtime prostředí pro běh .NET aplikací. Důvodem pro vytvoření nového runtime prostředí bylo výrazně zjednodušit vývoj aplikací, připravit robustní a bezpečné prostředí pro běh aplikací, podpořit velké množství programovacích jazyků a zjednodušit nasazení a administraci aplikací.

CLR poskytuje sadu služeb programově dostupných ve formě tříd implementujících různá veřejná rozhraní. CLR poskytuje následující služby:

- služby zapouzdřující existující Win32 funkce a služby,
- systém pro řízení životnosti objektů (*Garbage collection*),
- překladače z jazyka *Microsoft Intermediate Language* (MSIL) do nativního kódu procesoru používané při *Just-In-Time* kompilaci,

- typová kontrola založená na přítomnosti metadat popisujících všechny typy použité v aplikaci,
- podpora existujícího COM modelu,
- deklarativní bezpečnostní model.

3.1 Jazyk MSIL

Jazyk *Microsoft Intermediate Language* (MSIL) je výstupem kompilátoru každého z jazyků schopných generovat řízený kód, což je kód, o jehož provádění se stará CLR. Naopak o neřízeném kódu mluvíme u aplikací, které nejsou napsány pro platformu .NET, nebo se explicitně zřekly služeb CLR.

Jazyk MSIL je procesorově nezávislý a je velmi podobný assembleru, avšak na rozdíl od něj podporuje práci s objekty, volaní virtuálních metod, přímou manipulaci s prvky polí nebo manipulaci s výjimkami. Vzhledem k vyspělosti tohoto jazyka neexistuje žádný procesor, který by jej uměl provádět. Důvodem jeho zavedení je snaha o snadnou přenositelnost existujícího kódu mezi různými hardwarovými platformami bez nutnosti rekompilace kódu.

Aby mohla být aplikace ve formě MSIL kódu spuštěna, musí být přeložena do nativního kódu procesoru. K tomu slouží tzv. *Just-in-time* kompilátory. Zjistí-li CLR, že se snažíme spustit aplikaci přeloženou do MSIL kódu, spustí JIT kompilátor pro překlad do nativního kódu. Překlad aplikace do nativního kódu může probíhat třemi způsoby:

- překlad v době instalace – překlad je prováděn v době instalace komponenty či aplikace na počítač. Tento překlad tedy vlastně nelze označit jako Just-in-time.
- úplný Just-in-time překlad – kód MSIL je celý přeložen a optimalizován těsně před spuštěním aplikace. Výsledkem je kód podobný tomu, který by vyprodukoval překladač produkující neřízený kód. Nevýhodou tohoto přístupu je potenciálně velké zdržení před vlastním spuštěním aplikace. Na druhou stranu, JIT překladač může při překladu zjistit přesný typ procesoru a optimalizovat pro něj použité instrukce.
- ekonomický Just-in-time překlad – opět se jedná o překlad těsně před spuštěním aplikace nebo zavedením komponenty. Při tomto překladu jsou vypnuty všechny optimalizační algoritmy produkující menší a rychlejší nativní kód a je překládána pouze část kódu. Zbývající kód je přeložen až za běhu v okamžiku, kdy je potřeba. Důsledkem tohoto přístupu je zrychlení činnosti překladače, rozložení prodlevy pro zavedení aplikace do doby běhu a zmenšení požadavků na operační paměť. Ve spojení s průběžným překladem se rovněž používá technika zahození kódu, kdy je při nedostatku paměti zahozena část zřídka volaných funkcí a nahrazena rutinami, které zajistí opětovný překlad, bude-li některá ze zahozených funkcí opět potřeba.

Výsledný nativní kód je ekvivalentní kódu dnešních neřízených aplikací a tedy i jeho vykonávání je stejně rychlé.

4 Assembly

Termín assembly označuje programovou jednotku určenou k nasazení a opakovanému použití, řízení verzí a bezpečnosti. Je to kolekce jednoho nebo více souborů obsahujících kód nebo zdroje (resources) doplněná tzv. manifestem, která vytváří jeden logický celek.

Manifest, jenž je součástí každé assembly, je blok metadat zahrnující následující informace o assembly:

- Identita assembly – je tvořena třemi částmi: jménem assembly, číslem verze a tzv. kulturou (označením jazykové mutace assembly).
- Seznam souborů tvořících danou assembly. Ke každému jménu souboru je rovněž připojena kryptografická charakteristika jeho obsahu (výsledek jednocestné rozptylovací funkce, hash), která se kontroluje vždy při spuštění aplikace.
- Odkazy na další assembly – kromě jmen používaných assembly jsou zde uložena i čísla verzí těchto assembly, proti kterým byla aktuální programová jednotka linkována.
- Exportované typy a zdroje – popis veřejných typů (tříd, struktur apod.) a zdrojů (resources) exportovaných aktuální assembly.
- Bezpečnostní požadavky – lze je rozdělit na požadavky nutné pro spuštění assembly, požadavky doporučené (jejich nesplnění uživatelem může znamenat třeba omezenou funkcionalitu) a požadavky, které by nikdy neměly být přiděleny.

Zdrojový kód lze přeložit také do formy modulu obsahujícího MSIL kód. Modul však neobsahuje manifest, ale pouze metadata popisující exportované typy, metody, atributy apod. a odkazy na jiné assembly, které používá. Vzhledem k absenci manifestu musí být modul vždy součástí nějaké vícesouborové assembly obsahující manifest pro všechny soubory, které ji tvoří.

4.1 Instalace .NET aplikací a komponent

Instalace nějaké aplikace nebo komponenty na platformě .NET znamená instalaci její assembly. Platforma .NET umožňuje instalovat assembly dvojím způsobem – jako soukromou, nebo jako sdílenou. Na sdílené assembly jsou pak kladeny větší požadavky ohledně pojmenování, číslování verzí a bezpečnostních požadavků.

Jako soukromé assembly se instalují programové jednotky, které jsou interně užívány pouze v dané aplikaci a nepředpokládá se jejich širší nasazení. Proto jsou vždy instalovány do aplikačního adresáře a stačí aby jejich jméno bylo jedinečné pouze v daném adresáři. Instalace probíhá pouhým překopírováním assembly do cílového adresáře. Tento způsob instalace se nazývá *xcopy instalace*.

Jako sdílené assembly se instalují takové programové jednotky, u nichž se předpokládá, že budou využívány více aplikacemi od různých výrobců. Strukturálně jsou sdílené assembly zcela identické se soukromými. Na rozdíl od soukromých assembly musí sdílené assembly splňovat tyto požadavky:

- každá sdílená assembly musí mít jedinečné jméno,
- musí být označena korektní verzí vyjadřující úroveň kompatibility a
- musí být instalována na speciálním místě souborového systému.

Sdílené assembly jsou nejčastěji instalovány do tzv. *Global Assembly Cache* (GAC) typicky realizované adresářem „assembly/“ v systémovém adresáři. Společné umístění sdílených assembly v GAC je důvodem požadavku na jedinečnost pojmenování.

Umístění sdílených assembly v GAC, přestože není povinné, je výhodné. Sdílené assembly jsou v GAC chráněny proti smazání, neboť práce s GAC je povolena pouze administrátorovi. V GAC vedle sebe může existovat několik různých verzí jedné stejně pojmenované komponenty – tzv. *Side-by-side* existence komponenty. V běžném adresáři by tyto různé verze musely mít odlišná jména. Pokud je potřeba načíst sdílenou assembly, runtime prostředí ji hledá nejdříve právě v GAC.

4.2 Silná jména

Požadavek na jedinečnost pojmenování sdílené assembly je řešen pomocí tzv. silných (sdílených) jmen vytvářených pomocí standardních kryptografických technik. Silné jméno je tvořeno kombinací jména assembly a veřejného klíče. Veřejný klíč slouží pro ověření digitálního podpisu assembly. Silné jméno umožňuje vyřešit problém existence několika assembly od různých výrobců se stejným souborovým jménem a v redukované podobě se také využívá v odkazech.

Vzhledem k tomu, že veřejný klíč tvoří poměrně velké množství dat, v odkazech se místo něj u jména assembly uvádí tzv. *token veřejného klíče*, což je dolních osm bajtů hashe veřejného klíče. Je ověřeno, že jde o statisticky jedinečnou hodnotu dostatečnou pro vyjádření odkazu.

4.3 Ověřování identity assembly a pozdržený podpis

Pro ověřování identity assembly se používají techniky digitálního podpisu. Během vytváření manifestu je obsah každého souboru, který je součástí dané assembly, hashován a tato hodnota je pak uložena v manifestu společně s názvem souboru. Po naplnění manifestu metadaty je hashován také obsah souboru nesoucího manifest. Výsledná hodnota je digitálně podepsána soukromým klíčem a podpis je uložen do speciální části manifestu, která nebyla hashována. Do manifestu je rovněž uložen veřejný klíč, který náleží k soukromému klíči použitému pro podpis a který spolu se jménem assembly tvoří její silné jméno. Takto podepsanou assembly lze distribuovat. Digitální podpis souboru nesoucího manifest je ověřován při instalaci assembly do GAC a při každém spuštění assembly. Při spuštění se dále pomocí hash hodnot uložených v manifestu kontroluje obsah všech souborů tvořících danou assembly.

S ověřováním identity assembly souvisí také technika tzv. pozdrženého podpisu. Na rozdíl od veřejného klíče je soukromý klíč kus dat, který si každá firma musí velice pečlivě střežit a ke kterému má tedy přístup pouze velmi omezený okruh lidí. Na vývoji programových produktů se však v dané firmě mohou podílet stovky vývojářů, kteří potřebují sestavovat a ladit svoje programy i bez přístupu k soukromému klíči. To jim umožní právě technika pozdrženého podpisu.

Použitím speciálního přepínače lze sestavovacímu programu poručit, aby běžným způsobem vytvořil manifest, vložil do něj volně šiřitelný veřejný klíč, ale místo pro digitální podpis ponechal volné pro dodatečné podepsání. Takto sestavenou nepodepsanou assembly by se však za normálních okolností nepodařilo nahrát, neboť by runtime prostředí ohlásilo chybu při ověřování její identity. Proto je ještě nutné přidat tuto assembly do vnitřně udržovaného seznamu těch assembly, které nebudou kontrolovány. Po dokončení vývoje je assembly podepsána soukromým klíčem a vyjmuta ze seznamu nekontrolovaných assembly.

4.4 Verze a jejich politika

Povinné přiřazení čísla verze každé sdílené assembly hraje na platformě .NET důležitou roli. Umožňuje souběžnou existenci několika různých verzí téže komponenty v GAC a rovněž umožňuje aplikacím odkazovat se na konkrétní verzi sdílené assembly. To by mělo zajistit, že při spuštění aplikace bude vždy nahrána taková verze sdílené assembly, se kterou je aplikace plně funkční, a že tedy již nebude docházet ke známému problému, kdy po instalaci nové verze nějaké komponenty přestanou fungovat dříve instalované aplikace, které správně fungovaly s verzí starší.

Každé číslo verze se skládá ze čtyř částí – hlavního čísla, vedlejšího čísla, čísla překladu a čísla revize. Pro jejich vytváření platí poněkud přísnější pravidla než doposud. Dojde-li u

nové verze assembly ke změně na pozici hlavního nebo vedlejšího čísla, znamená to, že assembly obsahuje úpravy zpětně nekompatibilní s verzí starší. Změny zbývajících dvou čísel pak znamenají opravy, které zpětnou kompatibilitu neovlivňují (tzv. *Quick Fix Engineering*, QFE).

Jestliže má dojít k zavedení assembly, CLR hledá mezi dostupnými verzemi dané assembly nejnovější verzi, která má stejné hlavní a vedlejší číslo. Pokud takovou verzi nenajde, zavedení selže.

Implicitní chování CLR při nahrávání nemusí být vždy vyhovující (nová verze assembly je nestabilní, obsahuje bezpečnostní díru apod.) a lze je ovlivnit pomocí konfiguračních XML souborů na úrovni jedné aplikace, jedné assembly a celého systému.

- Konfigurace na úrovni jedné aplikace – konfigurační soubor je umístěn v aplikačním adresáři a ovlivňuje nahráváni assembly pouze pro danou aplikaci. Tento soubor je spravován administrátorem.
- Konfigurace na úrovni jedné assembly – vydavatel může pomocí tzv. *vydavatelské assembly s politikou* přikázat všem aplikacím užívajícím starší verzi jeho assembly, aby začaly užívat verzi novou. Vydavatelská assembly s politikou obsahuje pouze konfigurační XML soubor, je podepsána stejným klíčem jako starší produkt, jehož se konfigurace týká, a má speciální pojmenování.
- Konfigurace na globální úrovni – je realizována prostřednictvím souboru *Machine.config* umístěného v systémovém adresáři. Tento soubor má podobné možnosti jako předchozí typ, ale je spravován administrátorem, zatímco předchozí typ je instalován jako součást aktualizace dodávané výrobcem assembly.

Někdy se může stát, že nová verze assembly dodaná vydavatelem je vadná. Proto má administrátor možnost pomocí konfiguračních souborů nařídit runtime prostředí, aby ignorovalo vydavatelskou politiku instalovanou s novou verzí assembly.

4.5 Kultura assembly

Kultura assembly je vlastnost udávající, pro jaký jazyk byla konkrétní assembly vyvinuta, a je identifikována řetězcem znaků složeným z primárního a sekundárního tagu (sekundární tag je využit např. pro rozlišení britské a americké angličtiny). Kultura assembly je vedle čísla verze další vlastností umožňující násobné uložení dané assembly v GAC.

4.6 Vyhledávání assembly

Proces vyhledávání assembly je odstartován v okamžiku, kdy aplikace hodlá využít funkcionality některé externí assembly. Runtime prostředí zjistí z manifestu jméno požadované assembly, token veřejného klíče, číslo verze a kulturu. Na základě těchto informací pak začne vyhledávat assembly následujícím způsobem:

1. Nejprve prozkoumá konfigurační soubory v pořadí: aplikační konfigurační soubor, vydavatelská assembly s politikou, globální konfigurační soubor.
2. Zkontroluje, zda požadovaná assembly není již nahrána v paměti. Je-li tomu tak, je tato assembly použita a vyhledávání končí.
3. Pokud není assembly dosud nahrána v paměti, runtime prostředí se ji pokusí najít v Global Assembly Cache.
4. Selžou-li předchozí kroky, dojde k tzv. sondování (*probing*):

- Pokud konfigurační soubor obsahuje element <codebase>, runtime prostředí hledá assembly na místě nastaveném tímto elementem; není-li assembly nalezena, hledání je ukončeno a je signalizována chyba.
- Postupně je prozkoumáno několik adresářů:
  - aplikační kořenový adresář,
  - podadresář pojmenovaný podle kultury assembly,
  - podadresář pojmenovaný po jménu assembly,
  - seznam uživatelsky definovaných adresářů zadaný v konfiguračním souboru elementem <probing privatePath>.

4.7 Jmenný prostor

Veškerá funkcionalita .NET Framework je organizována v různých jmenných prostorech. Jmenné prostory, připomínající adresářovou strukturu, umožňují seskupit spolu související funkce a typy na jednom pojmenovaném logickém místě, čímž je jejich uživateli usnadněna orientace. Dále řeší konflikty v pojmenování, neboť jméno funkce či typu musí být jedinečné pouze v daném jmenném prostoru.

Pro zápis se používá tečková notace, kdy jména jednotlivých do sebe zanořených prostorů a jméno typu umístěného ve jmenném prostoru jsou oddělena tečkami.

Každá assembly může definovat kód v několika různých jmenných prostorech, ať už jsou nezávislé nebo vnořené. Stejně tak může být jeden jmenný prostor tvořen více než jednou assembly. Rozdělit jmenný prostor do více assembly je vhodné v okamžiku, kdy je tento prostor příliš rozsáhlý a je zřejmé, že některé funkce budou používány nezávisle na ostatních. V případě nerozdělení jmenného prostoru by se při použití takovéto funkce zbytečně nahrávala celá assembly.

5 Společný typový systém (Common Type System)

Aby mohla firma Microsoft dostát myšlence hluboké jazykové integrace, musela realizovat společný typový systém (*common type system*, CTS) nezávislý na jazykové implementaci. Vedle společného typového systému definovaného na systémové úrovni musely být dále realizovány také typová bezpečnost a obecný objektově orientovaný model.

Základem typového systému .NET Framework je plná objektovost – instance každého typu je objekt. Plná objektovost není ve světe programovacích jazyků žádnou novinkou. Přínosem typového systému v .NET Framework je však jeho výrazná jednoduchost a výkonnost, které bylo dosaženo zavedením dvou skupin typů – hodnotových a referenčních.

Hodnotové typy jsou takové, jejichž instance v paměti vždy reprezentuje skutečné hodnoty. Naopak, instance referenčních typů neobsahují hodnoty samotné, ale udávají odkaz na místo v paměti, kde je hodnota uložena.

Základem každého typu na platformě .NET je třída System.Object, která definuje základní množinu operací pro všechny typy. Z této třídy jsou odvozeny jak typy zahrnuté do CTS, tak všechny uživatelsky definované typy. Díky společnému předku lze referenci libovolného typu přiřadit do reference typu Object.

5.1 Hodnotové typy

Hodnotové typy v .NET Framework lze rozdělit na základní hodnotové typy a uživatelsky definované hodnotové typy. Základní hodnotové typy jsou definovány v Base Class Library ve formě tříd. V každém jazyce jsou dostupné pod jiným názvem a vždy se jeví jako tzv. primitivní datové typy.

Hodnotové datové typy jsou charakteristické těmito vlastnostmi:

- Jsou vždy alokovány na zásobníku a vždy obsahují jen data uvedená v deklaraci. Neobsahují žádné ukazatele na tabulku virtuálních metod apod.
- Vždy musí být inicializovány.
- Při přiřazení jedné hodnotové proměnné druhé dochází k fyzickému kopírování obsahu.
- Všechny jsou odvozeny z třídy System.ValueType, která je téměř identická s třídou System.Object a vytváří pro hodnotové typy společnou základní funkcionalitu.
- Nepodporují dědičnost a nemohou se tedy stát základní třídou pro jiné typy.
- Neobsahují virtuální funkce.
- Ze zásobníku jsou odstraněny při ukončení metody nebo bloku kódu, kde byly definovány.

Díky svému umístění nacházejí hodnotové typy své uplatnění zejména tam, kde jde o rychlost.

5.2 Referenční typy

Referenční typy jsou velice podobné ukazatelům v jazyce C. Na rozdíl od nich jsou však vždy typově bezpečné, tj. reference může odkazovat vždy jen na objekt deklarovaného typu nebo na typ z něj odvozený.

Charakteristické vlastnosti referenčních typů jsou tyto:

- Objekt je vždy alokován na hromadě a reference obsahuje jen jeho adresu.
- Reference může mít hodnotu null a je na tuto hodnotu implicitně inicializována. Při pokusu o použití takovéto reference je vygenerována výjimka NullReferenceException.
- Hodnotu samotné reference není obvykle možné získat.
- Životnost objektů je řízena službou Garbage Collection. Proto není třeba je explicitně odstraňovat.
- Podporují jednoduchou dědičnost a polymorfismus.
- Všechny jsou přímo nebo nepřímo odvozeny z třídy System.Object.
- Přiřazením jedné reference do druhé dochází k pouhému kopírování adres. Po provedení přiřazení pak obě reference odkazují na jediný objekt.

.NET Framework ve svém typovém systému definuje tyto referenční typy:

- ukazatele – ukazatele jsou obecně tří typů: řízené, neřízené a neřízené funkční. Jejich dostupnost v jazycích je různá. Neřízené a neřízené funkční ukazatele nejsou povoleny v řízeném kódu; řízené ukazatele jsou dostupné pouze v některých jazycích.
- rozhraní (*interface*) – rozhraní jsou množiny statických členů, vnořených typů, abstraktních metod, vlastností a událostí. Neobsahují žádný kód a každá třída může implementovat jedno nebo více rozhraní.

- třídy – nejznámější typ; členy jsou spojeny v jeden logický celek. Členy tříd v .NET Framework mohou být položky, metody, vlastnosti a události.
- uživatelsky definované třídy – každá uživatelsky definovaná třída je implicitně odvozena ze System.Object, ale může být explicitně odvozena z libovolné třídy CTS, nebo jiné uživatelské třídy. Dědičnost je omezena pouze na jednoduchou, je však povoleno implementovat neomezené množství rozhraní.
- pole – v .NET Framework jsou podporována na úrovni CTS. Díky tomu jsou jednotně reprezentována ve všech jazycích. Navíc je zajištěna kontrola indexů. Prvky polí jsou vždy objekty.
- delegáti – jedná se o typově bezpečnou náhradu ukazatelů na funkce známých z jazyka C/C++. Mohou odkazovat na statické, nestatické i virtuální metody a navíc mohou udržovat seznam více funkcí, které jsou všechny postupně volány.
- boxované hodnotové typy – referenční protějšky předdefinovaných hodnotových typů.

5.3 Konverze mezi hodnotovými a referenčními typy

Přestože jsou hodnotové typy z výkonnostního hlediska velmi výhodné, občas je potřeba s nimi pracovat jako s typy referenčními. Typický je případ, kdy je potřeba hodnotový datový typ (např. celé číslo) použít v nějaké dynamické abstraktní datové struktuře (např. seznam). Třídy realizující tyto typy jsou často naprogramovány tak, že pracují s referencemi typu Object. Pokud je tedy potřeba hodnotový typ uložit do takovéto struktury, je nutné jej nejprve převést na typ referenční. .NET Framework definuje pro všechny předdefinované hodnotové typy jejich referenční protějšky. Proces převodu hodnotového typu na odpovídající referenční se nazývá *boxing*, převod referenčního typu zpět na hodnotový se nazývá *unboxing*.

Proces boxing probíhá ve třech krocích:

1. Na hromadě je alokováno potřebné místo pro uložení hodnoty proměnné, rozšířené o místo pro uložení dodatečných informací, jako je ukazatel na tabulku virtuálních funkcí.
2. Obsah proměnné se zkopíruje do alokovaného prostoru na hromadě.
3. Adresa paměti obsahující zkopírovanou hodnotu včetně dodatečných informací je uložena do reference typu Object.

Opačný proces převodu referenčního typu zpět na hodnotový, unboxing, probíhá ve dvou krocích:

1. Zkontroluje se, zda reference nemá hodnotu null. Pak se zkontroluje typ objektu, na nějž reference odkazuje, aby bylo zajištěno, že je operace unboxing přípustná a logicky správná.
2. Po kontrolách je vrácen ukazatel na hodnotu uloženou v objektu. Obsah odkazované paměti je pak většinou překopírován do hodnotové proměnné odpovídajícího typu.

5.4 Typová bezpečnost

Jedním z důležitých požadavků na typový systém je typová bezpečnost. Tu můžeme chápat také jako garanci, že nad definovanými typy nemohou být provedeny nepovolené operace. Každý objekt vytvořený v programu je striktně typový a striktně typová je i reference, která na něj odkazuje. Typový systém nepřipustí, aby reference nějakého typu odkazovala na objekt, který tohoto typu není a ani z něj nebyl odvozen. Typový systém k tomu nelze přinutit ani explicitně.

Každý typ je zodpovědný za přístupová práva ke svým členským položkám. .NET Framework definuje následující typy přístupů.

Public	Člen je přístupný veškerému kódu z libovolné assembly.
Private	Člen je přístupný pouze metodám deklarovaným v daném typu.
Family	Člen je přístupný metodám deklarovaným v daném typu a metodám odvozených tříd.
Assembly	Člen je přístupný veškerému kódu umístěnému v dané assembly.
Family and Assembly	Člen je přístupný metodám vlastní třídy a metodám tříd odvozených, pokud jsou definovány ve stejné assembly.
Family or Assembly	Člen je přístupný metodám všech odvozených tříd a dále všem metodám na úrovni aktuální assembly.

6 Správa paměti

Obecným cílem správy paměti je uvolnit systémové zdroje držené objektem a následně i paměť, kterou zabírá. To vše ve chvíli, kdy objekt již nikdo nevyužívá. K úklidu systémových zdrojů využívá správa paměti destruktory nebo jiné obdobné metody, které automaticky vyvolá v okamžiku, kdy danému objektu končí životnost a má být odstraněn z paměti. Pro řízení životnosti objektů se pak používají různé techniky.

V jazyce C++ je řízení životnosti dynamicky alokovaných objektů v podstatě ponecháno na programátorovi, který je zodpovědný za použití operátoru delete na nepotřebný objekt. V rozsáhlém kódu se však může stát, že je použití tohoto operátoru opomenuto, nebo je naopak tento operátor na jeden objekt použit vícekrát. První případ pravděpodobně povede ke ztrátě systémových zdrojů, druhý pak může vést k havárii celé aplikace.

Jazyk Visual Basic, stejně jako technologie COM, využívá pro řízení životnosti objektů počítání referencí. V této technologii jsou značným problémem kruhové reference, kdy jeden objekt ukazuje na druhý a druhý na první. Není-li problém kruhových referencí ošetřen, dojde opět k úniku paměti.

Dalším problémem při správě paměti je vzájemná nekompatibilita algoritmů a technologií alokace paměti v různých programovacích jazycích, nebo i v rámci jednoho jazyka (např. v jazyce C++). Tento problém se stane aktuálním např. v okamžiku, kdy jedna komponenta paměť alokuje a jiná, napsaná v jiném jazyce, ji má vrátit.

Na platformě .NET byla pro správu paměti zvolena technologie *Garbage Collection* (GC), která by měla řešit všechny výše uvedené problémy.

GC se automaticky postará o navrácení dynamicky alokované paměti ve chvíli, kdy o ni ze strany uživatele již není zájem. Uživatel se tedy nemusí o nic starat. Také zde není nutné řešit různé aspekty související s počítáním referencí. GC je na platformě .NET implementována jako součást runtime služeb a je společná pro všechny jazyky. Tím je zajištěna jazyková nezávislost alokace a dealokace paměti. Nevýhodou této technologie je, že paměť nemusí být navrácena ihned, jakmile není potřeba.

GC vytváří dvě vlákna. První monitoruje počty referencí na objekty a v okamžiku, kdy zjistí, že nějaký objekt již není používán, dává pokyn k odstranění objektu z hromady. Druhé vlákno zajišťuje vlastní úklid. Ještě před tím, než druhé vlákno odstraní objekt z paměti, zavolá úklidovou rutinu objektu, tzv. *finalizer*.

GC se přesouváním objektů rovněž stará o zkompaktňování hromady. Přesunutí objektů má však za následek změnu adres a proto s nimi programátor nesmí přímo pracovat (to je ovšem v řízeném kódu vždy splněno).

Výhody GC:

- Nemůže dojít k porušení ochrany paměti – ukazatele se v řízeném kódu nevyskytují a reference vždy odkazují na platné objekty, pokud nejsou null.
- Interně je řešena problematika kruhových referencí.

Nevýhody GC:

- Jistá spotřeba výpočetního výkonu nutná pro sledování a úklid objektů.
- Nedeterministické chování – není jasné, kdy přesně bude nepotřebný objekt odstraněn. Může nějakou dobu ještě setrvávat v paměti a blokovat některé systémové zdroje (databázová spojení, souborová spojení apod.).

Má-li objekt před svým zánikem provést nějaké úklidové akce, má obvykle za tímto účelem implementován destruktor. Obdobou destruktoru na platformě .NET je již výše zmíněný finalizer realizovaný metodou *Finalize*. GC ovšem nezaručuje okamžité vyvolání této metody a odstranění objektu.

Je-li potřeba dosáhnout deterministického úklidu objektu, je nutné objektu implementovat rozhraní IDisposable s metodou Dispose. Tato metoda by měla provést následující kroky:

1. Ověřit, zda nejde o opakované volání. V tom případě by se metoda pouze ukončila.
2. Uvolnit vlastní systémové zdroje.
3. Vyvolat metodu Dispose základní třídy, pokud základní třída existuje a metodu obsahuje.
4. Zakázat spuštění metody Finalize pro konkrétní objekt.

Explicitním vyvoláním metody Dispose dojde k okamžitému uvolnění systémových zdrojů držených objektem. Objekt samotný však nadále zůstává v paměti až do té doby, než jej GC uklidí.

Pokud je implementováno rozhraní IDisposable, mělo by být zajištěno, že přístup k objektu s uvolněnými zdroji vyvolá výjimku.

7 .NET a COM

Platforma .NET plně podporuje vzájemnou spolupráci s komponentami a aplikacemi starší technologie COM. Je možné využít jak stávající COM komponenty v .NET aplikacích, tak .NET komponenty v COM aplikacích. Vzájemná spolupráce mezi COM a .NET je zajištěna součástí .NET Framework nazvanou *COM Interop*.

7.1 COM komponenty v .NET aplikacích

Pro práci s COM komponentou v .NET aplikaci slouží tzv. *Runtime Callable Wrapper* (RCW). Jeho úkolem je zprostředkovat komunikaci mezi COM komponentou a .NET klientem a to tak, že se COM komponenta klientské aplikaci jeví identicky jako .NET komponenta a klientská aplikace se COM komponentě jeví jako běžná neřízená aplikace.

Pokud je k dispozici typová knihovna COM komponenty, lze RCW snadno vygenerovat pomocí automatizovaných nástrojů. Výsledkem použití těchto nástrojů je assembly, která je v klientské aplikaci použita pro přístup ke COM objektu. RCW zajistí po svém vytvoření založení COM objektu, zajistí správné provolání metod rozhraní COM objektu, při chybě signalizované chybovým kódem zajistí vygenerování odpovídající výjimky a před svým odstraněním z paměti garbage collectorem zajistí zrušení COM objektu.

Pokud chybí typová knihovna, nebo COM rozhraní komponenty není Automation kompatibilní (není implementováno rozhraní IDispatch), je třeba RCW vytvořit ručně. Zrovna tak, pokud je identita komponenty známa až za běhu programu, je vytvoření RCW poněkud složitější.

7.2 .NET komponenty v COM aplikacích

Pro začlenění .NET komponenty do COM aplikace slouží tzv. *COM Callable Wrapper* (CCW), který zaobaluje .NET objekty a vytváří aplikaci dojem, že jde o COM komponentu. Má-li CCW dobře interpretovat všechny vlastnosti .NET komponenty ve formě COM rozhraní, musí komponenta mít některé vlastnosti.

- Je vhodné, aby komponenta implementovala jedno rozhraní, které se vystaví COM klientovi.
- Všechny třídy dostupné COM klientovi musí být deklarovány jako veřejné.
- Členské položky, vlastnosti, metody a události musí být veřejné a označené atributem System.Runtime.InteropServices.ComVisible.
- Každá třída musí obsahovat implicitní konstruktor.

Assembly dostupná ve formě COM komponenty musí mít sdílené jméno a pokud možno by měla být instalována v GAC. Vzhledem k tomu, že COM technologie intenzivně využívá registry, je nutné zde pomocí speciální utility zaregistrovat potřebné informace o assembly. Dále je k dispozici utilita, která z assembly vygeneruje typovou knihovnu vyžadovanou mnohými aplikacemi.

8 Bezpečnost .NET Framework

Bezpečnost v operačních systémech a sítích je většinou založena na základě zjišťování identity uživatele. Operace prováděné uživatelem jsou pak kontrolovány podle práv přidělených administrátorem. Tento systém však pomalu přestává vyhovovat. Microsoft Transaction Server zavedl od svých prvních verzí bezpečnostní systém založený na rolích. Role sdružují uživatele s identickými privilegii nezávisle na tom, v jakých systémových skupinách se nacházejí. .NET Framework tento princip převzal a rozšířil.

Kontrola bezpečnosti začíná již při nahrávání aplikace do paměti. Probíhají základní ověření přístupu ke zdrojům formou tzv. přístupové bezpečnosti kódu (*Code Access Security*) a je zjišťována identita uživatele sloužící k řízení bezpečnosti pomocí rolí. Tyto operace fungují přes hranice procesů a počítačů, aby nemohlo snadno docházet k falešným prověřením. Přístupová bezpečnost kódu a bezpečnost založená na rolích jsou hluboce integrovány do jádra .NET Framework a intenzivně využívány ostatní infrastrukturou.

Bezpečnost na platformě .NET se dotýká oblastí:

- typové bezpečnosti,
- identity kódu,
- přístupové bezpečnosti kódu,
- povolení,
- deklarativního a imperativního řízení bezpečnosti,
- bezpečnosti založené na rolích a
- kryptografických služeb.

8.1 Typová bezpečnost

Typově bezpečné programy pracují pouze s pamětí, která byla alokována pro jejich objekty. V kontextu řízení bezpečnosti se pojem typová bezpečnost liší od téhož pojmu v kontextu programovacích jazyků a týká se pouze bezpečného přístupu k paměti objektu. Příkladem typově bezpečného kódu je třída, k jejímž soukromým členům nelze přistupovat žádným jiným způsobem. Bezpečnost je garantována pouze při přístupu přes definované rozhraní.

Během JIT kompilace probíhají verifikační procesy, které zkoumají obsah metadat v manifestu a obsah MSIL kódu a ověřují, zda je kód typově bezpečný. Pokud je kód typově bezpečný, runtime prostředí je schopno zajistit úplnou izolaci jednotlivých assembly v rámci procesu. Jejich komponenty pak mohou běžet zcela bezpečně. Jedním z hlavních cílů tohoto postupu je spolehlivost aplikace. U typově bezpečného kódu mohou runtime mechanismy zajistit, že nebude přistupovat k nativnímu kódu získanému JIT kompilací, pokud k tomu nemá speciální povolení.

8.2 Identita kódu

Identita kódu je dána tzv. evidencí, což jsou základní informace zahrnující např. místo, ze kterého kód pochází, sdílené jméno či identitu vydavatele. Na základě těchto informací může bezpečnostní politika určit, jaká práva budou specifické aplikaci nebo množině aplikací přidělena. Zmíněné informace jsou schopny poskytnout služby, které mají na starosti zavedení aplikace do paměti.

8.3 Přístupová bezpečnost kódu

Tento typ bezpečnosti umožňuje nastavit důvěryhodnost kódu v závislosti na tom, odkud kód pochází a dalších aspektech identity kódu. Lze nastavit, které operace kód s konkrétní identitou může vykonávat a které naopak vykonávat nesmí. Lze takto omezit nevhodné chování kódu, které by mohlo poškodit některý systémový zdroj. Přístupovou bezpečnost kódu mohou využívat pouze aplikace typově bezpečné.

8.4 Povolení

CLR umožňuje aplikacím vykonávat pouze ty operace, na které má jejich kód povolení. Povolení jsou logicky seskupována do množin povolení, které jsou assembly přiřazeny bezpečnostním podsystémem .NET Framework a které jednoznačně určují, co je kódu povoleno a co nikoliv. Typ přiřazené množiny povolení je určen evidencí kódu, tj. zdrojem kódu, silným jménem atd. Povolení jsou pak většinou vyžadována důvěrným kódem, což je např. takový, který pracuje se souborovým systémem.

Rozlišují se dva typy povolení – povolení pro přístup kódu a tzv. povolení identity. Povolení pro přístup kódu jsou základním nástrojem runtime prostředí v oblasti zajištění bezpečnosti řízeného kódu. Dělí se na:

- práva pro přístup k chráněným systémovým zdrojům, jako jsou soubory nebo systémové proměnné, a
- práva pro spouštění chráněných operací, jako například volání funkcí neřízeného kódu.

Pokud nějaká funkce požaduje přístup k jednomu ze zmíněných chráněných zdrojů, CLR projde zásobník, aby zjistil, odkud volání pochází a snaží se ověřit, zda všichni účastníci ve volací sekvenci mají příslušné povolení.

Povolení identity jsou založena na základě identity asociované s konkrétní assembly. CLR přiřadí tento typ povolení v době zavedení assembly do paměti a dekódování identity. Tato

povolení se uplatní v případě, kdy kód volající assembly požaduje, aby tato assembly měla určité sdílené jméno, nebo pocházela z nějakého důvěryhodného zdroje.

8.5 Deklarativní a imperativní řízení bezpečnosti

Požadavky na kontrolu bezpečnosti lze do kódu aplikace vložit deklarativním nebo imperativním způsobem. Deklarativní způsob využívá pro stanovení bezpečnostních požadavků atributy doplněné k jednotlivým částem kódu. Tyto požadavky se pak stanou součástí metadat assembly. Imperativní způsob využívá funkčních volání uvnitř aplikačního kódu a jeho výhodou je možnost programově modifikovat bezpečnostní požadavky.

8.6 Bezpečnost založená na rolích

Pro bezpečnost založenou na rolích se využívá tzv. objekt *Principal*, který vzniká z informací o identitě uživatele. Ta může být založena na základě účtů Windows, ale může být implementována i zcela jinak. Objekt Principal je vytvořen jedním z autentikačních providerů na základě informací získaných od uživatele a asociován s daným exekučním kontextem. Pak je dostupný kódu aplikace, která může ověřovat jeho hodnotu v definovaných rolích a rozhodovat o provedení operací.

8.7 Kryptografické služby

.NET Framework obsahuje sadu kryptografických objektů implementujících známé algoritmy. Implementovány jsou symetrické šifrovací algoritmy (DES, RC2, 3DES), asymetrické šifrovací algoritmy (DSA pro digitální podpis a RSA) a standardní hashovací funkce (MD5, SHA1). Tyto objekty jsou často využívány vnitřními službami .NET Framework, ale jsou rovněž dostupné i aplikačním programátorům.

Literatura

[1] Dalibor Kačmář: *Programujeme .NET aplikace ve Visual Stuidu .NET,* Computer Press, Praha, 2001.

[2] Dalibor Kačmář: *Programujeme v COM a COM+,* Computer Press, Praha, 2000.